Secrets of the JavaScript Ninja

SECOND EDITION

John Resig
Bear Bibeault
Josip Maras

SAMPLE CHAPTER

MANNING

Secrets of the JavaScript Ninja
Second Edition

by John Resig
Bear Bibeault
and Josip Maras

Chapter 7

brief contents

Part 3

Digging into objects and fortifying your code

Now that you've learned the ins and outs of functions, we'll continue our exploration of JavaScript by taking a closer look at object fundamentals in chapter 7.

In chapter 8, we'll study how to control access to and monitor our objects with getters and setters, and with proxies, a completely new type of object in JavaScript.

We'll take a look at collections in chapter 9—traditional ones such as arrays, as well as completely new types such as maps and sets.

From there, we'll move on to regular expressions in chapter 10. You'll learn that many tasks that used to take reams of code to accomplish can be condensed to a mere handful of statements through the proper use of JavaScript regular expressions.

Finally, in chapter 11, we'll show you how to structure your JavaScript applications into smaller, well-organized units of functionality called modules.

7 Object orientation with prototypes

This chapter covers

- Exploring prototypes
- Using functions as constructors
- Extending objects with prototypes
- Avoiding common gotchas
- Building classes with inheritance

You've learned that functions are first-class objects in JavaScript, that closures make them incredibly versatile and useful, and that you can combine generator functions with promises to tackle the problem of asynchronous code. Now we're ready to tackle another important aspect of JavaScript: object prototypes.

A *prototype* is an object to which the search for a particular property can be delegated to. Prototypes are a convenient means of defining properties and functionality that will be automatically accessible to other objects. Prototypes serve a similar purpose to that of classes in classical object-oriented languages. Indeed, the main use of prototypes in JavaScript is in producing code written in an object-oriented

way, similar to, but not exactly like, code in more conventional, class-based languages such as Java or C#.

In this chapter, we'll delve into how prototypes work, study their connection with constructor functions, and see how to mimic some of the object-oriented features often used in other, more conventional object-oriented languages. We'll also explore a new addition to JavaScript, the `class` keyword, which doesn't exactly bring full-featured classes to JavaScript but does enable us to easily mimic classes and inheritance. Let's start exploring.

Do you know?

- How do you test whether an object has access to a particular property?
- Why is a prototype chain important for working with objects in JavaScript?
- Do ES6 classes change how JavaScript works with objects?

7.1 Understanding prototypes

In JavaScript, objects are collections of named properties with values. For example, we can easily create new objects with object-literal notation:

```
let obj = {
  prop1: 1,
  prop2: function(){},
  prop3: {}
}
```

Assigns a simple value

Assigns a function

Assigns another object

As we can see, object properties can be simple values (such as numbers or strings), functions, and even other objects. In addition, JavaScript is a highly dynamic language, and the properties assigned to an object can be easily changed by modifying and deleting existing properties:

```
obj.prop1 = 1;
obj.prop1 = [];
delete obj.prop2;
```

prop1 stores a simple number.

Assigns a value of a completely different type, here an array

Removes the property from the object

We can even add completely new properties:

```
obj.prop4 = "Hello";
```

Adds a completely new property

In the end, all these modifications have left our simple object in the following state:

```
{
  prop1: [],
  prop3: {},
  prop4: "Hello"
};
```

When developing software, we strive not to reinvent the wheel, so we want to reuse as much code as possible. One form of code reuse that also helps organize our programs is *inheritance*, extending the features of one object into another. In JavaScript, inheritance is implemented with prototyping.

The idea of prototyping is simple. Every object can have a reference to its *prototype*, an object to which the search for a particular property can be delegated to, if the object itself doesn't have that property. Imagine that you're in a game quiz with a group of people, and that the game show host asks you a question. If you know the answer, you give it immediately, and if you don't, you ask the person next to you. It's as simple as that.

Let's take a look at the following listing.

Listing 7.1 With prototypes, objects can access properties of other objects

```
const yoshi = { skulk: true };
const hattori = { sneak: true };
const kuma = { creep: true };

assert("skulk" in yoshi, "Yoshi can skulk");
assert(!("sneak" in yoshi)), "Yoshi cannot sneak");
assert(!("creep" in yoshi)), "Yoshi cannot creep");

Object.setPrototypeOf(yoshi, hattori);

assert("sneak" in yoshi, "Yoshi can now sneak");
assert(!("creep" in hattori)), "Hattori cannot creep");

Object.setPrototypeOf(hattori, kuma);
assert("creep" in hattori, "Hattori can now creep");
assert("creep" in yoshi, "Yoshi can also creep");
```

Creates three objects, each with its own property

yoshi has access to only its own, skulk, property.

Use the Object .setPrototypeOf method to set one object as the prototype of another object.

By setting hattori as yoshi's prototype, yoshi now has access to hattori's properties.

Currently, hattori can't creep.

Sets kuma as a prototype of hattori

Now hattori has access to creep.

yoshi also has access to creep, through hattori.

In this example, we start by creating three objects: `yoshi`, `hattori`, and `kuma`. Each has one specific property accessible only to that object: Only `yoshi` can `skulk`, only `hattori` can `sneak`, and only `kuma` can `creep`. See figure 7.1.

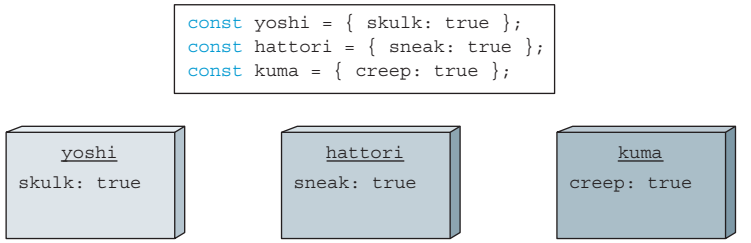

Figure 7.1 Initially, each object has access to only its own properties.

To test whether an object has access to a particular property, we can use the `in` operator. For example, executing `skulk in yoshi` returns `true`, because `yoshi` has access to the `skulk` property; whereas executing `sneak in yoshi` returns `false`.

In JavaScript, the object's prototype property is an internal property that's not directly accessible (so we mark it with `[[prototype]]`). Instead, the built-in method `Object.setPrototypeOf` takes in two object arguments and sets the second object as the prototype of the first. For example, calling `Object.setPrototypeOf(yoshi, hattori);` sets up `hattori` as a prototype of `yoshi`.

As a result, whenever we ask `yoshi` for a property that it doesn't have, `yoshi` delegates that search to `hattori`. We can access `hattori`'s `sneak` property through `yoshi`. See figure 7.2.

We can do a similar thing with `hattori` and `kuma`. By using the `Object.setPrototypeOf` method, we can set `kuma` as the prototype of `hattori`. If we then ask `hattori` for a property that he doesn't have, that search will be delegated to `kuma`. In this case, `hattori` now has access to `kuma`'s `creep` property. See figure 7.3.

It's important to emphasize that every object can have a prototype, and an object's prototype can also have a prototype, and so on, forming a *prototype chain*. The search delegation for a particular property occurs up the whole chain, and it stops only when there are no more prototypes to explore. For example, as shown in figure 7.3, asking `yoshi` for the value of the `creep` property triggers the search for the property first in `yoshi`. Because the property isn't found, `yoshi`'s prototype, `hattori`, is searched. Again, `hattori` doesn't have a property named `creep`, so `hattori`'s prototype, `kuma`, is searched, and the property is finally found.

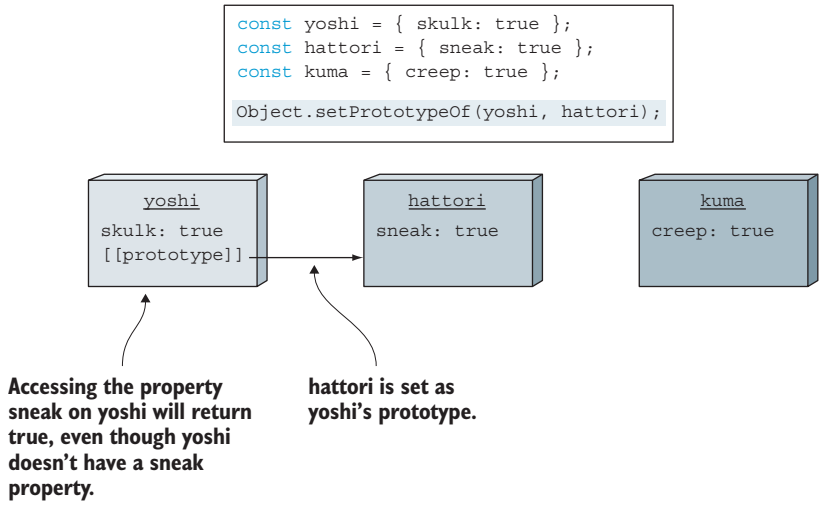

Figure 7.2 When we access a property that the object doesn't have, the object's prototype is searched for that property. Here, we can access `hattori`'s `sneak` property through `yoshi`, because `yoshi` is `hattori`'s prototype.

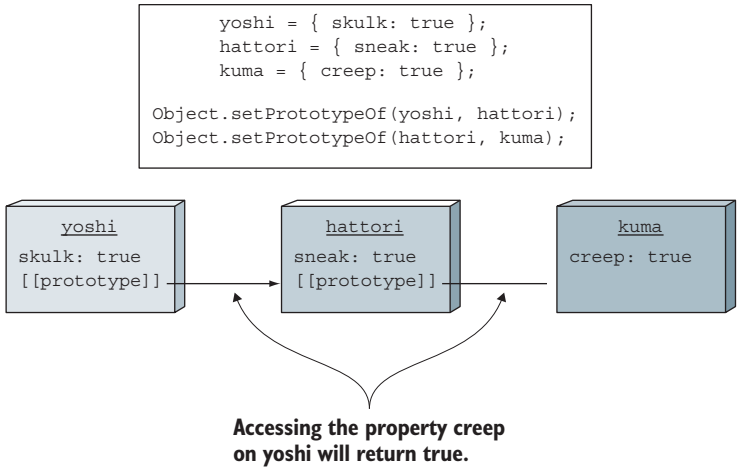

Figure 7.3 The search for a particular property stops when there are no more prototypes to explore. Accessing `yoshi.creep` triggers the search first in `yoshi`, then in `hattori`, and finally in `kuma`.

Now that we have a basic idea of how the search for a particular property occurs through the prototype chain, let's see how prototypes are used when constructing new objects with constructor functions.

7.2 Object construction and prototypes

The simplest way to create a new object is with a statement like this:

```
const warrior = {};
```

This creates a new and empty object, which we can then populate with properties via assignment statements:

```
const warrior = {};
warrior.name = 'Saito';
warrior.occupation = 'marksman';
```

But those coming from an object-oriented background might miss the encapsulation and structuring that comes with a class constructor, a function that serves to initialize an object to a known initial state. After all, if we're going to create multiple instances of the same type of object, assigning the properties individually isn't only tedious but also highly error-prone. We'd like to be able to consolidate the set of properties and methods for a class of objects in one place.

JavaScript provides such a mechanism, though in a different form than most other languages. Like object-oriented languages such as Java and C++, JavaScript employs the `new` operator to instantiate new objects via constructors, but there's no true class definition in JavaScript. Instead, the `new` operator, applied to a constructor function (as you saw in chapter 3), triggers the creation of a newly allocated object.

What we didn't learn in the previous chapters was that every function has a prototype object that's automatically set as the prototype of the objects created with that function. Let's see how that works in the following listing.

Listing 7.2 Creating a `new` instance with a prototyped method

```
function Ninja(){}
Ninja.prototype.swingSword = function(){
  return true;
};

const ninja1 = Ninja();
assert(ninja1 === undefined,
       "No instance of Ninja created.");

const ninja2 = new Ninja();
assert(ninja2 &&
       ninja2.swingSword &&
       ninja2.swingSword(),
       "Instance exists and method is callable." );
```

Defines a function that does nothing and returns nothing

Every function has a built-in prototype object, which we can freely modify.

Calls the function as a function. Testing confirms that nothing at all seems to happen.

Calls the function as a constructor. Testing confirms that not only is a new object instance created, but it possesses the method from the prototype of the function.

In this code, we define a seemingly do-nothing function named `Ninja` that we'll invoke in two ways: as a "normal" function, `const ninja1 = Ninja();` and as a constructor, `const ninja2 = new Ninja();`.

When the function is created, it immediately gets a new object assigned to its prototype object, an object that we can extend just like any other object. In this case, we add a `swingSword` method to it:

```
Ninja.prototype.swingSword = function(){
  return true;
};
```

Then we put the function through its paces. First we call the function normally and store its result in variable `ninja1`. Looking at the function body, we see that it returns no value, so we'd expect `ninja1` to test as `undefined`, which we assert to be `true`. As a simple function, `Ninja` doesn't appear to be all that useful.

Then we call the function via the `new` operator, invoking it as a *constructor*, and something completely different happens. The function is once again called, but this time a newly allocated object has been created and set as the context of the function (and is accessible through the `this` keyword). The result returned from the `new` operator is a reference to this new object. We then test that `ninja2` has a reference to the newly created object, and that that object has a `swingSword` method that we can call. See figure 7.4 for a glimpse of the current application state.

```
        Ninja(){}
Ninja.prototype.swingSword = function(){
        true;
};
…
      ninja2 =     Ninja();
```

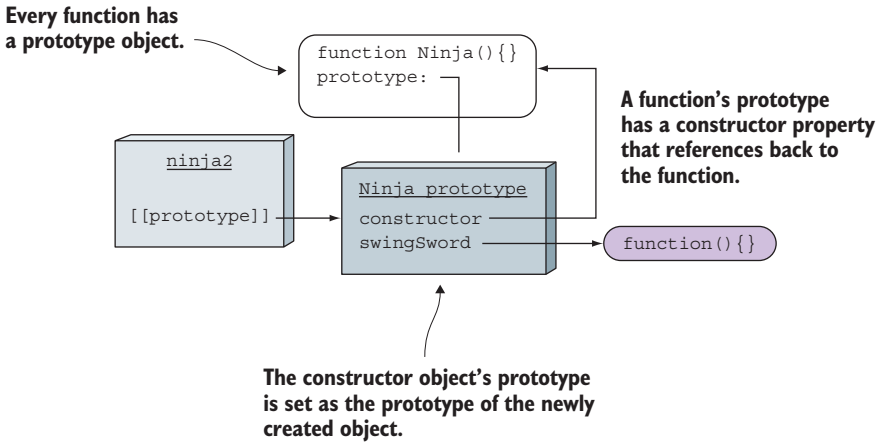

Figure 7.4 Every function, when created, gets a new prototype object. When we use a function as a constructor, the constructed object's prototype is set to the function's prototype.

As you can see, a function, when created, gets a new object that's assigned to its prototype property. The prototype object initially has only one property, `constructor`, that references back to the function (we'll revisit the `constructor` property later).

When we use a function as a constructor (for example, by calling `new Ninja()`), the prototype of the newly constructed object is set to the object referenced by the constructor function's prototype.

In this example, we've extended the `Ninja.prototype` with the `swingSword` method, and when the `ninja2` object is created, its `prototype` property is set to `Ninja`'s prototype. Therefore, when we try to access the `swingSword` property on `ninja2`, the search for that property is delegated to the `Ninja` prototype object. Notice that *all* objects created with the `Ninja` constructor will have access to the `swingSword` method. Now that's code reuse!

The `swingSword` method is a property of the `Ninja`'s prototype, and not a property of `ninja` instances. Let's explore this difference between instance properties and prototype properties.

7.2.1 Instance properties

When the function is called as a constructor via the `new` operator, its context is defined as the new object instance. In addition to exposing properties via the prototype, we can

initialize values within the constructor function via the `this` parameter. Let's examine the creation of such instance properties in the next listing.

Listing 7.3 Observing the precedence of initialization activities

```
function Ninja(){
 this.swung = false;
 this.swingSword = function(){
   return !this.swung;
 };
}
Ninja.prototype.swingSword = function(){
  return this.swung;
};

const ninja = new Ninja();
assert(ninja.swingSword(),
      "Called the instance method, not the prototype method.");
```

Creates an instance variable that holds a Boolean value initialized to false

Creates an instance method that returns the inverse of the swung instance variable value

Defines a prototype method with the same name as the instance method. Which will take precedence?

Constructs a Ninja instance and asserts that the instance method will override the prototype method of the same name

Listing 7.3 is similar to the previous example in that we define a `swingSword` method by adding it to the `prototype` property of the constructor:

```
Ninja.prototype.swingSword = function(){
   return this.swung;
};
```

But we also add an identically named method within the constructor function itself:

```
function Ninja(){
  this.swung = false;
  this.swingSword = function(){
    return !this.swung;
  };
}
```

The two methods are defined to return opposing results so we can tell which will be called.

> **NOTE** This isn't anything we'd advise doing in real-world code; quite the opposite. We're doing it here just to demonstrate the precedence of properties.

When you run the test, you see that it passes! This shows that instance members will hide properties of the same name defined in the prototype. See figure 7.5.

```
function Ninja(){
  this.swung = false;
  this.swingSword = function(){
    return !this.swung;
  };
 }
 Ninja.prototype.swingSword = function(){
   return this.swung;
 };
 const ninja = new Ninja();
```

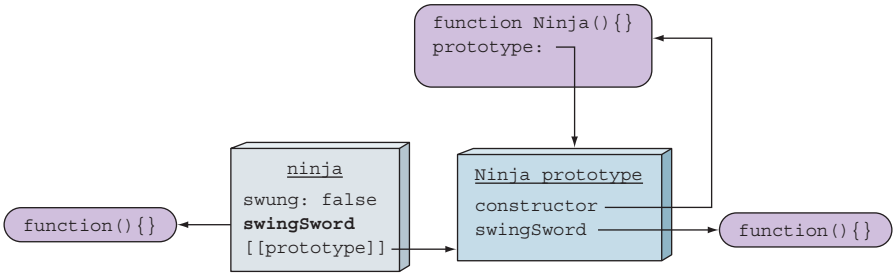

Figure 7.5 **If a property can be found on the instance itself, the prototype isn't even consulted!**

Within the constructor function, the `this` keyword refers to the newly created object, so the properties added within the constructor are created directly on the new `ninja` instance. Later, when we access the property `swingSword` on `ninja`, there's no need to traverse the prototype chain (as shown in figure 7.4); the property created within the constructor is immediately found and returned (see figure 7.5).

This has an interesting side effect. Take a look at figure 7.6, which shows the state of the application if we create three `ninja` instances.

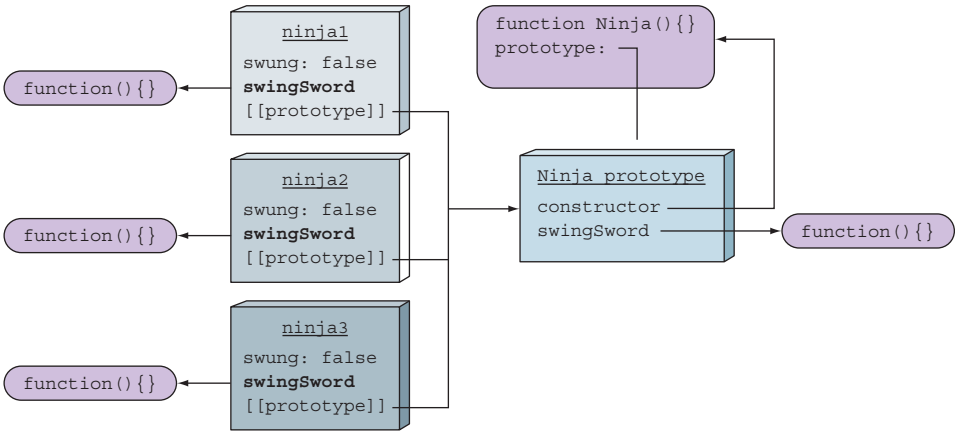

Figure 7.6 **Every instance gets its own version of the properties created within the constructor, but they all have access to the same prototype's properties.**

As you can see, every `ninja` instance gets its own version of the properties that were created within the constructor, while they all have access to the same prototype's properties. This is okay for value properties (for example, `swung`) that are specific to each object instance. But in certain cases it might be problematic for methods.

In this example, we'd have three versions of the `swingSword` method that all perform the same logic. This isn't a problem if we create a couple of objects, but it's something to pay attention to if we plan to create large numbers of objects. Because each method copy behaves the same, creating multiple copies often doesn't make sense, because it only consumes more memory. Sure, in general, the JavaScript engine might perform some optimizations, but that's not something to rely on. From that perspective, it makes sense to place object methods only on the function's prototype, because in that way we have a single method shared by all object instances.

NOTE Remember chapter 5 on closures: Methods defined within constructor functions allow us to mimic private object variables. If this is something we need, specifying methods within constructors is the only way to go.

7.2.2 Side effects of the dynamic nature of JavaScript

You've already seen that JavaScript is a dynamic language in which properties can be easily added, removed, and modified at will. The same thing holds for prototypes, both function prototypes and object prototypes. See the following listing.

Listing 7.4 With prototypes, everything can be changed at runtime

```
function Ninja(){
  this.swung = true;
}

const ninja1 = new Ninja();

Ninja.prototype.swingSword = function(){
  return this.swung;
};
assert(ninja1.swingSword(),
       "Method exists, even out of order.");

Ninja.prototype = {
 pierce: function() {
   return true;
 }
}

assert(ninja1.swingSword(),
       "Our ninja can still swing!");

const ninja2 = new Ninja();
assert(ninja2.pierce(),"Newly created ninjas can pierce");
assert(!ninja2.swingSword, "But they cannot swing!");
```

Defines a constructor that creates a Ninja with a single Boolean property

Creates an instance of Ninja by calling the constructor function via the "new" operator

Adds a method to the prototype after the object has been created

Shows that the method exists in the object

Completely overrides the Ninja's prototype with a new object via the pierce method

Even though we've completely replaced the Ninja constructor's prototype, our Ninja can still swing a sword, because it keeps a reference to the old Ninja prototype.

Newly created ninjas reference the new prototype, so they can pierce but can't swing.

```
function Ninja(){
   this.swung = true;
 }

 const ninja1 = new Ninja();
```

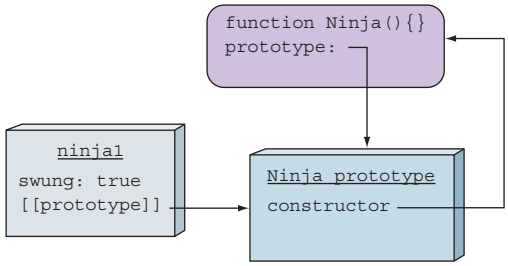

Figure 7.7 **After construction, `ninja1` has the property `swung`, and its prototype is the `Ninja` prototype that has only a `constructor` property.**

Here we again define a `Ninja` constructor and proceed to use it to create an object instance. The state of the application at this moment is shown in figure 7.7.

After the instance has been created, we add a `swingSword` method to the prototype. Then we run a test to show that the change we made to the prototype after the object was constructed takes effect. The current state of the application is shown in figure 7.8.

```
function Ninja(){
  this.swung = true;
}

const ninja1 = new Ninja();

Ninja.prototype.swingSword = function(){
  return this.swung;
};
```

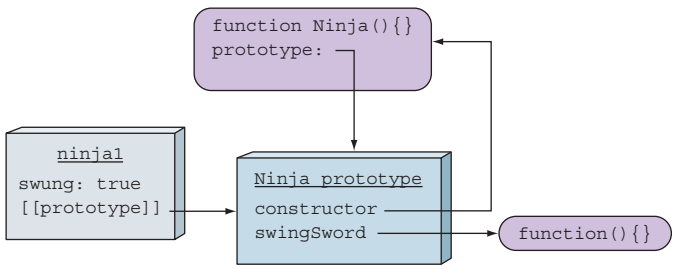

Figure 7.8 **Because the `ninja1` instance references the `Ninja` prototype, even changes made after the instance was constructed are accessible.**

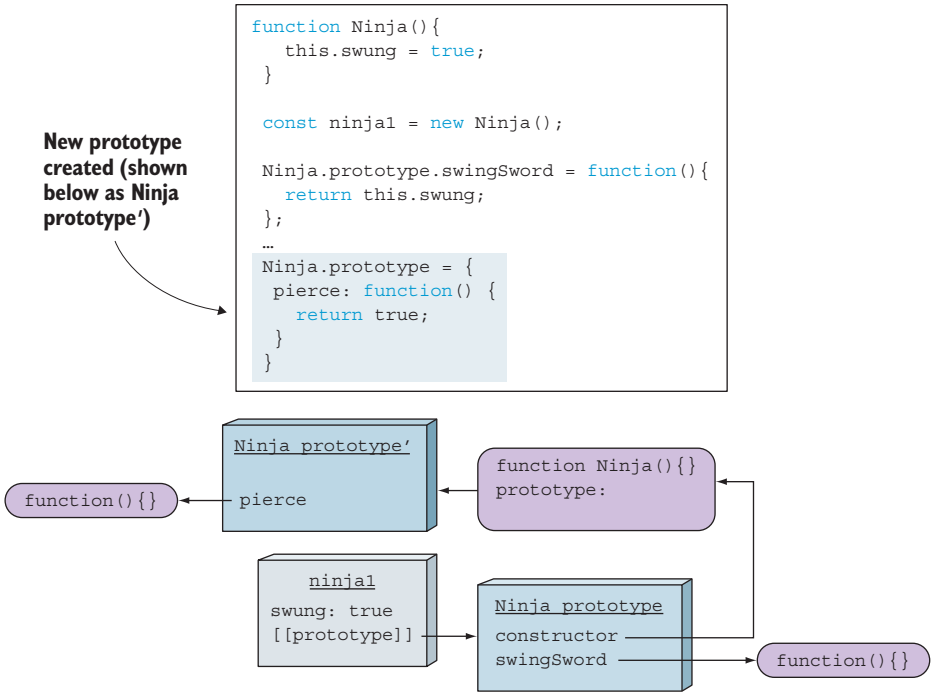

Figure 7.9 The function's prototype can be replaced at will. The already constructed instances reference the old prototype!

Later, we override the `Ninja` function's prototype by assigning it to a completely new object that has a `pierce` method. This results in the application state shown in figure 7.9.

As you can see, even though the `Ninja` function doesn't reference the old `Ninja` prototype, the old prototype is still kept alive by the `ninja1` instance, which can still, through the prototype chain, access the `swingSword` method. But if we create new objects after this prototype switcheroo, the state of the application will be as shown in figure 7.10.

The reference between an object and the function's prototype is established at the time of object instantiation. Newly created objects will have a reference to the new prototype and will have access to the `pierce` method, whereas the old, pre-prototype-change objects keep their original prototype, happily swinging their swords.

We've explored how prototypes work and how they're related to object instantiation. Well done! Now take a quick breath, so we can continue onward by learning more about the nature of those objects.

```
function Ninja(){
   this.swung = true;
 }

 const ninja1 = new Ninja();

 Ninja.prototype.swingSword = function(){
   return this.swung;
 };
 …
 Ninja.prototype = {
  pierce: function() {
    return true;
  }
 }
 …
 const ninja2 = new Ninja();
```

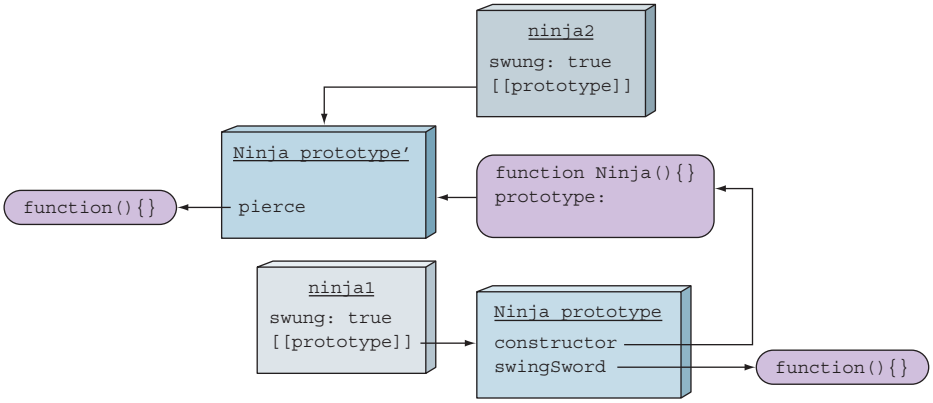

Figure 7.10 **All newly created instances reference the new prototype.**

7.2.3 Object typing via constructors

Although it's great to know how JavaScript uses the prototype to find the correct property references, it's also handy to know which function constructed an object instance. As you've seen earlier, the constructor of an object is available via the constructor property of the constructor function prototype. For example, figure 7.11 shows the state of the application when we instantiate an object with the `Ninja` constructor.

```
function Ninja(){}
const ninja1 = new Ninja();
```

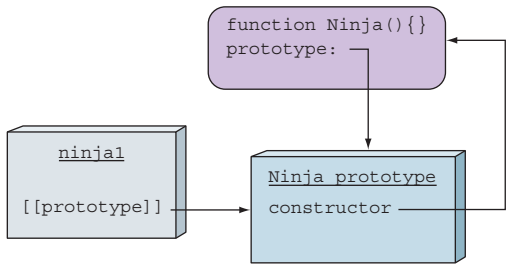

Figure 7.11 **The prototype object of each function has a constructor property that references the function.**

By using the `constructor` property, we can access the function that was used to create the object. This information can be used as a form of type checking, as shown in the next listing.

Listing 7.5 Examining the type of an instance and its constructor

```
function Ninja(){}
const ninja = new Ninja();

assert(typeof ninja === "object",
      "The type of the instance is object.");
assert(ninja instanceof Ninja,
       "instanceof identifies the constructor." );
assert(ninja.constructor === Ninja,
       "The ninja object was created by the Ninja function.");
```

Tests the type of ninja via typeof. This tells us it's an object, but not much else.

Tests the type of ninja via instanceof. This provides more information—that it was constructed from Ninja.

Tests the type of ninja via the constructor reference. This gives a reference to the constructor function.

We define a constructor and create an object instance using it. Then we examine the type of the instance by using the `typeof` operator. This doesn't reveal much, as all instances will be objects, thus always returning `object` as the result. Much more interesting is the `instanceof` operator, which gives us a way to determine whether an instance was created by a particular function constructor. You'll learn more about how the `instanceof` operator works later in the chapter.

In addition, we can use the `constructor` property, that we now know is accessible to all instances, as a reference to the original function that created it. We can use this to verify the origin of the instance (much as we can with the `instanceof` operator).

Additionally, because this is just a reference to the original constructor, we can instantiate a new `Ninja` object using it, as shown in the next listing.

Listing 7.6 Instantiating a new object using a reference to a constructor

```
function Ninja(){}

const ninja = new Ninja();
const ninja2 = new ninja.constructor();

assert(ninja2 instanceof Ninja, "It's a Ninja!");
assert(ninja !== ninja2, "But not the same Ninja!");
```

Constructs a second Ninja from the first

Proves the new object's Ninja-ness

They aren't the same object, but two distinct instances.

Here we define a constructor and create an instance using that constructor. Then we use the `constructor` property of the created instance to construct a second instance. Testing shows that a second `Ninja` has been constructed and that the variable doesn't merely point to the same instance.

What's especially interesting is that we can do this without even having access to the original function; we can use the reference completely behind the scenes, even if the original constructor is no longer in scope.

> **NOTE** Although the `constructor` property of an object can be changed, doing so doesn't have any immediate or obvious constructive purpose (though we might be able to think of some malicious ones). The property's reason for being is to indicate from where the object was constructed. If the `constructor` property is overwritten, the original value is lost.

That's all useful, but we've just scratched the surface of the superpowers that prototypes confer on us. Now things get interesting.

7.3 Achieving inheritance

Inheritance is a form of reuse in which new objects have access to properties of existing objects. This helps us avoid the need to repeat code and data across our code base. In JavaScript, inheritance works slightly differently than in other popular object-oriented languages. Consider the following listing, in which we attempt to achieve inheritance.

Listing 7.7 Trying to achieve inheritance with prototypes

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = { dance: Person.prototype.dance };

const ninja = new Ninja();
assert(ninja instanceof Ninja,
       "ninja receives functionality from the Ninja prototype" );
assert(ninja instanceof Person, "... and the Person prototype" );
assert(ninja instanceof Object, "... and the Object prototype" );
```

Defines a dancing Person via a constructor and its prototype

Defines a Ninja

Attempts to make Ninja a dancing Person by copying the dance method from the Person prototype

Because the prototype of a function is an object, there are multiple ways of copying functionality (such as properties or methods) to effect inheritance. In this code, we define a `Person` and then a `Ninja`. And because a `Ninja` is clearly a person, we want `Ninja` to inherit the attributes of `Person`. We attempt to do so by copying the `dance` property of the `Person` prototype's method to a similarly named property in the `Ninja` prototype.

Running our test reveals that although we may have taught the ninja to dance, we failed to make the `Ninja` a `Person`, as shown in figure 7.12. We taught the `Ninja` to mimic the dance

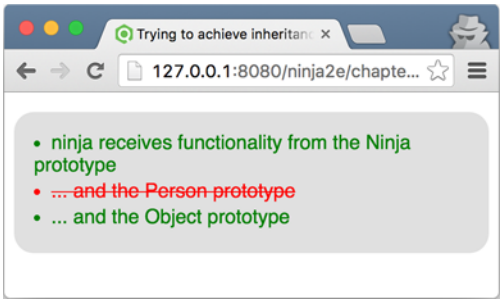

Figure 7.12 Our `Ninja` isn't really a `Person`. No happy dance!

of a person, but that hasn't *made* the `Ninja` a `Person`. That's not inheritance—it's just copying.

Apart from the fact that this approach isn't exactly working, we'd also need to copy each property of `Person` to the `Ninja` prototype individually. That's no way to do inheritance. Let's keep exploring.

What we really want to achieve is a *prototype chain* so that a `Ninja` can *be* a `Person`, and a `Person` can be a `Mammal`, and a `Mammal` can be an `Animal`, and so on, all the way to `Object`. The best technique for creating such a prototype chain is to use an instance of an object as the other object's prototype:

```
SubClass.prototype = new SuperClass();
```

For example:

```
Ninja.prototype = new Person();
```

This preserves the prototype chain, because the prototype of the `SubClass` instance will be an instance of the `SuperClass`, which has a prototype with all the properties of `SuperClass`, and which will in turn have a prototype pointing to an instance of *its* superclass, and on and on. In the next listing, we change listing 7.7 slightly to use this technique.

Listing 7.8 Achieving inheritance with prototypes

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();

const ninja = new Ninja();
assert(ninja instanceof Ninja,
      "ninja receives functionality from the Ninja prototype");
assert(ninja instanceof Person, "... and the Person prototype");
assert(ninja instanceof Object, "... and the Object prototype");
assert(typeof ninja.dance === "function", "... and can dance!")
```

Makes a Ninja a Person by making the Ninja prototype an instance of Person

The only change to the code is to use an instance of `Person` as the prototype for `Ninja`. Running the tests shows that we've succeeded, as shown in figure 7.13. Now we'll take a closer look at the inner workings by looking at the state of the application after we've created the new `ninja` object, as shown in figure 7.14.

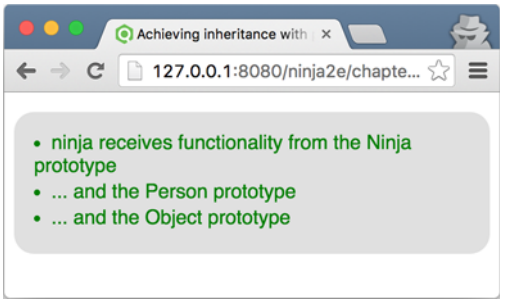

Figure 7.13 Our `Ninja` is a `Person`! Let the victory dance begin.

Figure 7.14 shows that when we define a `Person` function, a `Person`

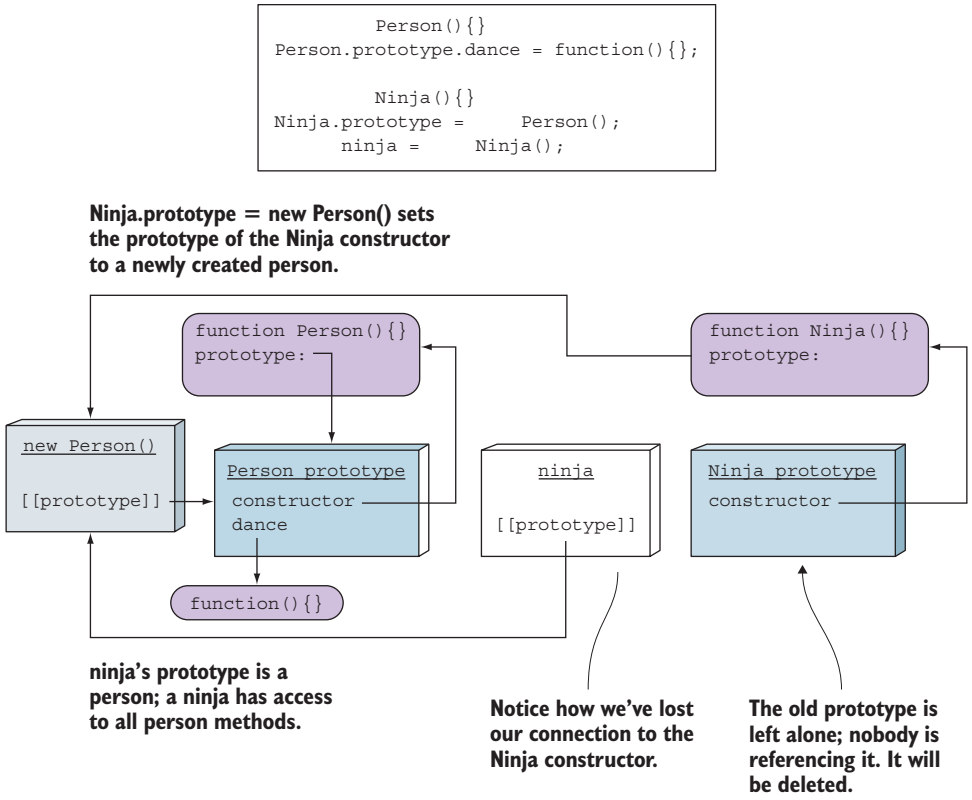

Figure 7.14 **We've achieved inheritance by setting the prototype of the `Ninja` constructor to a new instance of a `Person` object.**

prototype is also created that references the `Person` function through its `constructor` property. Normally, we can extend the `Person` prototype with additional properties, and in this case, we specify that every person, created with the `Person` constructor, has access to the `dance` method:

```
function Person(){}
Person.prototype.dance = function(){};
```

We also define a `Ninja` function that gets its own prototype object with a `constructor` property referencing the `Ninja` function: `function Ninja(){}`.

Next, in order to achieve inheritance, we replace the prototype of the `Ninja` function with a new `Person` instance. Now, when we create a new `Ninja` object, the internal prototype property of the newly created `ninja` object will be set to the object to which the current `Ninja` prototype property points to, the previously constructed `Person` instance:

```
function Ninja(){}
Ninja.prototype = new Person();
var ninja = new Ninja();
```

When we try to access the `dance` method through the `ninja` object, the JavaScript runtime will first check the `ninja` object itself. Because it doesn't have the `dance` property, its prototype, the `person` object, is searched. The `person` object also doesn't have the `dance` property, so its prototype is searched, and the property is finally found. This is how to achieve inheritance in JavaScript!

Here's the important implication: When we perform an `instanceof` operation, we can determine whether the function inherits the functionality of any object in its prototype chain.

> **NOTE** Another technique that may have occurred to you, and that we advise strongly against, is to use the `Person` prototype object directly as the `Ninja` prototype, like this: `Ninja.prototype = Person.prototype`. Any changes to the `Ninja` prototype will then also change the `Person` prototype (because they're the same object), and that's bound to have undesirable side effects.

An additional happy side effect of doing prototype inheritance in this manner is that all inherited function prototypes will continue to live-update. Objects that inherit from the prototype always have access to the current prototype properties.

7.3.1 The problem of overriding the constructor property

If we take a closer look at figure 7.14, we'll see that by setting the new `Person` object as a prototype of the `Ninja` constructor, we've lost our connection to the `Ninja` constructor that was previously kept by the original `Ninja` prototype. This is a problem, because the `constructor` property can be used to determine the function with which the object was created. Somebody using our code could make a perfectly reasonable assumption that the following test will pass:

```
assert(ninja.constructor === Ninja,
      "The ninja object was created by the Ninja constructor");
```

But in the current state of the application, this test fails. As figure 7.14 shows, if we search the `ninja` object for the `constructor` property, we won't find it. So we go over to its prototype, which also doesn't have a constructor property, and again, we follow the prototype and end up in the prototype object of `Person`, which has a `constructor` property referencing the `Person` function. In effect, we get the wrong answer: If we ask the `ninja` object which function has constructed it, we'll get `Person` as the answer. This can be the source of some serious bugs.

It's up to us to fix this situation! But before we can do that, we have to take a detour and see how JavaScript enables us to configure properties.

CONFIGURING OBJECT PROPERTIES

In JavaScript, every object property is described with a *property descriptor* through which we can configure the following keys:

- `configurable`—If set to `true`, the property's descriptor can be changed and the property can be deleted. If set to `false`, we can do neither of these things.
- `enumerable`—If set to `true`, the property shows up during a `for-in` loop over the object's properties (we'll get to the `for-in` loop soon).
- `value`—Specifies the value of the property. Defaults to `undefined`.
- `writable`—If set to `true`, the property value can be changed by using an assignment.
- `get`—Defines the *getter* function, which will be called when we access the property. Can't be defined in conjunction with `value` and `writable`.
- `set`—Defines the *setter* function, which will be called whenever an assignment is made to the property. Also can't be defined in conjunction with `value` and `writable`.

Say we create a property through a simple assignment, for example:

```
ninja.name = "Yoshi";
```

This property will be configurable, enumerable, and writable, its value will be set to `Yoshi`, and functions `get` and `set` would be `undefined`.

When we want to fine-tune our property configuration, we can use the built-in `Object.defineProperty` method, which takes an object on which the property will be defined, the name of the property, and a property descriptor object. As an example, take a look at the following code.

Listing 7.9 Configuring properties

```
var ninja = {};
ninja.name = "Yoshi";
ninja.weapon = "kusarigama";
```

Creates an empty object; uses assignments to add two properties

```
Object.defineProperty(ninja, "sneaky", {
  configurable: false,
  enumerable: false,
  value: true,
  writable: true
});
```

The built-in Object.defineProperty method is used to fine-tune the property configuration details.

```
assert("sneaky" in ninja, "We can access the new property");

for(let prop in ninja){
  assert(prop !== undefined, "An enumerated property: " + prop);
}
```

Uses the for-in loop to iterate over ninja's enumerable properties

We start with the creation of an empty object, to which we add two properties: `name` and `weapon`, in the good old-fashioned way, by using assignments. Next, we use the built-in `Object.defineProperty` method to define the property `sneaky`, which isn't `configurable`, isn't `enumerable`, and has its `value` set to `true`. This `value` can be changed because it's `writable`.

Finally, we test that we can access the newly created `sneaky` property, and we use the `for-in` loop to go through all enumerable properties of the object. Figure 7.15 shows the result.

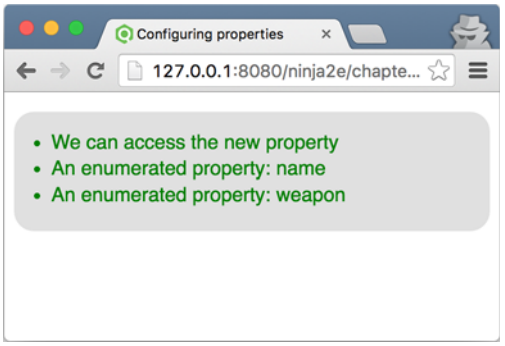

Figure 7.15 Properties `name` and `weapon` will be visited in the `for-in` loop, whereas our specially added `sneaky` property won't (even though we can access it normally).

By setting `enumerable` to `false`, we can be sure that the property won't appear when using the `for-in` loop. To understand why we'd want to do something like this, let's go back to the original problem.

FINALLY SOLVING THE PROBLEM OF OVERRIDING THE CONSTRUCTOR PROPERTY

When trying to extend `Person` with `Ninja` (or to make `Ninja` a subclass of `Person`), we ran into the following problem: When we set a new `Person` object as a prototype to the `Ninja` constructor, we lose the original `Ninja` prototype that keeps our `constructor` property. We don't want to lose the `constructor` property, because it's useful for determining the function used to create our object instances and it might be expected by other developers working on our code base.

We can solve this problem by using the knowledge that we've just obtained. We'll define a new `constructor` property on the new `Ninja.prototype` by using the `Object.defineProperty` method. See the following listing.

Listing 7.10 Fixing the constructor property problem

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();

Object.defineProperty(Ninja.prototype, "constructor", {
  enumerable: false,
  value: Ninja,
  writable: true
});

var ninja = new Ninja();
```

We define a new non-enumerable constructor property pointing back to Ninja.

We've reestablished the connection.

```
assert(ninja.constructor === Ninja,
       "Connection from ninja instances to Ninja constructor
         reestablished!");
for(let prop in Ninja.prototype){
  assert(prop === "dance", "The only enumerable property is dance!");
}
```

We haven't added any enumerable properties to the Ninja.prototype.

Now if we run the code, we'll see that everything is peachy. We've reestablished the connection between `ninja` instances and the `Ninja` function, so we can know that they were constructed by the `Ninja` function. In addition, if anybody tries to loop through the properties of the `Ninja.prototype` object, we've made sure that our patched-on property `constructor` won't be visited. Now that's the mark of a true ninja; we went in, did our job, and got out, without anybody noticing anything from the outside!

7.3.2 The instanceof operator

In most programming languages, the straightforward approach for checking whether an object is a part of a class hierarchy is to use the `instanceof` operator. For example, in Java, the `instanceof` operator works by checking whether the object on the left side is either the same class or a subclass of the class type on the right.

Although certain parallels could be made with how the `instanceof` operator works in JavaScript, there's a little twist. In JavaScript, the `instanceof` operator works on the prototype chain of the object. For example, say we have the following expression:

```
ninja instanceof Ninja
```

The `instanceof` operator works by checking whether the *current* prototype of the `Ninja` function is in the prototype chain of the `ninja` instance. Let's go back to our persons and ninjas, for a more concrete example.

Listing 7.11 Studying the `instanceof` operator

```
function Person(){}
function Ninja(){}

Ninja.prototype = new Person();

const ninja = new Ninja();

assert(ninja instanceof Ninja, "Our ninja is a Ninja!");
assert(ninja instanceof Person, "A ninja is also a Person. ");
```

A ninja instance is both a Ninja and a Person.

As expected, a ninja is, at the same time, a `Ninja` and a `Person`. But, to nail down this point, figure 7.16 shows how the whole thing works behind the scenes.

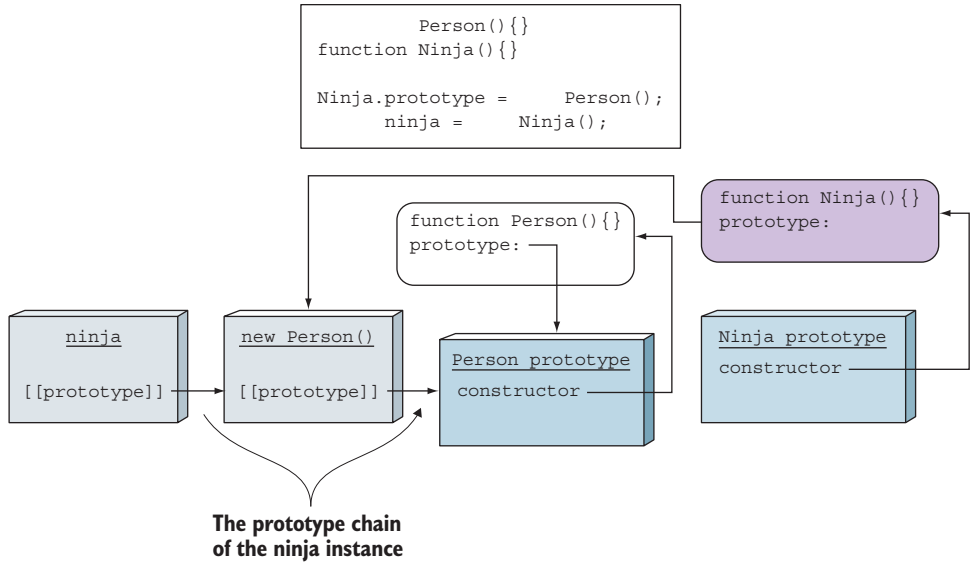

Figure 7.16 The prototype chain of a `ninja` instance is composed of a `new Person()` object and the `Person prototype`.

The prototype chain of a ninja instance is composed of a `new Person()` object, through which we've achieved inheritance, and the `Person prototype`. When evaluating the expression `ninja instanceof Ninja`, the JavaScript engine takes the prototype of the `Ninja` function, the `new Person()` object, and checks whether it's in the prototype chain of the `ninja` instance. Because the `new Person()` object is a direct prototype of the `ninja` instance, the result is `true`.

In the second case, where we check `ninja instanceof Person`, the JavaScript engine takes the prototype of the `Person` function, the `Person prototype`, and checks whether it can be found in the prototype chain of the `ninja` instance. Again, it can, because it's the prototype of our `new Person()` object, which, as we've already seen, is the prototype of the `ninja` instance.

And that's all there is to know about the `instanceof` operator. Although its most common use is in providing a clear way to determine whether an instance was created by a particular function constructor, it doesn't exactly work like that. Instead, it checks whether the prototype of the right-side function is in the prototype chain of the object on the left. Therefore, there is a caveat that we should be careful about.

THE INSTANCEOF CAVEAT

As you've seen multiple times throughout this chapter, JavaScript is a dynamic language in which we can modify a *lot* of things during program execution. For example, there's nothing stopping us from changing the prototype of a constructor, as shown in the following listing.

Listing 7.12 Watch out for changes to constructor prototypes

```
function Ninja(){}

const ninja = new Ninja();

assert(ninja instanceof Ninja, "Our ninja is a Ninja!");

Ninja.prototype = {};

assert(!(ninja instanceof Ninja), "The ninja is now not a Ninja!?");
```

We change the prototype of the Ninja constructor function.

Even though our ninja instance was created by the Ninja constructor, the instanceof operator now says that ninja isn't an instance of Ninja anymore!

In this example, we again repeat all the basic steps of making a `ninja` instance, and our first test goes fine. But if we change the prototype of the `Ninja` constructor function *after* the creation of the `ninja` instance, and again test whether `ninja` is an `instanceof Ninja`, we'll see that the situation has changed. This will surprise us only if we cling to the inaccurate assumption that the `instanceof` operator tells us whether an instance was created by a particular function constructor. If, on the other hand, we take the real semantics of the `instanceof` operator—that it checks only whether the prototype of the function on the right side is in the prototype chain of the object on the left side—we won't be surprised. This situation is shown in figure 7.17.

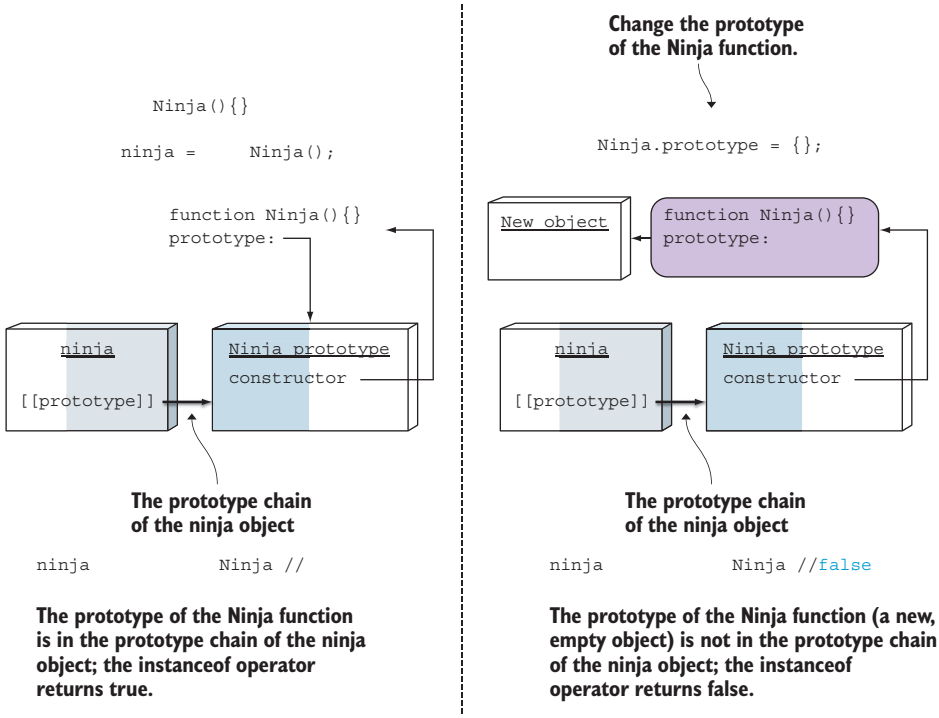

Figure 7.17 The `instanceof` operator checks whether the prototype of the function on the right is in the prototype chain of the object on the left. Be careful; the function's prototype can be changed anytime!

Now that we understand how prototypes work in JavaScript, and how to use prototypes in conjunction with constructor functions to implement inheritance, let's move on to a new addition in the ES6 version of JavaScript: classes.

7.4 Using JavaScript "classes" in ES6

It's great that JavaScript lets us use a form of inheritance via prototypes. But many developers, especially those from a classical object-oriented background, would prefer a simplification or abstraction of JavaScript's inheritance system into one that they're more familiar with.

This inevitably leads toward the realm of classes, even though JavaScript doesn't support classical inheritance natively. As a response to this need, several JavaScript libraries that simulate classical inheritance have popped up. Because each library implements classes in its own way, the ECMAScript committee has standardized the syntax for simulating class-based inheritance. Notice how we said *simulating*. Even though now we can use the `class` keyword in JavaScript, the underlying implementation is still based on prototype inheritance!

NOTE The `class` keyword has been added to the ES6 version of JavaScript, and not all browsers implement it (see http://mng.bz/3ykA for current support).

Let's start by studying the new syntax.

7.4.1 Using the class keyword

ES6 introduces a new `class` keyword that provides a much more elegant way of creating objects and implementing inheritance than manually implementing it ourselves with prototypes. Using the `class` keyword is easy, as shown in the following listing.

Listing 7.13 Creating a class in ES6

Uses the class keyword to start specifying an ES6 class

Defines a constructor function that will be called when we call the class with the keyword new

Defines an additional method accessible to all Ninja instances

Instantiates a new ninja object with the keyword new

Tests for the expected behavior

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

  swingSword(){
    return true;
  }
}

var ninja = new Ninja("Yoshi");

assert(ninja instanceof Ninja, "Our ninja is a Ninja");
assert(ninja.name === "Yoshi", "named Yoshi");
assert(ninja.swingSword(), "and he can swing a sword");
```

Listing 7.13 shows that we can create a `Ninja` class by using the `class` keyword. When creating ES6 classes, we can explicitly define a `constructor` function that will be invoked when instantiating a `Ninja` instance. In the constructor's body, we can access the newly created instance with the `this` keyword, and we can easily add new properties, such as the `name` property. Within the class body, we can also define methods that will be accessible to all `Ninja` instances. In this case, we've defined a `swingSword` method that returns `true`:

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

  swingSword(){
    return true;
  }
}
```

Next we can create a `Ninja` instance by calling the `Ninja` class with the keyword `new`, just as we would if `Ninja` was a simple constructor function (as earlier in the chapter):

```
var ninja = new Ninja("Yoshi");
```

Finally, we can test that the `ninja` instance behaves as expected, that it's an `instanceof` `Ninja`, has a `name` property, and has access to the `swingSword` method:

```
assert(ninja instanceof Ninja, "Our ninja is a Ninja");
assert(ninja.name === "Yoshi", "named Yoshi");
assert(ninja.swingSword(), "and he can swing a sword");
```

CLASSES ARE SYNTACTIC SUGAR

As mentioned earlier, even though ES6 has introduced the `class` keyword, under the hood we're still dealing with good old prototypes; classes are syntactic sugar designed to make our lives a bit easier when mimicking classes in JavaScript.

Our class code from listing 7.13 can be translated to functionally identical ES5 code:

```
function Ninja(name) {
  this.name = name;
}
Ninja.prototype.swingSword = function() {
  return true;
};
```

As you can see, there's nothing especially new with ES6 classes. The code is more elegant, but the same concepts are applied.

STATIC METHODS

In the previous examples, you saw how to define object methods (prototype methods), accessible to all object instances. In addition to such methods, classical object-oriented languages such as Java use static methods, methods defined on a class level. Check out the following example.

Listing 7.14 Static methods in ES6

```
class Ninja{
  constructor(name, level){
    this.name = name;
    this.level = level;
  }

  swingSword() {
    return true;
  }

  static compare(ninja1, ninja2){
    return ninja1.level - ninja2.level;
  }
}

var ninja1 = new Ninja("Yoshi", 4);
var ninja2 = new Ninja("Hattori", 3);

assert(!("compare" in ninja1) && !("compare" in ninja2),
       "A ninja instance doesn't know how to compare");

assert(Ninja.compare(ninja1, ninja2) > 0,
      "The Ninja class can do the comparison!");

assert(!("swingSword" in Ninja),
      "The Ninja class cannot swing a sword");
```

Uses the static keyword to make a static method

ninja instances don't have access to compare.

The class Ninja has access to the compare method.

We again create a `Ninja` class that has a `swingSword` method accessible from all `ninja` instances. We also define a static method, `compare`, by prefixing the method name with the keyword `static`.

```
static compare(ninja1, ninja2){
    return ninja1.level - ninja2.level;
}
```

The `compare` method, which compares the skill levels of two ninjas, is defined on the class level, and not the instance level! Later we test that this effectively means that the `compare` method isn't accessible from `ninja` instances but is accessible from the `Ninja` class:

```
assert(!("compare" in ninja1) && !("compare" in ninja2),
       "The ninja instance doesn't know how to compare");
assert(Ninja.compare(ninja1, ninja2) > 0,
      "The Ninja class can do the comparison!");
```

We can also look at how "static" methods can be implemented in pre-ES6 code. For this, we have to remember only that classes are implemented through functions. Because static methods are class-level methods, we can implement them by taking advantage of functions as first-class objects, and adding a method property to our constructor function, as in the following example:

```
function Ninja(){}
Ninja.compare = function(ninja1, ninja2){...}
```

Extends the constructor function with a method to mimic static methods in pre-ES6 code

Now let's move on to inheritance.

7.4.2 Implementing inheritance

To be honest, performing inheritance in pre-ES6 code can be a pain. Let's go back to our trusted Ninjas, Persons example:

```
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}
Ninja.prototype = new Person();

Object.defineProperty(Ninja.prototype, "constructor", {
  enumerable: false,
  value: Ninja,
  writable: true
});
```

There's a lot to keep in mind here: Methods accessible to all instances should be added directly to the prototype of the constructor function, as we did with the `dance` method and the `Person` constructor. If we want to implement inheritance, we have to set the prototype of the derived "class" to the instance of the base "class." In this case, we assigned a new instance of `Person` to `Ninja.prototype`. Unfortunately, this messes up the `constructor` property, so we have to manually restore it with the `Object.defineProperty` method. This is a lot to keep in mind when trying to achieve a relatively simple and commonly used feature (inheritance). Luckily, with ES6, all of this is significantly simplified.

Let's see how it's done in the following listing.

Listing 7.15 Inheritance in ES6

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

  dance(){
    return true;
  }
```

```
}

class Ninja extends Person {
  constructor(name, weapon){
    super(name);
    this.weapon = weapon;
  }

  wieldWeapon(){
    return true;
  }
}

var person = new Person("Bob");

assert(person instanceof Person, "A person's a person");
assert(person.dance(), "A person can dance.");
assert(person.name === "Bob", "We can call it by name.");
assert(!(person instanceof Ninja), "But it's not a Ninja");
assert(!("wieldWeapon" in person), "And it cannot wield a weapon");


var ninja = new Ninja("Yoshi", "Wakizashi");
assert(ninja instanceof Ninja, "A ninja's a ninja");
assert(ninja.wieldWeapon(), "That can wield a weapon");
assert(ninja instanceof Person, "But it's also a person");
assert(ninja.name === "Yoshi" , "That has a name");
assert(ninja.dance(), "And enjoys dancing");
```

Uses the extends keyword to inherit from another class

Uses the super keyword to call the base class constructor

Listing 7.15 shows how to achieve inheritance in ES6; we use the `extends` keyword to inherit from another class:

```
class Ninja extends Person
```

In this example, we create a `Person` class with a constructor that assigns a `name` to each `Person` instance. We also define a `dance` method that will be accessible to all `Person` instances:

```
class Person {
  constructor(name){
    this.name = name;
  }
  dance(){
    return true;
  }
}
```

Next we define a `Ninja` class that extends the `Person` class. It has an additional `weapon` property, and a `wieldWeapon` method:

```
class Ninja extends Person {
  constructor(name, weapon){
    super(name);
```

```
    this.weapon = weapon;
  }

  wieldWeapon(){
    return true;
  }
}
```

In the constructor of the derived, `Ninja` class, there's a call to the constructor of the base, `Person` class, through the keyword `super`. This should be familiar, if you've worked with any class-based language.

We continue by creating a `person` instance and checking that it's an instance of the `Person` class that has a `name` and can `dance`. Just to be sure, we also check that a person who *isn't* a Ninja can't wield a weapon:

```
var person = new Person("Bob");

assert(person instanceof Person, "A person's a person");
assert(person.dance(), "A person can dance.");
assert(person.name === "Bob", "We can call it by name.");
assert(!(person instanceof Ninja), "But it's not a Ninja");
assert(!("wieldWeapon" in person), "And it cannot wield a weapon");
```

We also create a `ninja` instance and check that it's an instance of `Ninja` and can wield a weapon. Because every `ninja` is also a `Person`, we check that a ninja is an instance of `Person`, that it has a `name`, and that it also, in the interim of fighting, enjoys dancing:

```
var ninja = new Ninja("Yoshi", "Wakizashi");
assert(ninja instanceof Ninja, "A ninja's a ninja");
assert(ninja.wieldWeapon(), "That can wield a weapon");
assert(ninja instanceof Person, "But it's also a person");
assert(ninja.name === "Yoshi" , "That has a name");
assert(ninja.dance(), "And enjoys dancing");
```

See how easy this is? There's no need to think about prototypes or the side effects of certain overridden properties. We define classes and specify their relationship by using the `extends` keyword. Finally, with ES6, hordes of developers coming from languages such as Java or C# can be at peace.

And that's it. With ES6, we build class hierarchies almost as easily as in any other, more conventional object-oriented language.

7.5 Summary

- JavaScript objects are simple collections of named properties with values.
- JavaScript uses prototypes.
- Every object can have a reference to a *prototype*, an object to which we delegate the search for a particular property, if the object itself doesn't have the searched-for property. An object's prototype can have its own prototype, and so on, forming a *prototype chain*.

- We can define the prototype of an object by using the `Object.setPrototypeOf` method.
- Prototypes are closely linked to constructor functions. Every function has a `prototype` property that's set as the prototype of objects that it instantiates.
- A function's `prototype` object has a `constructor` property pointing back to the function itself. This property is accessible to all objects instantiated with that function and, with certain limitations, can be used to find out whether an object was created by a particular function.
- In JavaScript, almost everything can be changed at runtime, including an object's prototypes and a function's prototypes!
- If we want the instances created by a `Ninja` constructor function to "inherit" (more accurately, have access to) properties accessible to instances created by the `Person` constructor function, set the prototype of the `Ninja` constructor to a new instance of the `Person` class.
- In JavaScript, properties have attributes (configurable, enumerable, writable). These properties can be defined by using the built-in `Object.defineProperty` method.
- JavaScript ES6 adds support for a `class` keyword that enables us to more easily mimic classes. Behind the scenes, prototypes are still in play!
- The `extends` keyword enables elegant inheritance.

7.6 Exercises

1 Which of the following properties points to an object that will be searched if the target object doesn't have the searched-for property?

a `class`
b `instance`
c `prototype`
d `pointTo`

2 What's the value of variable `a1` after the following code is executed?

```
function Ninja(){}
Ninja.prototype.talk = function (){
  return "Hello";
};

const ninja = new Ninja();
const a1 = ninja.talk();
```

3 What's the value of `a1` after running the following code?

```
function Ninja(){}
Ninja.message = "Hello";

const ninja = new Ninja();

const a1 = ninja.message;
```

4 Explain the difference between the `getFullName` method in these two code fragments:

```
//First fragment
function Person(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;

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

//Second fragment
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

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

5 After running the following code, what will `ninja.constructor` point to?

```
function Person() { }
function Ninja() { }

const ninja = new Ninja();
```

6 After running the following code, what will `ninja.constructor` point to?

```
function Person() { }
function Ninja() { }
Ninja.prototype = new Person();
const ninja = new Ninja();
```

7 Explain how the `instanceof` operator works in the following example.

```
function Warrior() { }

function Samurai() { }
Samurai.prototype = new Warrior();

var samurai = new Samurai();

samurai instanceof Warrior; //Explain
```

8 Translate the following ES6 code into ES5 code.

```
class Warrior {
  constructor(weapon){
```

```
    this.weapon = weapon;
  }

  wield() {
    return "Wielding " + this.weapon;
  }

  static duel(warrior1, warrior2){
    return warrior1.wield() + " " + warrior2.wield();
  }
}
```

JAVASCRIPT/WEB DEVELOPMENT

Secrets of the JavaScript Ninja Second Edition

Resig • Bibeault • Maras

JavaScript is rapidly becoming a universal language for every type of application, whether on the web, on the desktop, in the cloud, or on mobile devices. When you become a JavaScript pro, you have a powerful skill set that's usable across all these domains.

Secrets of the JavaScript Ninja, Second Edition uses practical examples to clearly illustrate each core concept and technique. This completely revised edition shows you how to master key JavaScript concepts such as functions, closures, objects, prototypes, and promises. It covers APIs such as the DOM, events, and timers. You'll discover best practice techniques such as testing, and cross-browser development, all taught from the perspective of skilled JavaScript practitioners.

What's Inside

- Writing more effective code with functions, objects, and closures
- Learning to avoid JavaScript application pitfalls
- Using regular expressions to write succinct text-processing code
- Managing asynchronous code with promises
- Fully revised to cover concepts from ES6 and ES7

You don't have to be a ninja to read this book—just be willing to become one. Are you ready?

John Resig is an acknowledged JavaScript authority and creator of the jQuery library. **Bear Bibeault** is a web developer and author of the first edition, as well as coauthor of *Ajax in Practice*, *Prototype and Scriptaculous in Action*, and *jQuery in Action*. **Josip Maras** is a post-doctoral researcher and teacher.

"Essential reading for developers of any discipline ... with powerful techniques to improve your JavaScript."
—Becky Huett, Big Shovel Labs

"Excellent and comprehensive insight into the magic of functions and closures for the efficient use of JavaScript."
—Gerd Klevesaat, Siemens

"The essential resource for moving your JavaScript skills to the next level."
—David Starkey, Blum

"Helps you master both the stealthy and bold techniques of modern JavaScript."
—Christopher Haupt
New Relic Inc.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/secrets-of-the-javascript-ninja-second-edition

ISBN-13: 978-1-61729-285-9
ISBN-10: 1-61729-285-0
54499
9 781617 292859

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]