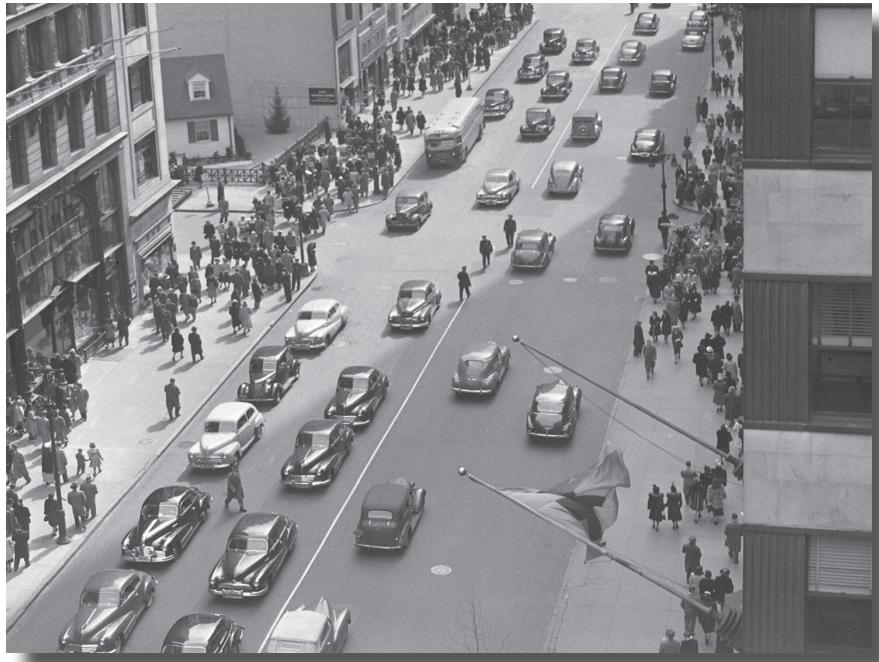


## 12 advanced object construction

# Creating objects



**So far we've been crafting objects by hand.** For each object, we've used an **object literal** to specify each and every property. That's okay on a small scale, but for serious code we need something better. That's where **object constructors** come in. With constructors we can create objects much more easily, and we can create objects that all adhere to the same **design blueprint**—meaning we can use constructors to ensure each object has the same properties and includes the same methods. And with constructors we can write object code that is much more **concise** and a lot less error prone when we're creating lots of objects. So, let's get started and after this chapter you'll be talking constructors just like you grew up in Objectville.

## Creating objects with object literals

So far in this book, you've been using *object literals* to create objects. With an object literal, you create an object by writing it out... well, literally. Like this:

```
var taxi = {  
    make: "Webville Motors",  
    model: "Taxi",  
    year: 1955,  
    color: "yellow",  
    passengers: 4,  
    convertible: false,  
    mileage: 281341,  
    started: false,  
  
    start: function() { this.started = true; },  
    stop: function() { this.started = false; },  
    drive: function {  
        // drive code here  
    }  
};
```



With an object literal you type out each part of the object within curly braces. When you're done, the result is an actual JavaScript object, which you typically assign to a variable for later use.

Object literals give you a convenient way to create objects anywhere in your code, but when you need to create lots of objects—say a whole fleet of taxis—you wouldn't want to type in a hundred different object literals now would you?



Think about creating a fleet of taxi objects. What other issues might using object literals cause?

- Tired fingers from a lot of typing!
- Can you ensure that each taxi has the same properties? What if you make a mistake or typo, or leave out a property?
- A lot of object literals means a lot of code. Isn't that going to lead to slow download times for the browser?
- The code for the start, stop and drive methods would have to be duplicated over and over.
- What if you decide to add or delete a property (or to change the way start or stop work)? You'd have to make the change in all the taxis.
- Who needs taxis when we have Uber?

# Using conventions for objects

The other thing we've been doing, so far, is creating objects *by convention*. For example, we've been putting properties and methods together and saying "it's a car!" or "it's a dog!", but the only thing that makes two such objects cars (or dogs) is that we've followed our own conventions.

Now, this technique might work on a small scale but it's problematic when we have lots of objects, or even lots of developers working in the same code who might not fully know or follow the conventions.

But don't take our word for it. Take a look at some of the objects we've seen earlier in the book, which we've been told are cars:

Okay, this looks a lot like our other car objects...but wait. This has a rocket thruster. Hmm, not sure this is really a car.

The tbird looks like a great car, but we're not seeing some of the basic properties it needs, like mileage or color. It also seems to have a few extra properties. That could be a problem...



```
var rocketCar = {
  make: "Galaxy",
  model: "4000",
  year: 2001,
  color: "white",
  passengers: 6,
  convertible: false,
  mileage: 60191919,
  started: false,
  start: function() {
    this.started = true;
  },
  stop: function() {
    this.started = false;
  },
  drive: function() {
    // drive code here
  },
  thrust: function(amount) {
    // code for thrust
  }
};
```



```
var toyCar = {
  make: "Mattel",
  model: "PeeWee",
  color: "blue",
  type: "wind up",
  price: "2.99"
};
```

Wait, this might be a car but it looks nothing like our other cars. It does have a make, model and color, but this looks like a toy, not a car. What's this doing here?

This definitely looks like the cars we've been dealing with. It has all the same properties and methods.



```
var taxi = {
  make: "Webville Motors",
  model: "Taxi",
  year: 1955,
  color: "yellow",
  passengers: 4,
  convertible: false,
  mileage: 281341,
  started: false,
  start: function() {
    this.started = true;
  },
  stop: function() {
    this.started = false;
  }
};
```



```
var tbird = {
  make: "Ford",
  model: "Thunderbird",
  year: 1957,
  passengers: 4,
  convertible: true,
  started: false,
  oilLevel: 1.0,
  start: function() {
    if (oilLevel > .75) {
      this.started = true;
    }
  },
  stop: function() {
    this.started = false;
  },
  drive: function() {
    // drive code here
  }
};
```



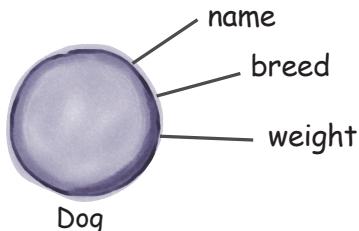
If only I could find a way to **create objects** that all share the same basic structure. That way all my objects would look the same by having all the right properties and all my methods would be defined in one place. It would be something like a cookie cutter that just stamps out copies of the object for me. That would be dreamy. But I know it's just a fantasy...

# Introducing Object Constructors

*Object constructors*, or “constructors” for short, are your path to better object creation. Think of a constructor like a little factory that can create an endless number of similar objects.

In terms of code, a constructor is quite similar to a function that returns an object: you define it once and invoke it every time you want to create a new object. But as you’ll see there’s a little extra that goes into a constructor.

The best way to see how constructors work is to create one. Let’s revisit our old friend, the dog object, from earlier in the book and write a constructor to create as many dogs as we need. Here’s a version of the dog object we’ve used before, with a name, a breed and a weight.



Now, if we were going to define such a dog with an object literal, it would look like this:

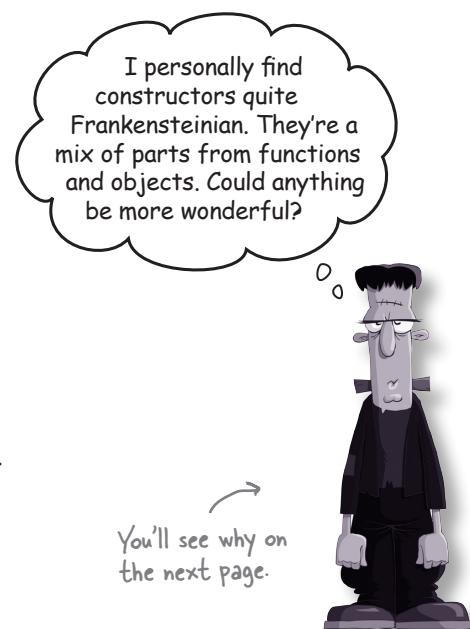
```
var dog = {
  name: "Fido",
  breed: "Mixed",
  weight: 38
};
```

Just a simple dog object created by an object literal. Now we need to figure out how to create a lot of these puppies.

But we don’t want *just a Fido* dog, we want a way to create *any dog* that has a name, a breed and a weight. And, again, to do that we’re going to write some code that looks like a function, with a dash of object syntax thrown in.

With that introduction, you must be a bit curious—go ahead and turn the page and let’s get these constructors figured out and working for us.

**Object constructors and functions are closely related. Keep that in mind as you’re learning how to write and use constructors.**



# How to create a Constructor

Using constructors is a two-step process: first we define a constructor, and then we use it to create objects. Let's first focus on creating a constructor.

What we want is a constructor that we can use to create dogs, and, more specifically, dogs with names, breeds and weights. So, we're going to define a function, called the constructor, that knows how to create dogs. Like this:



A constructor function looks just like a regular function. ↴

But notice that we give the name of the constructor function a capital letter. This isn't required; but everyone does it as a convention.

The parameters of the function match the properties we want to supply for each individual dog.

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
```

← This part feels more like an object because we're assigning each parameter to what looks like a property.

The property names and parameter names don't have to be the same, but they often are—again, by convention.

Hmm, we're not using local variables like in most functions. Instead we're using the `this` keyword, and we've only used that inside objects so far.

↑ Hang on; we'll look at how we use the constructor next and then all this is going to fall into place and make more sense.



## Sharpen your pencil

We need your help. We've been using object literals to create ducks. Given what you learned above, can you write a constructor to create ducks for us? You'll find one of our object literals below to base your constructor on:

```
var duck = {
    type: "redheaded",
    canFly: true
}
```

Here's an example duck object literal.

Write a constructor for creating ducks.

P.S. We know you haven't fully figured out how this all works yet, so for now concentrate on the syntax.

# How to use a Constructor

We said using a constructor is a two-step process: first we create a constructor, then we use it. Well, we've created a Dog constructor, so let's use it. Here's how we do that:

To create a dog, we use the new operator with the constructor.  
 ↓      ↗  
 Followed by a call to the constructor.  
 ↓  
 And the arguments.  
 ↗  
**var fido = new Dog("Fido", "Mixed", 38);**

Try saying it out loud:  
 "to create fido, I create  
 a new dog object with  
 the name Fido that is a  
 mixed breed and weighs  
 38 pounds."

So, to create a new dog object with a name of "Fido", a breed of "Mixed" and a weight of 38, we start with the new keyword and follow it by a call to the constructor function with the appropriate arguments. After this statement is evaluated, the variable `fido` will hold a reference to our new dog object.

Now that we have a constructor for dogs, we can keep making them:

```
var fluffy = new Dog("Fluffy", "Poodle", 30);  
var spot = new Dog("Spot", "Chihuahua", 10);
```

That's a bit easier than using object literals isn't it? And by creating dog objects this way, we know each dog has the same set of properties: name, breed, and weight.



```
function Dog(name, breed, weight) {  
    this.name = name;  
    this.breed = breed;  
    this.weight = weight;  
}  
  
var fido = new Dog("Fido", "Mixed", 38);  
var fluffy = new Dog("Fluffy", "Poodle", 30);  
var spot = new Dog("Spot", "Chihuahua", 10);  
var dogs = [fido, fluffy, spot];  
  
for (var i = 0; i < dogs.length; i++) {  
    var size = "small";  
    if (dogs[i].weight > 10) {  
        size = "large";  
    }  
    console.log("Dog: " + dogs[i].name  
        + " is a " + size  
        + " " + dogs[i].breed);  
}
```

Let's get some quick hands-on experience to help this all sink in. Go ahead and put this code in a page and give it a test drive. Write your output here.



# How constructors work

We've seen how to declare a constructor and also how to use it to create objects, but we should also take a look behind the scenes to see how a constructor actually works. Here's the key: to understand constructors we need to know what the new operator is doing.

We'll start with the statement we used to create fido:

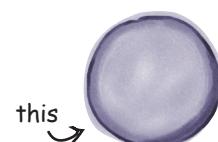
```
var fido = new Dog("Fido", "Mixed", 38);
```

Take a look at the right-hand side of the assignment, where all the action is. Let's follow its execution:

- ① The first thing new does is create a new, empty object:



- ② Next, new sets this to point to the new object.



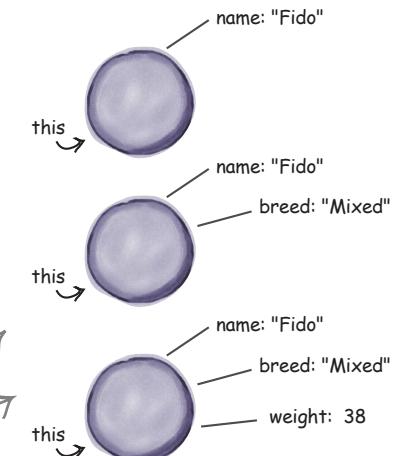
← Remember from Chapter 5 that this holds a reference to the current object our code is dealing with.

- ③ With this set up, we now call the function Dog, passing "Fido", "Mixed" and 38 as arguments.

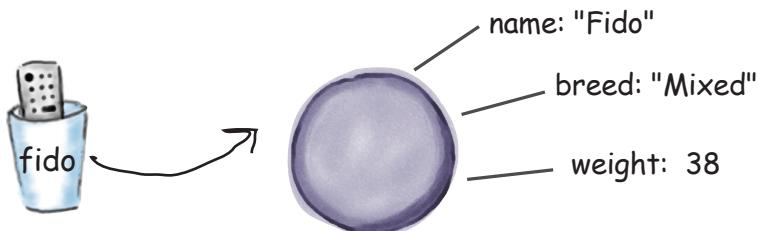
```
"Fido"      "Mixed"      38
      ↓          ↓          ↓
function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
}
```

- ④ Next the body of the function is invoked. Like most constructors, Dog assigns values to properties in the newly created this object.

Executing the body of the Dog function customizes the new object with three properties, assigning them the values of the respective parameters.



- ⑤ Finally, once the Dog function has completed its execution the `new` operator returns `this`, which is a reference to the newly created object. Notice `this` is returned for you; you don't have to explicitly return it in your code. And after the new object has been returned, we assign that reference to the variable `fido`.



## BE the Browser

Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors in the code. After you've done the exercise look at the end of the chapter to see if you found them all. And, hey by the way, this is Chapter 12. Feel free to make style comments too. You've earned the right.

```

function widget(partNo, size) {
    var this.no = partNo;
    var this.breed = size;
}
function FormFactor(material, widget) {
    this.material = material,
    this.widget = widget,
    return this;
}

var widgetA = widget(100, "large");
var widgetB = new widget(101, "small");
var formFactorA = newFormFactor("plastic", widgetA);
var formFactorB = new ForumFactor("metal", widgetB);

```

## You can put methods into constructors as well

The dog objects that the Dog constructor creates are just like the dogs from earlier in the book... except that our newly constructed dogs can't bark (because they don't have a bark method). This is easily fixed because in addition to assigning values to properties in the constructor, we can set up methods too. Let's extend the code to include a bark method:

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}
```

To add a bark method we simply assign a function, in this case an anonymous function, to the property this.bark.

Notice that, just like all the other objects we've created in the past, we use this to refer to the object we're calling the method on.

By the way, as you know, methods in objects are properties too. They just happen to have a function assigned to them.

Now every dog object will also have a bark method that you can invoke.

### Take the bark method for a quick test drive



Enough talking about constructors, let's add the code above to an HTML page, and then add the code below to test it:

```
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    dogs[i].bark();
}
```

Make sure your dog objects bark like they're supposed to.





We've got a constructor to create coffee drinks, but it's missing its methods.

We need a method, `getSize`, that returns a string depending on the number of ounces of coffee:

- 8oz is a small
- 12oz is a medium
- 16oz is a large



We also need a method, `toString`, that returns a string that represents your order, like "You've ordered a small House Blend coffee."

Write your code below, and then test it in the browser. Try creating a few different sizes of coffee. Check your answer before you go on.

```
function Coffee(roast, ounces) {
  this.roast = roast;
  this.ounces = ounces;

}

var houseBlend = new Coffee("House Blend", 12);
console.log(houseBlend.toString());

var darkRoast = new Coffee("Dark Roast", 16);
console.log(darkRoast.toString());
```

← Write the two methods for this constructor here.

Here's our output; yours should look similar.

JavaScript console

```
You've ordered a medium House Blend coffee.
You've ordered a large Dark Roast coffee.
```

## there are no Dumb Questions

**Q:** Why do constructor names start with a capital letter?

**A:** This is a convention that JavaScript developers use so they can easily identify which functions are constructors, and which functions are just regular old functions. Why? Because with constructor functions, you need to use the new operator. In general, using a capital letter for constructors makes them easier to pick out when you're reading code.

**Q:** So, other than setting up the properties of the `this` object, a constructor's just like a regular function?

**A:** If you mean computationally, yes. You can do anything in a constructor you can do in a regular function, like declare and use variables, use for loops, call other functions, and so on. The only thing you don't want to do is return a value (other than `this`) from a constructor because that will cause the constructor to not return the object it's supposed to be constructing.

**Q:** Do the parameter names of a constructor function have to match the property names?

**A:** No. You can use whatever names you want for the parameters. The parameters are just used to hold values that we want to assign to the object's properties to customize the object. What matters is the name of the properties you use for the object. That said, we often do use the same names for clarity, so we know which properties we're assigning by looking at the constructor function definition.

**Q:** Is an object created by a constructor just like an object created with a literal?

**A:** Yes, until you get into more advanced object design, which we'll do in the next chapter.

**Q:** Why do we need `new` to create objects? Couldn't we create an object in a regular function and return it (kind of like we did with `makeCar` in chapter 5)?

**A:** Yes, you could create objects that way, but like we said in the previous answer, there are some extra things that happen when you use `new`. We'll get more into these issues later in this chapter, and again in Chapter 13.

**Q:** I'm still a bit confused by this in the constructor. We're using `this` to assign properties to the object, and we're also using `this` in the methods of the object. Are these the same thing?

**A:** When you call a constructor (to create an object) the value of `this` is set to the new object that's being created so all the code that is evaluated in the constructor applies to that new object.

Later, when you call a method on an object, `this` is set to the object whose method you called. So the `this` in your methods will always refer to the object whose method was called.

**Q:** Is it better to create objects with a constructor than with object literals?

**A:** Both are useful. A constructor is useful when you want to create lots of objects with the same property names and methods. Using them is convenient, reuses code, and provides consistency across your objects.

But sometimes we just need a quick object, perhaps a one-time-use only object, and literals are concise and expressive to use for this.

So it really depends what your needs are. Both are great ways to create an object.

We'll see a good example of this a bit later.



## DANGER ZONE

There's one aspect of constructors you need to be very careful about: don't forget to use the `new` keyword. It's easy to do because a constructor is, after all, a function, and you can call it without `new`. But if you forget `new` on a constructor it can lead to buggy code that is hard to troubleshoot. Let's take a look at what can happen when you forget the `new` keyword...

```
function Album(title, artist, year) {
    this.title = title;
    this.artist = artist;
    this.year = year;
    this.play = function() {
        // code here
    };
}
var darkside = Album("Dark Side of the Cheese", "Pink Mouse", 1971);
darkside.play();
```

↓      Oops we forgot to use new!

← This looks like a well-constructed constructor.

← But maybe that's okay because Album is a function.

← Let's try to call the play method anyway. Oh, this isn't good... ↴

**Uncaught TypeError: Cannot call method 'play' of undefined**

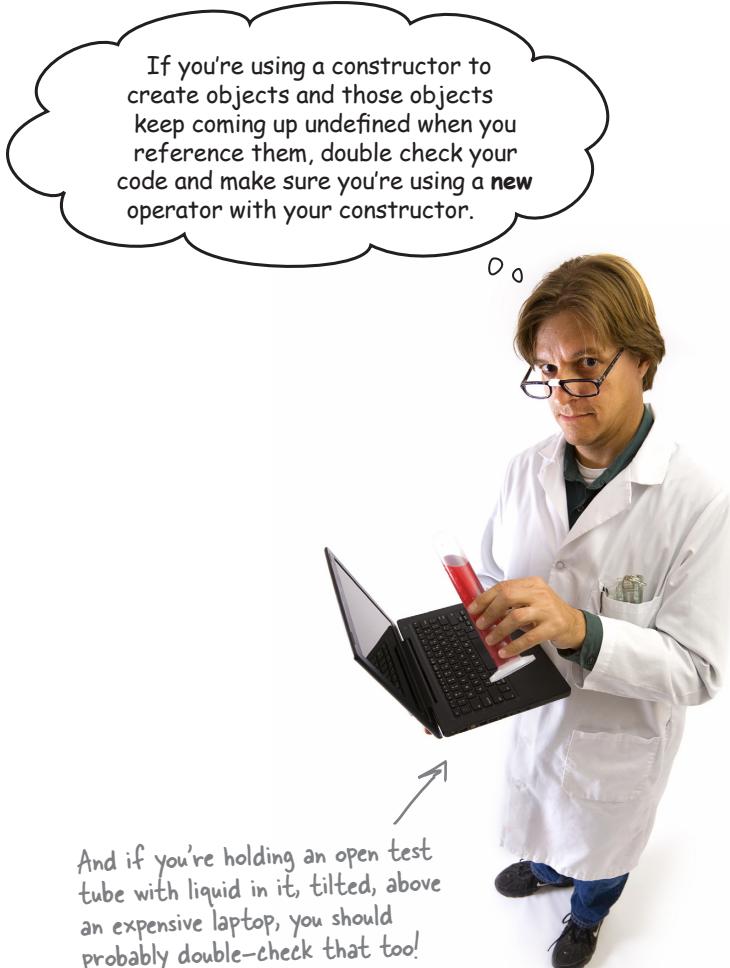
## SAFETY CHECKLIST

Okay, let's read the checklist to see why this might have happened:

- ❑ Remember that `new` first creates a new object before assigning it to `this` (and then calling your constructor function). If you don't use `new`, a new object will never be created.
- ❑ That means any references to `this` in your constructor won't refer to a new album object, but rather, will refer to the global object of your application.
- ❑ If you don't use `new` there's no object to return from the constructor, which means there is no object assigned to the `darkside` variable, so `darkside` is `undefined`. That's why when we try to call the `play` method, we get an error saying the object we're trying to call it on is `undefined`.

The global object is the top-level object, which is where global variables get stored. In browsers, this object is the `window` object.





If you're using a constructor to create objects and those objects keep coming up undefined when you reference them, double check your code and make sure you're using a `new` operator with your constructor.

And if you're holding an open test tube with liquid in it, tilted, above an expensive laptop, you should probably double-check that too!



## The Constructor Exposed

This week's interview:  
Getting to know new

**Head First:** new, where have you been hiding? How did we get to Chapter 12 before seeing you?

**new:** There are still a lot of scripts out there that don't use me, or use me without understanding me.

**Head First:** Why is that?

**new:** Because many scripters just use object literals or copy & paste code that uses me, without understanding how I work.

**Head First:** That's a good point... object literals are convenient, and I myself am not quite clear on when or how to use you just yet.

**new:** Well it's true, I am kind of an advanced feature. After all, to know how to use me, you first have to know how objects work, and how functions work, and how this works... it's a lot to wrap your head around before you even learn about me at all!

**Head First:** Can you give us the elevator pitch about yourself? Now that our readers know about objects, functions, and this, it would be great for them to get motivated for learning about you.

**new:** Let me think for a second... Okay here you go: I'm the operator that operates on constructor functions to create new objects.

**Head First:** Umm, I hate to break it to you but that isn't the best elevator pitch.

**new:** Gimme a break, I'm an operator, not a PR lackey.

**Head First:** Well, you do raise several questions with that pitch. First of all, you're an operator?

**new:** Yup! I'm an operator. Put me in front of a function call and I change everything. An operator operates on its operands. In my case, I have only one operand and that operand is a function call.

**Head First:** Right, so explain exactly how you operate.

**new:** Well, first, I make a new object. Everyone thinks that the constructor function is what does it, but it's actually me. It's a thankless job.

**Head First:** Go on...

**new:** Okay, so then I call the constructor function and make sure that the new object I've created is referenced by the `this` keyword in the body of the function.

**Head First:** Why do you do that?

**new:** So that the statements in the body of the function have a way to refer to the object. After all, the whole point of a constructor function is to extend that object using new properties and methods. If you're using the constructor to create objects like dogs and cars, you're going to want those objects to have some properties, right?

**Head First:** Right. And then?

**new:** Then I make sure that the new object that was created is returned from the constructor. It's a nice convenience so that developers don't have to remember to return it themselves.

**Head First:** It does sound very convenient. Now why would anyone use an object literal after learning you?

**new:** Oh, object literal and I go way back. He's a great guy, and I'd use him in a second if I had to create a quick object. But, you want me when you've got to create a lot of similar objects, when you want to make sure your objects are taking advantage of code reuse, when you want to ensure some consistency, and after you've learned a little more, to support some even more advanced uses.

**Head First:** More advanced? Oh do tell!

**new:** Now now, let's keep these readers focused. We'll talk more in the next chapter.

**Head First:** I think I need to re-read this interview first! Until then...

# It's Production Time!

You've learned your object construction skills just in time because we've just received a big order for cars and we can't be creating them all by hand. We need to use a constructor so we can get the job done on time. We're going to do that by taking the car object literals we've used so far in the book, and using them as a guide for creating a constructor to make cars.



Check out the various kinds of cars we need to build below. Notice we've already taken the liberty of making their properties and methods uniform, so they all match across each car. For now, we won't worry about special options, or toy cars and rocket cars (we'll come back to that later). Go ahead and take a look, and then let's build a constructor that can create car objects for any kind of car that has these property names and methods:



```
var chevy = {
  make: "Chevy",
  model: "Bel Air",
  year: 1957,
  color: "red",
  passengers: 2,
  convertible: false,
  mileage: 1021,
  started: false,

  start: function() {
    this.started = true;
  },

  stop: function() {
    this.started = false;
  },

  drive: function() {
    if (this.started) {
      console.log(this.make + " " +
        this.model + " goes zoom zoom!");
    } else {
      console.log("Start the engine first.");
    }
  }
};
```



```
var cadi = {
  make: "GM",
  model: "Cadillac",
  year: 1955,
  color: "tan",
  passengers: 5,
  convertible: false,
  mileage: 12892,
  : false,
  function() {...},
  function() {...},
  function() {...}

var fiat = {
  make: "Fiat",
  model: "500",
  year: 1957,
  color: "Medium Blue",
  passengers: 2,
  convertible: false,
  mileage: 88000,
  started: false,
  start: function() {...},
  stop: function() {...},
  drive: function() {...}
};
```



```
var taxi = {
  make: "Webville Motors",
  model: "Taxi",
  year: 1955,
  color: "yellow",
  passengers: 4,
  convertible: false,
  mileage: 281341,
  started: false,
  start: function() {...},
  stop: function() {...},
  drive: function() {...}
};
```



## Exercise

Use everything you've learned to create a Car constructor. We suggest the following order:

- ① Start by providing the function keyword (actually we did that for you) followed by the constructor name. Next supply the parameters; you'll need one for each property that you want to supply an initial value for.
- ② Next, assign each property in the object its initial value (make sure you use `this` along with the property name).
- ③ Finally, add in the three car methods: start, drive and stop.

```
function _____(_____){
```

← All your work  
goes here.

```
}
```

Make sure you check all your work with the answer at  
the end of the chapter before proceeding!

## Let's test drive some new cars



Now that we have a way to mass-produce car objects, let's make some, and put them through their paces. Start by putting the Car constructor in an HTML page, then add some test code.

Here's the code we used; feel free to alter and extend it:

>Note: you won't be able to do this unless you did the exercise on the previous page! 😊

First we're using the constructor to create all the cars from Chapter 5.

```
var chevy = new Car("Chevy", "Bel Air", 1957, "red", 2, false, 1021);
var cadi = new Car("GM", "Cadillac", 1955, "tan", 5, false, 12892);
var taxi = new Car("Webville Motors", "Taxi", 1955, "yellow", 4, false, 281341);
var fiat = new Car("Fiat", "500", 1957, "Medium Blue", 2, false, 88000);
```

But why stop there?

```
var testCar = new Car("Webville Motors", "Test Car", 2014, "marine", 2, true, 21);
```

Let's create the book's test drive car! → 

Are you starting to see how easy creating new objects can be with constructors? Now let's take these cars for a test drive:

```
var cars = [chevy, cadi, taxi, fiat, testCar];

for(var i = 0; i < cars.length; i++) {
  cars[i].start();
  cars[i].drive();
  cars[i].drive();
  cars[i].stop();
}
```

Here's the output we got. Did you add your own car to the mix? Try changing what the cars do (like driving before the car is started). Or, maybe you can make the number of times we call the drive method random?

JavaScript console

```
Chevy Bel Air goes zoom zoom!
Chevy Bel Air goes zoom zoom!
GM Cadillac goes zoom zoom!
GM Cadillac goes zoom zoom!
Webville Motors Taxi goes zoom zoom!
Webville Motors Taxi goes zoom zoom!
Fiat 500 goes zoom zoom!
Fiat 500 goes zoom zoom!
Webville Motors Test Car goes zoom zoom!
Webville Motors Test Car goes zoom zoom!
```

# Don't count out object literals just yet

We've had some discussion of object constructors versus object literals, and mentioned that object literals are still quite useful, but you really haven't seen a good example of that. Well, let's do a little reworking of the Car constructor code, so you can see where using some object literals actually cleans up the code and makes it more readable and maintainable.

Let's look at the Car constructor again and see how we might be able to clean it up a bit.

Notice we're using a lot of parameters here. We count seven.

The more we add (and we always end up adding more as the requirements for objects grow), the harder this is to read.

```
function Car(make, model, year, color, passengers, convertible, mileage) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.color = color;
    this.passengers = passengers;
    this.convertible = convertible;
    this.mileage = mileage;
    this.started = false;

    this.start = function() {
        this.started = true;
    };
    //rest of the methods here
}
```

And when we write code that calls this constructor we have to make sure we get the arguments all in exactly the right order.

So the problem we're highlighting here is that we have a heck of a lot of parameters in the Car constructor, making it difficult to read and maintain. It's also difficult to write code to call this constructor. While that might seem like a minor inconvenience, it actually causes more bugs than you might think, and not only that, they're often nasty bugs that are hard to diagnose at first.

However, there is a common technique that we can use when passing all these arguments that can be used for any function, whether or not it's a constructor. The technique works like this: take all your arguments, throw them in an object literal, and then pass that literal to your function—that way you're passing all your values in one container (the literal object) and you don't have worry about matching the order of your arguments and parameters.

Let's rewrite the code to call the Car constructor, and then do a slight rework of the constructor code to see how this works.

They're hard to diagnose because if you switch two variables, the code is still syntactically correct, but it doesn't function correctly because you've switched two values.

Or if you leave out a value, all kinds of craziness can ensue!

## Rewiring the arguments as an object literal

Let's take the call to the Car constructor and rework its arguments into an object literal:



All you need to do is take each argument and place it in an object literal with an appropriate property name. We use the same property names used in the constructor.

```
var cadi = new Car("GM", "Cadillac", 1955, "tan", 5, false, 12892);
```

```
var cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};
```

We've kept the same order, but there is no reason you'd have to.

And then we can rewrite the call to the Car constructor like this:

```
var cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};
```

Wow, talk about a makeover.  
Not only is this much cleaner,  
it's a lot more readable, at  
least in our humble opinion.

```
var cadi = new Car(cadiParams); ← Now we're passing a single argument to the Car constructor.
```

But we're not done yet because the constructor itself is still expecting seven arguments, not one object. Let's rework the constructor code, and then we'll give this a test.

# Reworking the Car constructor

Now you need to remove all the individual parameters in the Car constructor and replace them with properties from the object that we're passing in. We'll call that parameter `params`. You also need to rework the code a bit to use this object. Here's how:

```
var cadiParams = {make: "GM",
                  model: "Cadillac",
                  year: 1955,
                  color: "tan",
                  passengers: 5,
                  convertible: false,
                  mileage: 12892};
```

No changes here, we've just reproduced the object literal and the call to the Car constructor from the previous page.

```
var cadi = new Car(cadiParams);
```

function Car(params) {
 this.make = params.make;
 this.model = params.model;
 this.year = params.year;
 this.color = params.color;
 this.passengers = params.passengers;
 this.convertible = params.convertible;
 this.mileage = params.mileage;
 this.started = false;

 this.start = function() {
 this.started = true;
 };
 this.stop = function() {
 this.started = false;
 };
 this.drive = function() {
 if (this.started) {
 alert("Zoom zoom!");
 } else {
 alert("You need to start the engine first.");
 }
 };
}

First things first. We'll replace the seven parameters of the Car constructor with one parameter, for the object we're passing in.

Then for each reference to a parameter, we substitute the corresponding property from the object passed into the function.

In our methods we never use a parameter directly. It wouldn't make sense to because we always want to use the object's properties (which we do using the `this` variable). So, no changes are needed to this code at all.

## Test drive



Update the `cadi` and all your other cars, and test your code.

```
cadi.start();
cadi.drive();
cadi.drive();
cadi.stop();
```



Copy the Car and Dog constructors into one file, and then add the code below along with it. Give this a run and capture the output.

```
var limoParams = {make: "Webville Motors",
                  model: "limo",
                  year: 1983,
                  color: "black",
                  passengers: 12,
                  convertible: true,
                  mileage: 21120};

var limo = new Car(limoParams);
var limoDog = new Dog("Rhapsody In Blue", "Poodle", 40);

console.log(limo.make + " " + limo.model + " is a " + typeof limo);
console.log(limoDog.name + " is a " + typeof limoDog);
```

You'll find the Dog constructor  
on page 530.

← Put the output here.



Say someone handed you an object and you wanted to know what type of object it was (is it a Car? a Dog? Superman?), or you wanted to see if it was the same type as another object. Would the `typeof` operator be helpful?

there are no  
**Dumb Questions**

**Q:** Remind me what `typeof` does again?

**A:** The `typeof` operator returns the type of its operand. If you pass it a string you'll get back "string", if you pass it an object you'll get back "object" and so on. You can pass it any type: a number, a string, a boolean, or a more complex type like an object or function. But `typeof` can't be more specific and tell you the object is a dog or a car.

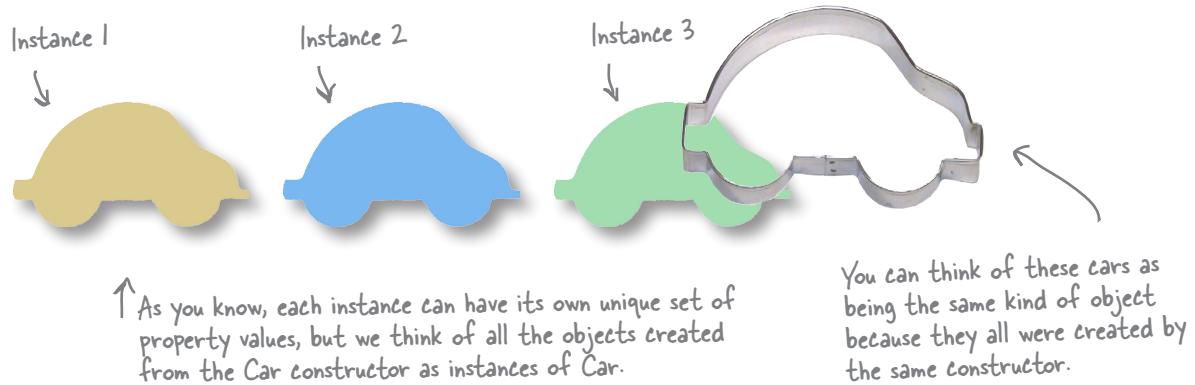
**Q:** So if `typeof` can't tell me that my object is a dog or car, how do I determine what is what?

**A:** Many other object-oriented languages, like Java or C++, have a strong notion of object typing. In those languages you can examine an object and determine exactly what type of object it is. But, JavaScript treats objects and their types in a looser, more dynamic way. Because of this, many developers have jumped to the conclusion that JavaScript has a less powerful object system, but the truth is, its object system is actually more general and flexible. Because JavaScript's type system is more dynamic, it's a little more difficult to determine if an object is a dog or a car, and it depends on what you think a dog is or a car is. However, we have another operator that can give us a little more information... so continue reading.

# Understanding Object Instances

You can't look at a JavaScript object and determine that it is an object of a specific type, like a dog or a car. In JavaScript, objects are dynamic structures, and the type of all objects is just "object," no matter what properties and methods it has. But we can get some information about an object if we know the *constructor* that created the object.

Remember that each time you call a constructor using the new operator, you are creating a new instance of an object. And, if you used, say, the Car constructor to do that, then we say, informally, that the object is a car. More formally, we say that object is an *instance* of a Car.



Now saying an object is an instance of some constructor is more than just talk. We can actually write code to inspect the constructor that made an object with the instanceof operator. Let's look at some code:

```
var cadiParams = {make: "GM", model: "Cadillac", year: 1955, color: "tan",
    passengers: 5, convertible: false, mileage: 12892};

var cadi = new Car(cadiParams);

if (cadi instanceof Car) {
    console.log("Congrats, it's a Car!");
}
```

The instanceof operator returns true if the object was created by the specified constructor.

In this case we're saying "Is the cadi object an instance that was created by the Car constructor?"

As it turns out, one of the things the new operator does behind the scenes when the object is created is to store information that allows it to determine, at any time, the constructor that created the object. And instanceof uses that to determine if an object is an instance of a certain constructor.

It's a bit more complicated than we're describing here, but we'll talk about that in the next chapter.





## Exercise

We need a function named dogCatcher that returns true if the object passed to it is a dog, and false otherwise. Write that function and test it with the rest of the code below. Don't forget to check your answer at the end of the chapter before you go on!

```
function dogCatcher(obj) {
    // Add your code here
    // to implement the
    // dogCatcher function.

}

// And here's your test code.

function Cat(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}

var meow = new Cat("Meow", "Siamese", 10);
var whiskers = new Cat("Whiskers", "Mixed", 12);

var fido = {name: "Fido", breed: "Mixed", weight: 38};

function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}

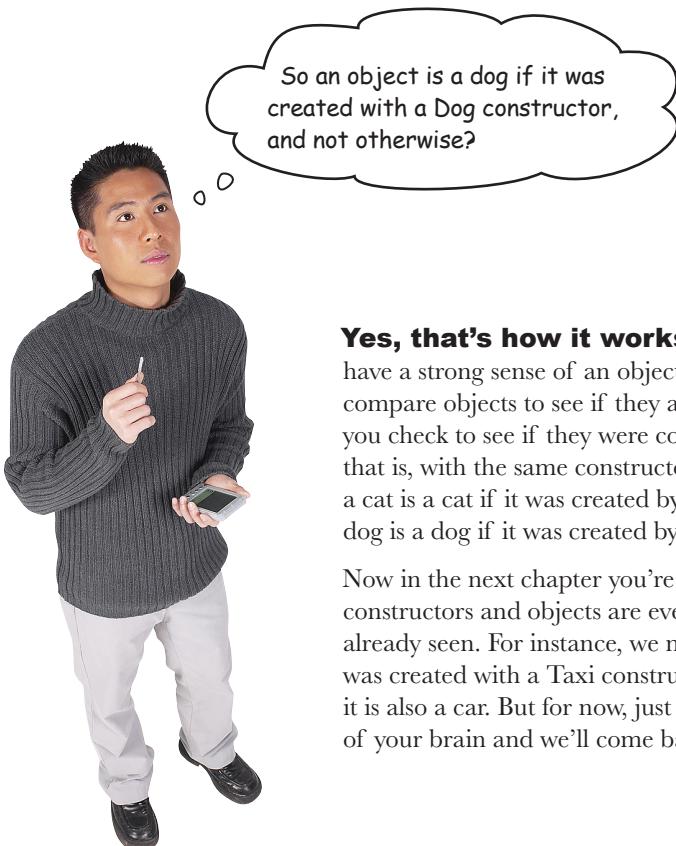
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [meow, whiskers, fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    if (dogCatcher(dogs[i])) {
        console.log(dogs[i].name + " is a dog!");
    }
}
```

Add your code here  
to implement the  
dogCatcher function.

✓ And here's your test code.





**Yes, that's how it works.** JavaScript doesn't have a strong sense of an object's type, so if you need to compare objects to see if they are both cats or both dogs, you check to see if they were constructed the same way—that is, with the same constructor function. As we've said, a cat is a cat if it was created by the Cat constructor, and a dog is a dog if it was created by the Dog constructor.

Now in the next chapter you're going to see JavaScript constructors and objects are even more flexible than we've already seen. For instance, we might have an object that was created with a Taxi constructor, and yet we know that it is also a car. But for now, just stash that idea in the back of your brain and we'll come back to it later.

## Even constructed objects can have their own independent properties

We've talked a lot about how to use constructors to create consistent objects—objects that have the same set of properties and the same methods. But what we haven't mentioned is that using constructors still doesn't prevent us from changing an object into something else later, because after an object has been created by a constructor, it can be altered.

What exactly are we talking about? Remember when we introduced object literals? We looked at how we could add and delete properties after the object was created. You can do the same with objects created from constructors:

Here's our dog Fido, created with the Dog constructor.

```
var fido = new Dog("Fido", "Mixed", 38);  
fido.owner = "Bob"; ← We can add a new property just by assigning it a value in our object.  
delete fido.weight; ← Or we can get rid of a property by using the delete operator.
```

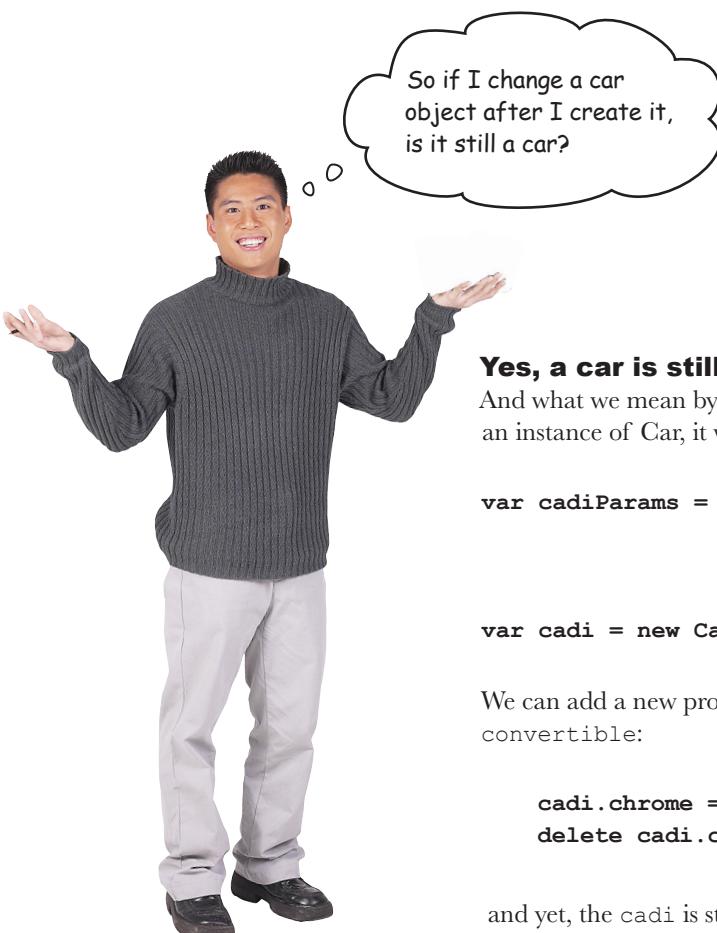
You can even add new methods if you like:

To add a method just assign the method to a new property name in the object.

```
fido.trust = function(person) {  
    return (person === "Bob");  
}; ← Anonymous function alert! See, they're everywhere!
```

Notice that here we're changing only the fido object. If we add a method to fido, only fido has that method. No other dogs have it:

```
var notBite = fido.trust("Bob"); ← This code works because trust is defined in the fido object. So notBite is true.  
  
var spot = new Dog("Spot", "Chihuahua", 10); } This code doesn't work because spot doesn't have  
notBite = spot.trust("Bob"); } a method trust, resulting in: "TypeError: Object #<Dog> has no method 'trust'"
```

**Yes, a car is still a car, even if you change it later.**

And what we mean by that is if you check to see if your object is still an instance of Car, it will be. For instance, if we create a car object:

```
var cadiParams = {make: "GM", model: "Cadillac",
                  year: 1955, color: "tan",
                  passengers: 5, convertible: false,
                  mileage: 12892};
var cadi = new Car(cadiParams);
```

We can add a new property `chrome` and delete the property `convertible`:

```
cadi.chrome = true;
delete cadi.convertible;
```

and yet, the `cadi` is still a car:

`cadi instanceof Car` ↪ Evaluates to true.

This is what we meant earlier when we said that JavaScript has a dynamic type system.

Now is it really a car in practical terms? What if we deleted every property in the object? Would it still be a car? The `instanceof` operator would tell us yes. But judging by our own terms, probably not.

Chances are, you won't often want to use a constructor to create an object and then later change it into something that's unrecognizable as an object created by that constructor. In general, you'll use constructors to create objects that are fairly consistent. But if you need objects that are more flexible, well, JavaScript can handle that. It's your job as a code designer to decide how to use constructors and objects in a way that makes sense for you (and don't forget your coworkers).

## Real World Constructors

JavaScript comes with a set of constructors for instantiating some handy objects—like objects that know how to deal with dates and times, objects that are great at finding patterns in text, and even objects that will give you a new perspective on arrays. Now that you know how constructors work, and also how to use the `new` keyword, you’re in a great position to make use of these constructors, or more importantly the objects they create. Let’s just take a quick dip into a couple, and then you’ll be all ready to go out and explore them on your own.

Let’s start with JavaScript’s built-in date object. To get one we just use its constructor:

```
var now = new Date();
```

Creates a new date representing  
the current date and time.

Calling the `Date` constructor gives you back an instance of `Date` that represents the current local date and time. With a `Date` object in hand, you can then use its methods to manipulate dates (and times) and also retrieve various properties of a date and time. Here are a few examples:

```
var dateString = now.toString();
```

Returns a string that represents  
the date, like "Thu Feb 06 2014  
17:29:29 GMT-0800 (PST)".

```
var theYear = now.getFullYear();
```

Returns the year in the date.

```
var theDayOfWeek = now.getDay();
```

Returns a number for the day of the week  
represented by the date object, like 1 (for Monday).

You can easily create date objects representing any date and time by passing additional arguments to the `Date` constructor. For instance, say you need a date object representing “May 1, 1983”, you can do that with:

```
var birthday = new Date("May 1, 1983");
```

You can pass a simple date string  
to the constructor like this.

And you can get even more specific by including a time:

```
var birthday = new Date("May 1, 1983 08:03 pm");
```

Now, we’re including a  
time in the string too.

We are, of course, just giving you a flyby of the `Date` object; you’ll want to check out its full set of properties and methods in *JavaScript: The Definitive Guide*.

These built-in objects  
really save me time. Heck, these  
days I get home early enough to  
watch a little "Golden Girls."



# The Array object

Next up, another interesting built-in object: the array object. While we've been creating arrays using the square bracket notation [1, 2, 3], you can create arrays using a constructor too:

```
var emptyArray = new Array();
```

Creates an empty array with length zero.

Here, we're creating a new, empty array object. And at any time we can add items to it, like this:

```
emptyArray[0] = 99;
```

This should look familiar. This is the same way we've always added items to an array.

We can also create array objects that have a specific size. Say we want an array with three items:

```
var oddNumbers = new Array(3);
oddNumbers[0] = 1;
oddNumbers[1] = 3;
oddNumbers[2] = 5;
```

We create an array of length three, and fill it in with values after we create it.

Here we've created an array of length three. Initially the three items in `oddNumbers` are undefined, but we then set each item in the array to a value. You could easily add more items to the array if you wanted.

None of this should be shockingly different than what you're used to. Where the array object gets interesting is in its set of methods. You already know about array's `sort` method, and here are a few other interesting ones:

```
oddNumbers.reverse();
```

This reverses all the values in the array (so we have 5, 3, 1 in `oddNumbers` now). Notice, the method changes the original array.

```
var aString = oddNumbers.join(" - ");
```

The join method creates a string from the values in `oddNumbers` placing a " - " between the values, and returns that string. So this returns the string "5 - 3 - 1".

```
var areAllOdd = oddNumbers.every(function(x) {
  return ((x % 2) !== 0);
});
```

The every method takes a function and tests each value of the array to see if the function returns true or false when called on that value. If the function returns true for all the array items, then the result of the every method is true.

Again, that's just the tip of the iceberg, so take a look at *JavaScript: The Definitive Guide* to fully explore the array object. You've got all the knowledge you need to take it on.



**Good catch.** The bracket notation, [ ], that you've been using to create arrays is actually just a shorthand for using the Array constructor directly. Check out these two equivalent ways of creating empty arrays:

```
var items = new Array(); ← These do the same thing. The  
var items = []; ← bracket notation is supported in the  
JavaScript language to make your life easier when creating arrays.
```

Likewise, if you write code like this:

```
var items = ["a", "b", "c"]; ← We call this array literal syntax.
```

That's just a shorthand for using the constructor in another way:

```
var items = new Array("a", "b", "c");
```

⌚ If you pass more than one argument, this creates an array holding the values you pass it.

And, the objects created from the literal notation or by using the constructor directly are the same, so you can use methods on either one.

You might be asking why you'd ever use the constructor rather than the literal notation. The constructor comes in handy when you need to create an array of a specific size you determine at runtime, and then add items to it later, like this:

```
var n = getNumberOfWidgetsFromDatabase();  
var widgets = new Array(n); ← This code presumably uses big arrays that we won't know the size of until runtime.  
for(var i=0; i < n; i++) {  
    widgets[i] = getDatabaseRecord(i);  
}
```

So, for creating a quick array, using the array literal syntax to create your array objects works wonderfully, but using the Array constructor might make sense when you're creating the array programmatically. You can use either or both as much as you want.

# Even more fun with built-in objects

Date and array aren't the only built-in objects in JavaScript. There are lots of other objects that come with the language you might find handy at times. Here's a short list (there are more, so search online for "JavaScript's standard built-in objects" if you're curious!).

## Object

By using the Object constructor you can create objects. Like arrays, the object literal notation {} is equivalent to using new Object(). More on this later.

## RegExp

Use this constructor to create regular expression objects, which allow you to search for patterns, even complex ones, in text.

## Math

This object has properties and methods for doing math stuff. Like Math.PI and Math.random().

## Error

This constructor creates standard error objects that are handy when catching errors in your code.

## there are no Dumb Questions

**Q:** I'm confused by how the Date and Array constructors work: they seem to support zero or more arguments. Like with Date, if I don't provide an argument, then I get today's date, but I can also pass arguments to get other dates. How does that work?

**A:** Right, good catch. It's possible to write functions that do different things based on the number of arguments. So if the Array constructor has zero arguments, the constructor knows it is creating an empty array; if it has one argument it knows that's the size of the array, and if it has more, then those arguments are all initial values.

**Q:** Can we do that with our constructors?

**A:** Of course. This is something we haven't covered, but every function gets passed an arguments object that contains all the arguments passed to the function. You can use this to determine what was passed

and act appropriately (check the appendix for more on the arguments object). There are other techniques based on checking to see which of your parameters is set to undefined.

**Q:** We used Math earlier in the book. Why don't I have to say "new Math" to instantiate a math object before I use it?

**A:** Great question. Actually, Math is not a constructor, or even a function. It's an object. As you know, Math is a built-in object that you can use to do things like get the value of pi (with Math.PI) or generate a random number (with Math.random). Think of Math as just like an object literal that has a bunch of useful properties and methods in it, built-in for you to use whenever you write JavaScript code. It just happens to have a capital first letter to let you know that it's built-in to JavaScript.

**Q:** I know how to check if an object is an instance of a constructor name, but how do I write the code to ask if two objects have the same constructor?

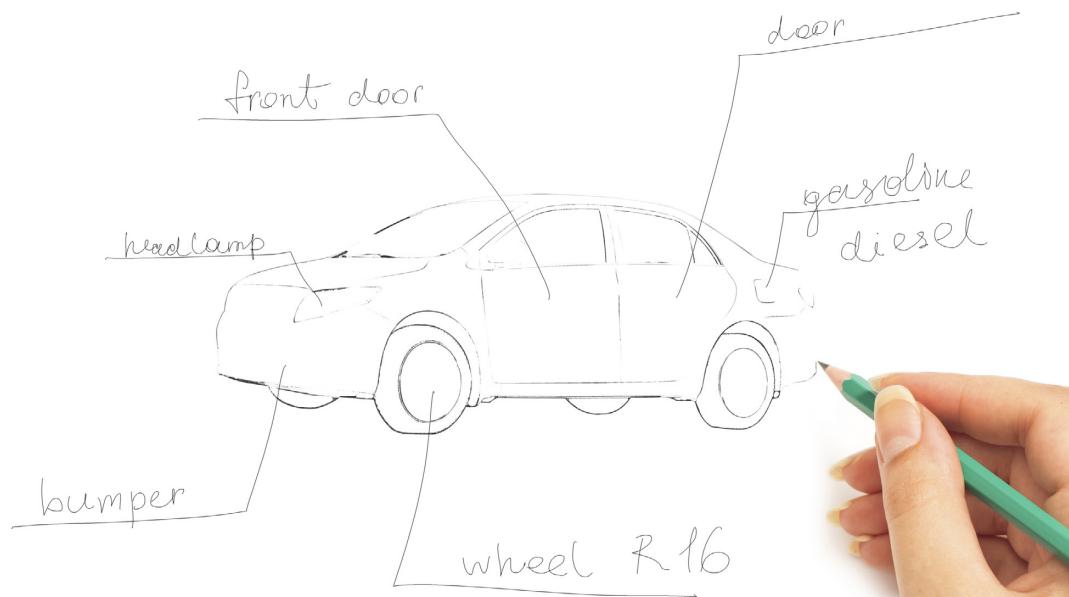
**A:** You can check to see if two objects have the same constructor like this:

```
((fido instanceof Dog) &&
  (spot instanceof Dog))
```

If this expression results in true, then fido and spot were indeed created by the same constructor.

**Q:** If I create an object with an object literal, what is it an instance of? Or is it not an instance of anything?

**A:** An object literal is an instance of Object. Think of Object as the constructor for the most generic kind of object in JavaScript. You'll learn much more about how Object figures into JavaScript's object system in the next chapter.



**Webville Motors** is revolutionizing car production by creating all their cars from a prototype car. The prototype gives you all the basics you need: a way to start, drive and stop it along with a couple properties like the make and year it was manufactured—but the rest is up to you. Want it to be red or blue? No problem, just customize it. Need for it to have a fancy stereo? No problem, go crazy, add it.

So this is your opportunity to design your perfect car. Create a CarPrototype object below, and make the car of your dreams. Check out our design at the end of the chapter before moving on.



} ← And customize  
the prototype  
here.

```
function CarPrototype() {  
    this.make = "Webville Motors";  
    this.year = 2013;  
    this.start = function() {...};  
    this.stop = function() {...};  
    this.drive = function() {...};  
}
```

Oh, and where are we going with this? You'll find out in the next chapter! By the way, you're done with this chapter... Oh, but there's still the bullet points and the crossword puzzle to do!



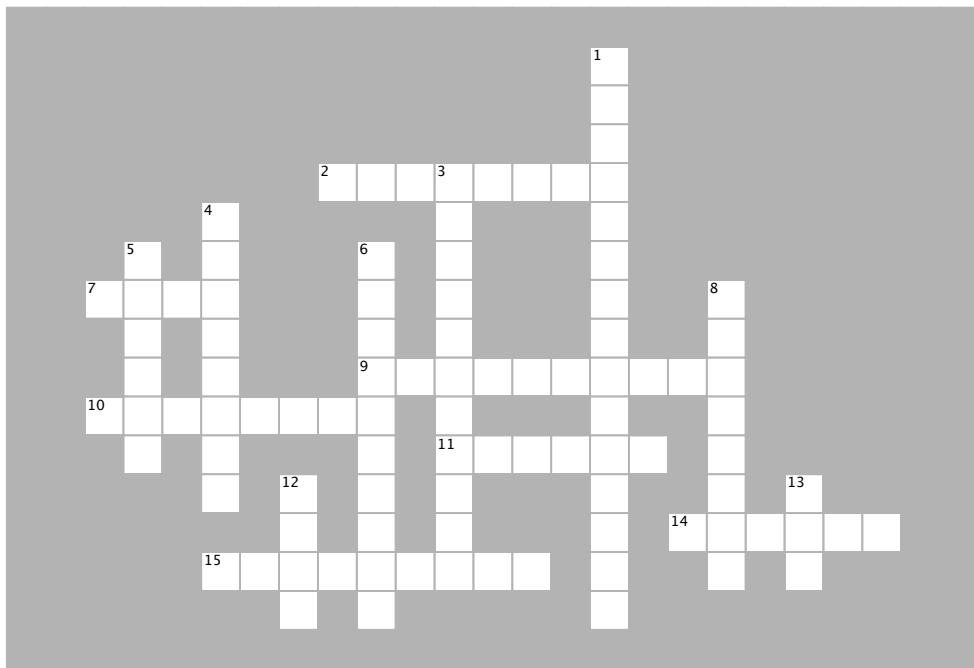
## BULLET POINTS

- An **object literal** works well when you need to create a small number of objects.
- A **constructor** works well when you need to create many similar objects.
- Constructors are functions that are meant to be used with the **new** operator. We capitalize the names of constructors by convention.
- Using a constructor we can create objects that are consistent, having the same property names and methods.
- Use the **new** operator with a constructor function call to create an object.
- When you use **new** with a constructor function call, it creates a new, empty object, which is assigned to **this** within the body of the constructor.
- Use **this** in a constructor function to access the object being constructed and add properties to the object.
- A new object is returned automatically by the constructor function.
- If you forget to use **new** with a constructor, no object is created. This will cause errors in your code that can be difficult to debug.
- To customize objects, we pass arguments to a constructor, and use those values to initialize the properties of the object being created.
- If a constructor has a lot of parameters, consider consolidating them into one object parameter.
- To know if an object was created by a specific constructor, use the **instanceof** operator.
- You can modify an object that was created by a constructor just like you can modify an object literal.
- JavaScript comes with a number of constructors you can use to create useful objects like date objects, regular expressions and arrays.



# JavaScript cross

Construct some new connections in your brain with this crossword puzzle.



## ACROSS

2. A constructor is a \_\_\_\_\_.
7. If you want to save my birthday in a variable, you'll need a \_\_\_\_\_ constructor.
9. You can use an object literal to pass arguments to a constructor when the constructor has lots of these.
10. When you create an object from a constructor, we say it is an \_\_\_\_\_ of the constructor.
11. A constructor is a bit like a \_\_\_\_\_ cutter.
14. The constructor function returns the newly constructed \_\_\_\_\_.
15. If you forget to use new with a constructor, you might see a \_\_\_\_\_.

## DOWN

1. Constructor syntax is a bit \_\_\_\_\_.
3. You can add a property to an object created by a \_\_\_\_\_ whenever you want.
4. new is an \_\_\_\_\_, not a PR lackey.
5. The Webville Motors test car comes in this color.
6. Using a constructor, we can make our cars so they have all the same \_\_\_\_\_.
8. Never hold a \_\_\_\_\_ over your laptop.
12. The limo and the limoDog are the same \_\_\_\_\_.
13. To create an object with a constructor, you use the \_\_\_\_\_ operator.



## Sharpen your pencil Solution

We need your help. We've been using object literals to create ducks. Given what you learned above, can you write a constructor to create ducks for us? You'll find one of our object literals below to base your constructor on. Here's our solution.

```
var duck = {
    type: "redheaded",
    canFly: true
}

function Duck(type, canFly) {
    this.type = type;
    this.canFly = canFly;
}
```

Here's an example duck object literal.

Write a constructor for creating ducks.

P.S. We know you haven't fully figured out how this all works yet, so for now concentrate on the syntax.



## Exercise SOLUTION

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}

var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    var size = "small";
    if (dogs[i].weight > 10) {
        size = "large";
    }
    console.log("Dog: " + dogs[i].name
                + " is a " + size
                + " " + dogs[i].breed);
}
```

Get some quick hands on experience to help this all sink in. Go ahead and put this code in a page and give it a test drive. Write your output here.

### JavaScript console

```
Dog: Fido is a large Mixed
Dog: Fluffy is a large Poodle
Dog: Spot is a small Chihuahua
```



# BE the Browser Solution

Below, you'll find JavaScript code with some mistakes in it.

Your job is to play like you're the browser and find the errors in the code. Here's our solution.

We don't need "var" in front of this. We're not declaring new variables, we're adding properties to an object.

We're using commas instead of semicolons. Remember, in the constructor we use normal statements rather than comma separated property name/value pairs.

Needs a space between new and the constructor name.

```
function widget(partNo, size) {
    var this.no = partNo;
    var this.breed = size;
}
```

If `widget` is to be a constructor, it needs a capital letter for `W`. That won't cause an error, but it's a good convention to follow.

Also, by convention we usually name the parameters the same as the property names. So probably `this.partNo` and `this.size` would be better.

```
function FormFactor(material, widget) {
    this.material = material,
    this.widget = widget,
    return this;
}
```

We're returning `this` and we don't need to. The constructor will do it for us. This statement won't cause an error, but it's not necessary.

```
var widgetA = widget(100, "large");
var widgetB = new widget(101, "small");
var formFactorA = newFormFactor("plastic", widgetA);
var formFactorB = new ForumFactor("metal", widgetB);
```

Forgot new!

Misspelled the name of the constructor.



## Exercise Solution

We've got a constructor to create coffee drinks, but it's missing its methods.

We need a method, getSize, that returns a string depending on the number of ounces of coffee:

- 8oz is a small
- 12oz is a medium
- 16oz is a large

We also need a method, toString, that returns a string specifying your order.

Write your code below, and then test it in the browser. Try creating a few different sizes of coffee. Here's our solution.



```
function Coffee(roast, ounces) {
    this.roast = roast;
    this.ounces = ounces;
    this.getSize = function() {
        if (this.ounces === 8) {
            return "small";
        } else if (this.ounces === 12) {
            return "medium";
        } else if (this.ounces === 16) {
            return "large";
        }
    };
    this.toString = function() {
        return "You've ordered a " + this.getSize() + " " +
            + this.roast + " coffee.";
    };
}
```

Remember, this will be the object whose method we call. So if we call houseBlend.size, then this will be the houseBlend object.

The getSize method looks at the ounces property of the object, and returns the corresponding size string.

The toString method just returns a string description of the object. It uses the getSize method to get the size of the coffee.

We create two coffee objects and call the toString method and display the resulting string.

```
var houseBlend = new Coffee("House Blend", 12);
console.log(houseBlend.toString());

var darkRoast = new Coffee("Dark Roast", 16);
console.log(darkRoast.toString());
```

Here's our output; yours should look similar.

### JavaScript console

```
You've ordered a medium House Blend coffee.
You've ordered a large Dark Roast coffee.
```



Use everything you've learned to create a Car constructor. We suggest the following order:

- ① Start by providing the function keyword (actually we did that for you) followed by the constructor name. Next supply the parameters; you'll need one for each property that you want to supply an initial value for.
- ② Next, assign each property in the object its initial value (make sure you use this along with the property name).
- ③ Finally, add in the three car methods: start, drive and stop.

Here's our solution.

The constructor name is Car.

```

① function Car(make, model, year, color, passengers, convertible, mileage) {
    ↴
    ↴ And seven parameters, one for each
    ↴ property we want to customize.

② this.make = make;
    this.model = model;
    this.year = year;
    this.color = color;
    this.passengers = passengers;
    this.convertible = convertible;
    this.mileage = mileage;
    this.started = false;   ↴ The started property is
                           ↴ just initialized to false.

③ this.start = function() {
    this.started = true;
};
this.stop = function() {
    this.started = false;
};
this.drive = function() {
    if (this.started) {
        alert("Zoom zoom!");
    } else {
        alert("You need to start the engine first.");
    }
};
}

```

Each property of the new car object that's customized with a parameter is set to the parameter name. Notice we're using the same name for the property and the parameter by convention.

The methods are exactly the same as before, but now they're assigned to properties in the object with slightly different syntax because we're in a constructor not an object literal.



Copy the Car and Dog constructors into one file, and then add the code below along with it. Give this a run and capture the output.  
Here's our result:

```
var limoParams = {make: "Webville Motors",
                  model: "limo",
                  year: 1983,
                  color: "black",
                  passengers: 12,
                  convertible: true,
                  mileage: 21120};

var limo = new Car(limoParams);
var limoDog = new Dog("Rhapsody In Blue", "Poodle", 40);

console.log(limo.make + " " + limo.model + " is a " + typeof limo);
console.log(limoDog.name + " is a " + typeof limoDog);
```

JavaScript console  
Webville Motors limo is a object  
Rhapsody In Blue is a object

← What we got.



## Exercise Solution

We need a function, `dogCatcher`, that returns true if the object passed to it is a Dog, and false otherwise. Write that function and test it with the rest of the code below. Here's our solution:

```

function dogCatcher(obj) {
    if (obj instanceof Dog) {
        return true;
    } else {
        return false;
    }
}

function Cat(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}
var meow = new Cat("Meow", "Siamese", 10);
var whiskers = new Cat("Whiskers", "Mixed", 12);

var fido = {name: "Fido", breed: "Mixed", weight: 38};

function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var dogs = [meow, whiskers, fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
    if (dogCatcher(dogs[i])) {
        console.log(dogs[i].name + " is a dog!");
    }
}

```

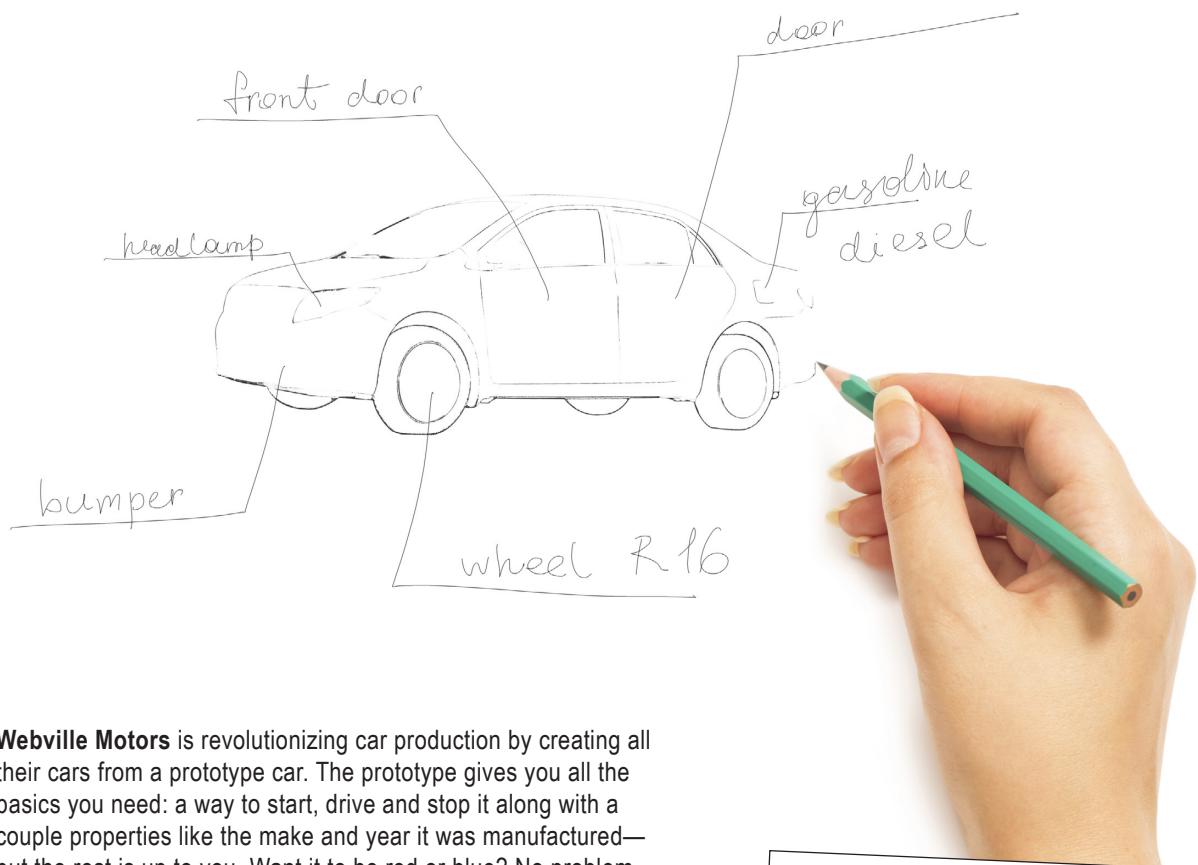
Or more succinctly:

```

function dogCatcher(obj) {
    return (obj instanceof Dog);
}

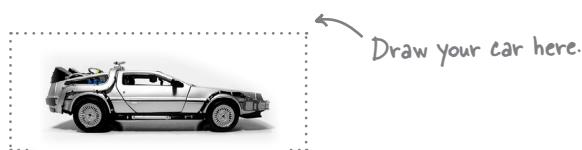
```





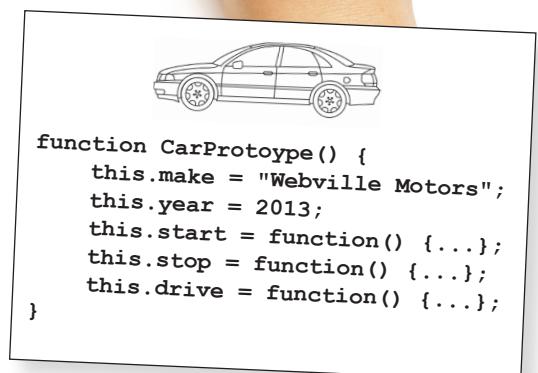
**Webville Motors** is revolutionizing car production by creating all their cars from a prototype car. The prototype gives you all the basics you need: a way to start, drive and stop it along with a couple properties like the make and year it was manufactured—but the rest is up to you. Want it to be red or blue? No problem, just customize it. Need for it to have a fancy stereo? No problem, go crazy, add it.

So this is your opportunity to design your perfect car. Check out our design below.



```
var taxi = new CarPrototype();
taxi.model = "Delorean Remake";
taxi.color = "silver";
taxi.currentTime = new Date();
taxi.fluxCapacitor = {type: "Mr. Fusion"};
taxi.timeTravel = function(date) {...};
```

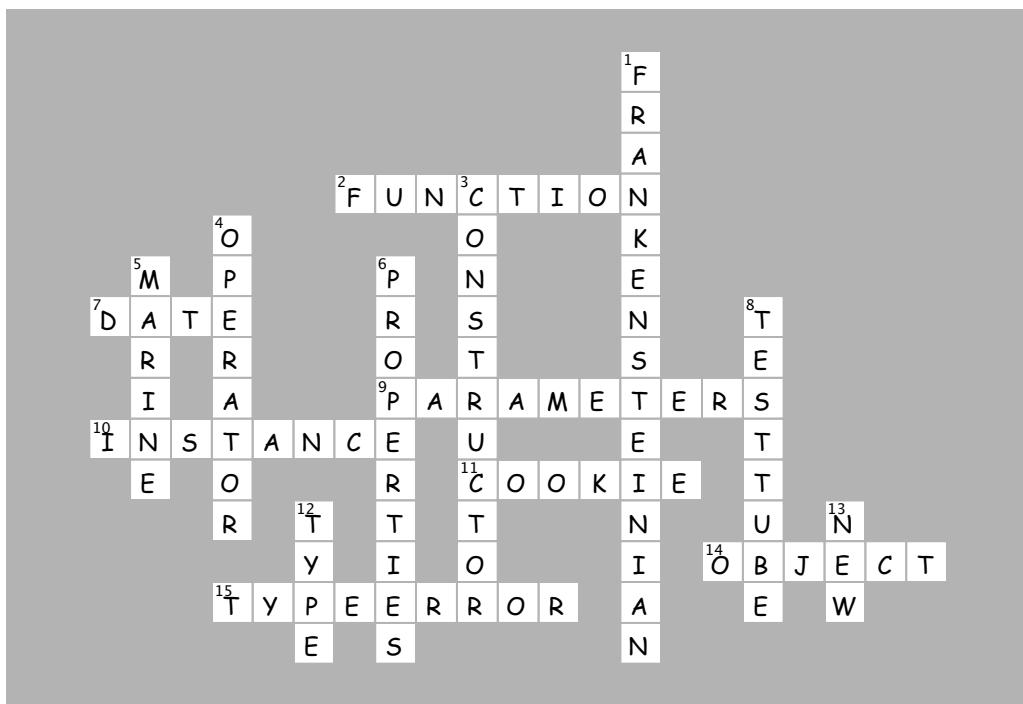
} ← And customize the prototype here.



Oh, and where are we going with this? You'll find out in the next chapter! By the way, you're done with this chapter now.



# JavaScript cross Solution



# *Extra strength objects*



### **Learning how to create objects was just the beginning.**

It's time to put some muscle on our objects. We need more ways to create **relationships** between objects and to **share code** among them. And, we need ways to extend and enhance existing objects. In other words, we need more tools. In this chapter, you're going to see that JavaScript has a very powerful **object model**, but one that is a bit different than the status quo object-oriented language. Rather than the typical class-based object-oriented system, JavaScript instead opts for a more powerful **prototype** model, where objects can inherit and extend the behavior of other objects. What is that good for? You'll see soon enough. Let's get started...



**If you're used to Java, C++, or any language based on classical object-oriented programming let's have a quick chat.**

And if you aren't... what, you got a date? Take a seat, and go along for the ride—you might just learn something as well.

We'll give it to you straight: JavaScript doesn't have a classical object-oriented model, where you create objects from classes. In fact, *JavaScript doesn't have classes at all*. In JavaScript, objects inherit behavior *from other objects*, which we call *prototypal inheritance*, or inheritance based on prototypes.

JavaScript gets a lot of groans (and confused looks) from those trained in object-oriented programming, but know this: prototype-based languages are more general than classical object oriented ones. They're more flexible, efficient and expressive. So expressive that if you wanted to, you could use JavaScript to implement classical inheritance.

So, if you are trained in the art of classical object-oriented programming, sit back, relax, open your mind and be ready for something a little different. And if you have no idea what we're talking about when we say "classical object-oriented programming," that just means you're starting fresh, which is often a very good thing.

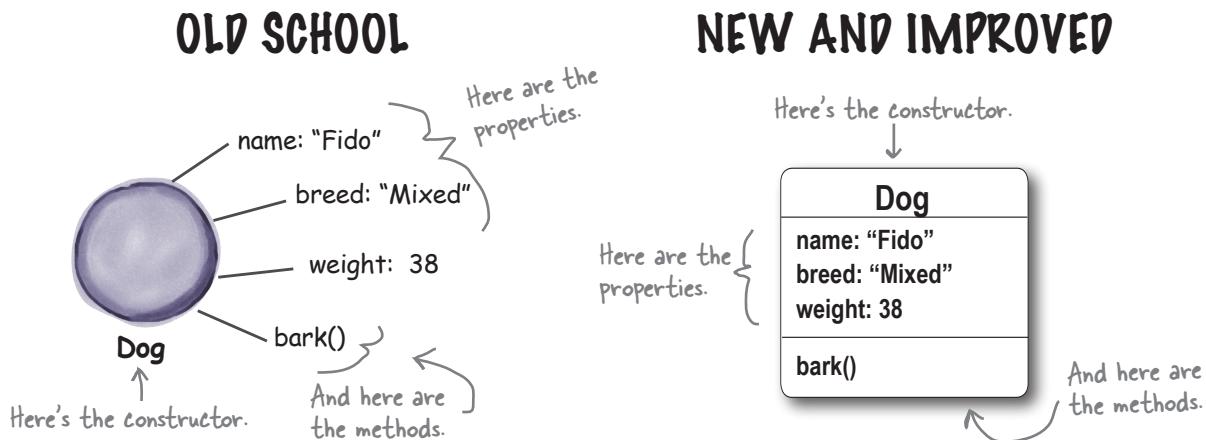
This may change in the future: the next version of JavaScript may add classes. So keep an eye out on [wickedlysmart.com/hfjs](http://wickedlysmart.com/hfjs) for the latest on this.

Left to the reader as an exercise.

# Hey, before we get started, we've got a better way to diagram our objects

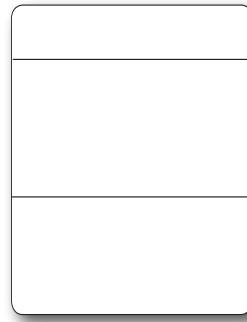
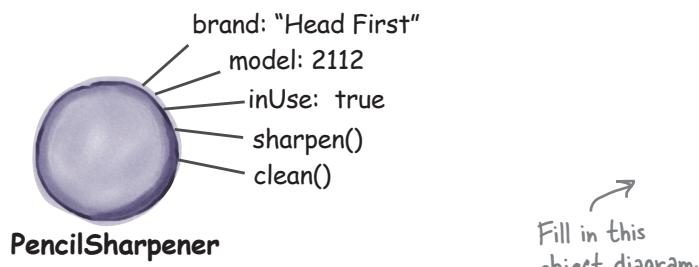
The object diagrams we've been using are cute and all that, but this is the *serious objects chapter*, so we're going to get more serious about our object diagrams. Actually, we really like the old ones, but the object diagrams in this chapter get complicated enough we just can't squeeze everything we need to into them.

So, without further ado, let us present the new format:



## Sharpen your pencil

Do a little practice just to make sure you've got the new format down. Take the object below and redo it in the new and improved object diagram.



# Revisiting object constructors: we're reusing code, but are we being efficient?

Remember the Dog constructor we created in the last chapter? Let's take another quick look and review what we're getting out of using the constructor:

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.bark = function() {
        if (this.weight > 25) {
            alert(this.name + " says Woof!");
        } else {
            alert(this.name + " says Yip!");
        }
    };
}
```

Every dog can have its own custom values and a consistent set of properties.

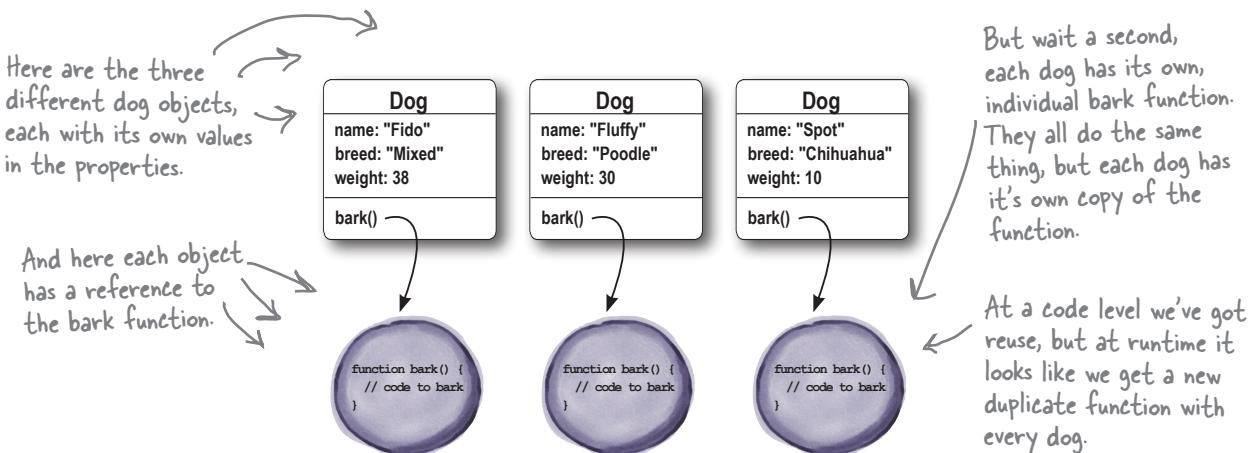
And every dog comes complete with a bark method.

Even better, we're totally reusing code across all the dogs.

So by using the constructor we get a nice, consistent dog object that we can customize to our liking, and, we also can leverage the methods that are defined in it (in this case there's only one, `bark`). Further, every dog gets the same code from the constructor, saving us lots of code headaches if things change in the future. That's all great, but let's look at what happens at runtime when we evaluate the code below:

```
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
```

This code causes three dog objects to be created. Let's use our new object diagrams to see what that looks like:

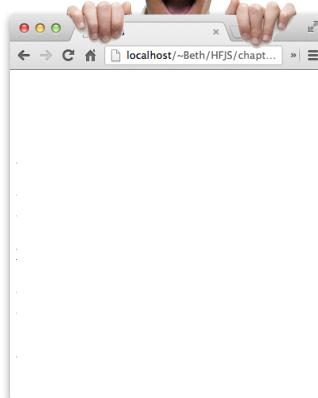


I've cornered the dog market and it's all thanks to your Dog constructor. Check it out...



Hey! You up there!

The browser.



You're killing us down here with all those extra methods you're creating. We're just about to run out of memory and then it's game over!

If this was a mobile device we'd be dead already.

Personally, I think every dog should have her very own bark method. Just sayin'.



## Is duplicating methods really a problem?

Actually, it is. In general we don't want a new set of methods being created every time you instantiate an object with a constructor. Doing so hurts the performance of your application and impacts resources on your computer, which can be a big deal, particularly on mobile devices. And, as you're going to see, there are more flexible and powerful ways to craft your JavaScript objects.

Let's take a step back and think about one of the main reasons we used constructors in the first place: we were trying to *reuse behavior*. For instance, remember that we had a bunch of dog objects and we wanted all those objects to use the same `bark` method. By using a constructor we achieved this at a code level by placing the `bark` method in one place—inside the `Dog` constructor—and so we reused the same `bark` code each time we instantiated an object. But, our solution doesn't look as promising at runtime because every dog instance is getting its own copy of the `bark` method.

Typically when we talk about an object's "behavior" we're referring to the set of methods it supports.

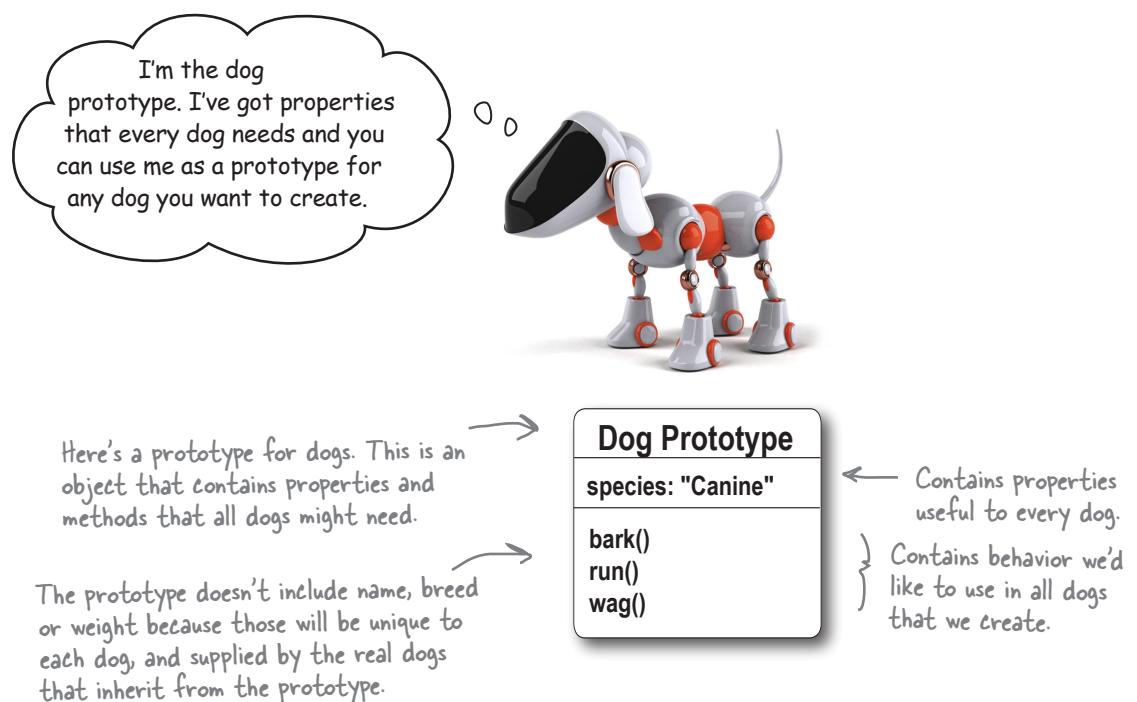
Now the reason we're running into this problem is because we aren't taking full advantage of JavaScript's object model, which is based on the idea of *prototypes*. In this model, we can create objects that are extensions of other objects—that is, of prototype objects.

To demonstrate prototypes, hmm... if only we had a *dog prototype* around that we could work from...

# What are prototypes?

JavaScript objects can inherit properties and behavior from other objects. More specifically, JavaScript uses what is known as *prototypal inheritance*, and the object you're inheriting behavior from is called the *prototype*. The whole point of this scheme is to inherit and reuse existing properties (including methods), while extending those properties in your brand new object. That's all quite abstract so let's work through an example.

We'll start with a prototype for a dog object. Here's what it might look like:

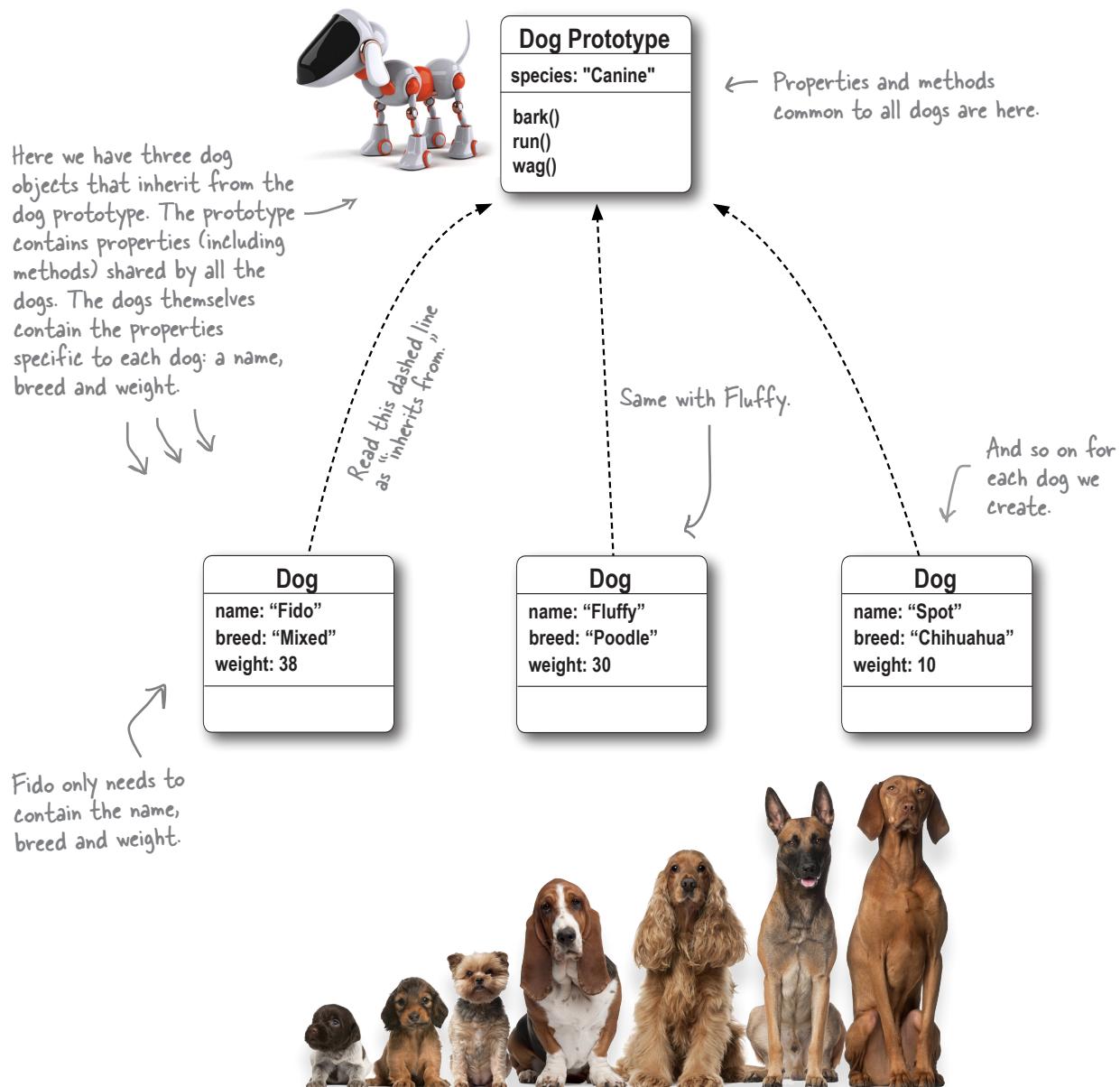


So now that we have a good dog prototype, we can create dog objects that inherit properties from that prototype. Our dog objects will also extend the prototype properties with dog-specific properties or behaviors. For example, we know we'll be adding a name, breed and weight to each dog.

You'll see that if any of these dogs needs to bark, run or wag their tails, they can rely on the prototype for those behaviors, because they inherit them from the prototype. So, let's create a few dog objects so you can see how this all works.

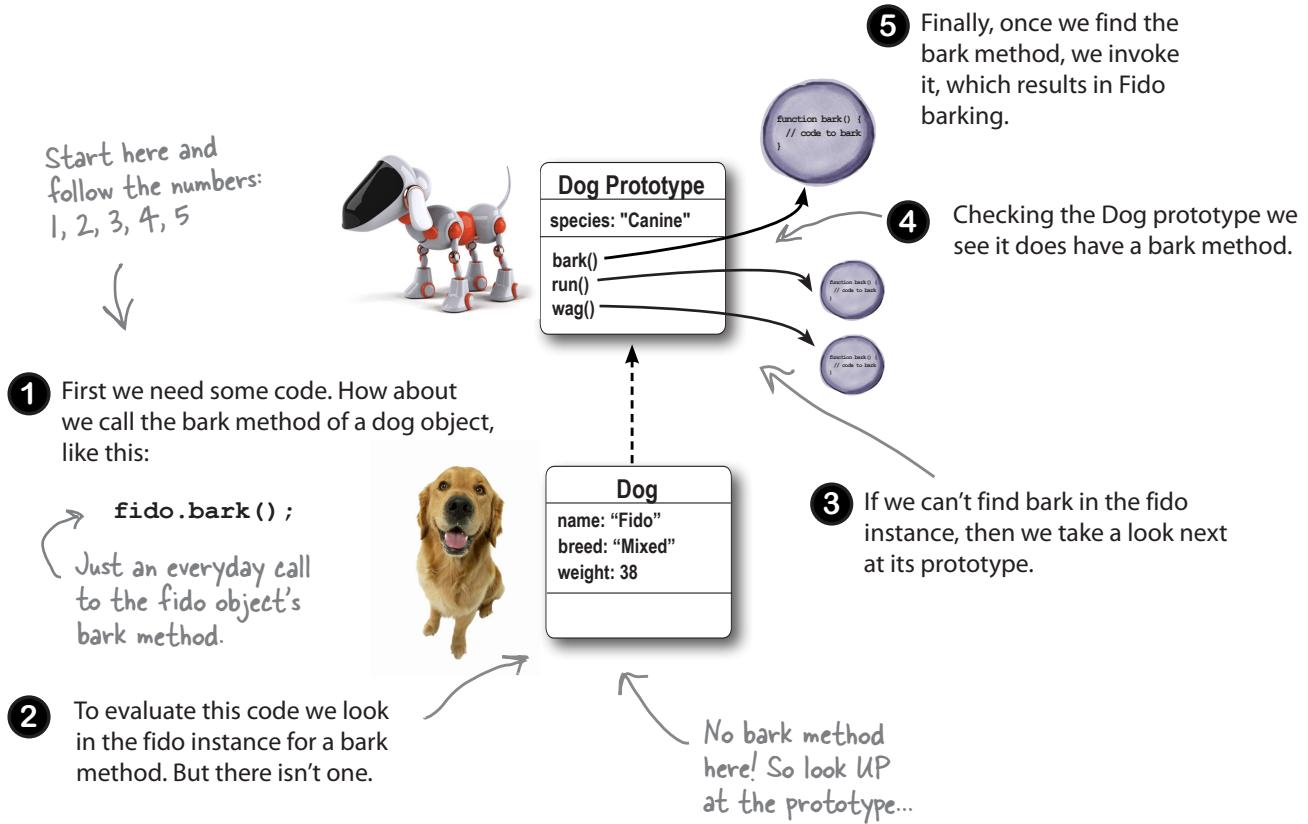
# Inheriting from a prototype

First, we need to create object diagrams for the Fido, Fluffy and Spot dog objects and have them inherit from the new dog prototype. We'll show inheritance by drawing a set of dashed lines from the dog instances to the prototype. And remember, we put only the methods and properties that are common to *all* dogs in the dog prototype, because *all* the dogs will inherit them. All the properties specific to an actual dog, like the dog's name, go into the dog instances, because they are different for each dog.



# How inheritance works

How do we make dogs bark if the bark method isn't in the individual dog instances, but rather is in the prototype? That's where inheritance comes in. When you call a method on an object instance, and that method isn't found in the instance, you check the prototype for that method. Here's how.

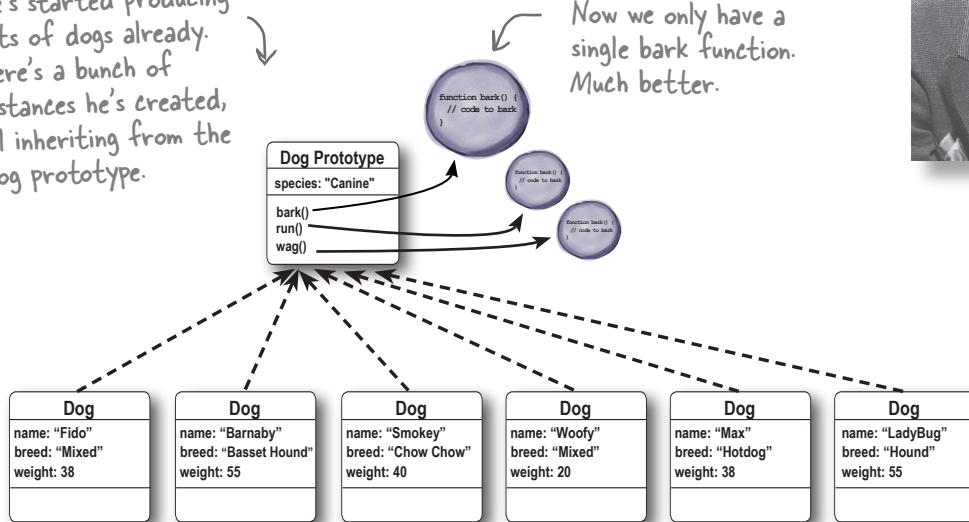


Properties work the same way. If we write code that needs `fido.name`, the value will come from the `fido` object. But if we want the value of `fido.species`, we first check the `fido` object, but when it isn't found there, we check the `Dog` prototype (and find it).

So now that you have this newfangled inheritance thing, I can fire up the dog factory again?



He's started producing lots of dogs already. Here's a bunch of instances he's created, all inheriting from the dog prototype.



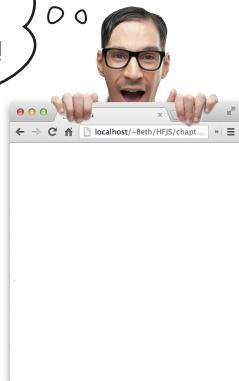
Now we only have a single bark function. Much better.

Every dog has been customized with its name, breed and weight, but relies on the prototype for the species property and the bark method.

Now that you understand how to use inheritance we can create a large number of dogs. All the dogs can all still bark, but now they're relying on the dog prototype object to supply that bark method. We have code reuse, not just by having our code written in one place, but by having all dog instances use the same bark method at runtime, which means we aren't causing lots of runtime overhead.

You're going to see that by using prototypes, you'll be able to quickly assemble objects that reuse code, and that can be extended with new behavior and properties.

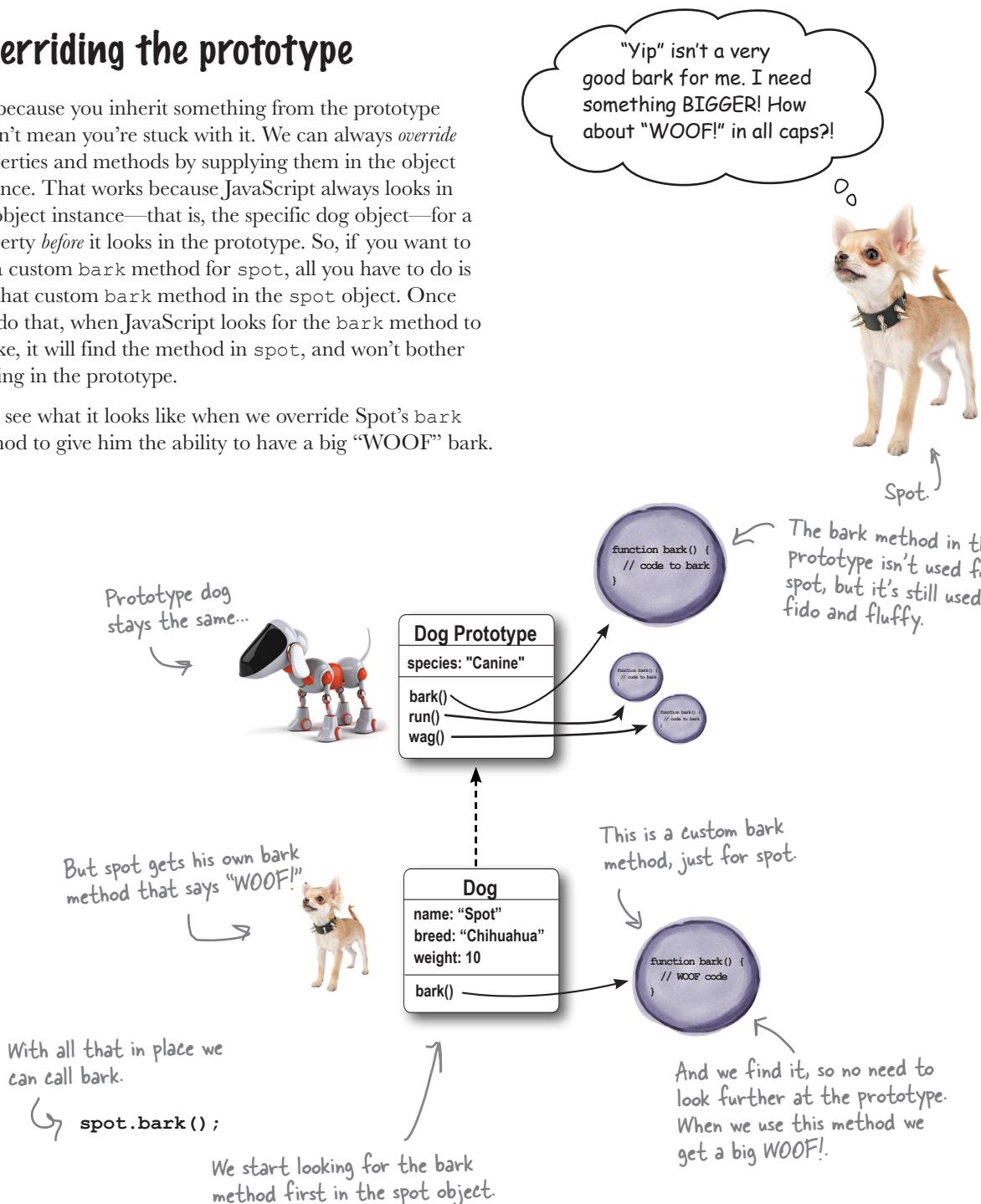
Thank you! Before you started using inheritance we were dying down here!



# Overriding the prototype

Just because you inherit something from the prototype doesn't mean you're stuck with it. We can always *override* properties and methods by supplying them in the object instance. That works because JavaScript always looks in the object instance—that is, the specific dog object—for a property *before* it looks in the prototype. So, if you want to use a custom bark method for spot, all you have to do is put that custom bark method in the spot object. Once you do that, when JavaScript looks for the bark method to invoke, it will find the method in spot, and won't bother looking in the prototype.

Let's see what it looks like when we override Spot's bark method to give him the ability to have a big "WOOF" bark.





## Code Magnets

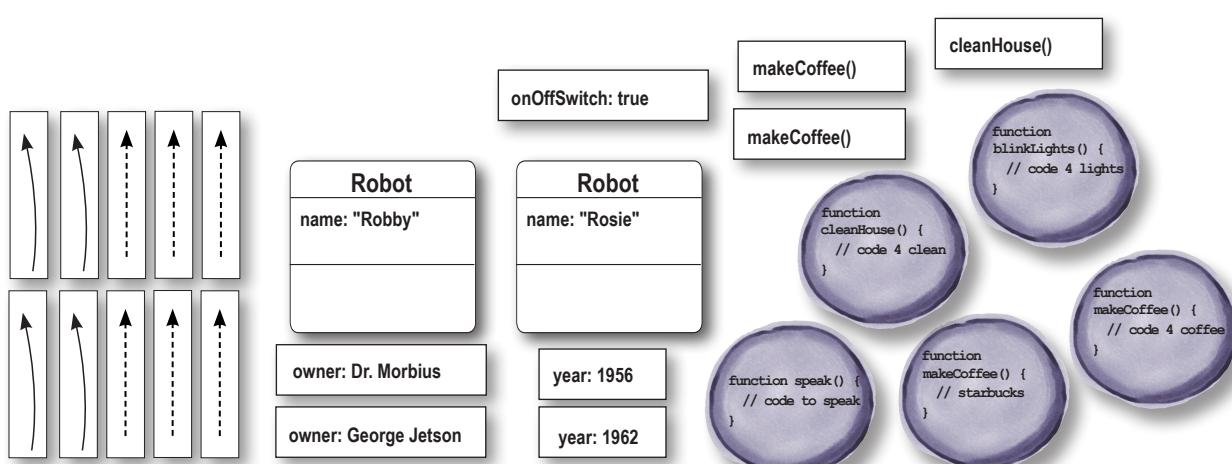
We had an object diagram on the fridge, and then someone came and messed it up. Can you help put it back together? To reassemble it, we need two instances of the robot prototype. One is Robby, created in 1956, owned by Dr. Morbius, has an on/off switch and runs to Starbucks for coffee. We've also got Rosie, created in 1962, who cleans house and is owned by George Jetson. Good luck (oh, and there might be some extra magnets below)!

Here's the  
prototype your  
robots can  
inherit from.



Robot Prototype
maker: "ObjectsRUs"
speak()
makeCoffee()
blinkLights()

Build the object diagram here.



# So where do you get a prototype?

We've talked a lot about the dog prototype, and at this point, you're probably ready to see an example that uses code rather than diagrams. So, how do we create or get a hold of a dog prototype? Well, it turns out, you've actually had one all along. You just didn't know it.

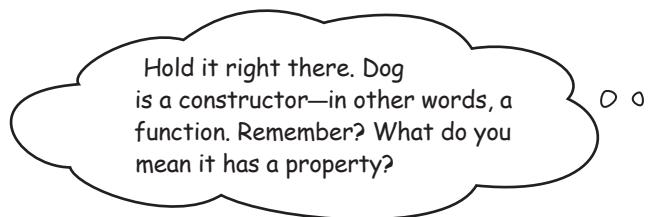
And here's how you access it in code:

`Dog.prototype`



If you look at your `Dog` constructor, it has a `prototype` property that holds a reference to the actual prototype.

Now, if you take this `prototype` property...



## Don't look at the man behind the curtain!

Just kidding; you're right. We were trying to gloss over that point (and we really still intend to, for now). Here's the short story: functions are objects in JavaScript. In fact, in JavaScript just about everything is an object underneath, even arrays if you haven't figured that one out yet.

But, for now, we don't want to get sidetracked on this. Just know that functions, in addition to doing everything you already know they can do, can also have properties, and in this case, the constructor always has a `prototype` property. More on functions and other things that are objects later, we promise.

## How to set up the prototype

As we were saying, you can access the prototype object through the Dog constructor's `prototype` property. But what properties and methods are in the prototype object? Well, until you set it up yourself, not much. In other words it's your job to add properties and methods to the prototype. We typically do that before we start using the constructor.

So, let's set up the dog prototype. First we need a constructor to work from, so let's look at our object diagram to see how to make that:

```
function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
}
```



But we're going to get our methods from the prototype, so we don't need them in the constructor.

This is the constructor to create an instance of a dog. Each instance has its own name, breed and weight, so let's incorporate those into the constructor.

Dog
name: "Spot"
breed: "Chihuahua"
weight: 10

Okay, now that we have a constructor, let's set up our dog prototype. We want it to have the `species` property and the `bark`, `run` and `wag` methods. Here's how we do that:

```
Dog.prototype.species = "Canine";  
  
Dog.prototype.bark = function() {  
  if (this.weight > 25) {  
    console.log(this.name + " says Woof!");  
  } else {  
    console.log(this.name + " says Yip!");  
  }  
};  
  
Dog.prototype.run = function() {  
  console.log("Run!");  
};  
  
Dog.prototype.wag = function() {  
  console.log("Wag!");  
};
```

We assign the string "Canine" to the prototype's `species` property.

And for each method, we assign the appropriate function to the prototype's `bark`, `run` and `wag` properties respectively.



### Serious Coding

Don't forget about chaining:

`Dog.prototype.species`

Start with Dog and grab its `prototype` property, which is a reference to an object that has a `species` property.

# Test drive the prototype with some dogs



Go ahead and get this code typed into a file ("dog.html") and loaded into your browser for testing. We've reproduced all the code after the changes we made on the previous page, and added a bit of testing code. Make sure all your dogs bark, run and wag like they should.

```
function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
}

Dog.prototype.species = "Canine";
```

Here's the Dog constructor.

```
Dog.prototype.bark = function() {
    if (this.weight > 25) {
        console.log(this.name + " says Woof!");
    } else {
        console.log(this.name + " says Yip!");
    }
};
```

And here's where we add properties and methods to the dog prototype.

```
Dog.prototype.run = function() {
    console.log("Run!");
};
```

We're adding one property and three methods to the prototype.

```
Dog.prototype.wag = function() {
    console.log("Wag!");
};
```

Now, we create the dogs like normal...

```
var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
```

But wait a second, didn't Spot want his bark to be WOOF!?

```
fido.bark();
fido.run();
fido.wag();
```

... and then we call the methods for each dog, just like normal. Each dog inherits the methods from the prototype.

```
fluffy.bark();
fluffy.run();
fluffy.wag();

spot.bark();
spot.run();
spot.wag();
```

Each dog is barking, running and wagging. Good.

JavaScript console

```
Fido says Woof!
Run!
Wag!
Fluffy says Woof!
Run!
Wag!
Spot says Yip!
Run!
Wag!
```





Hey, don't forget about me. I requested a bigger WOOF!

## Give Spot his WOOF! in code

Don't worry, we didn't forget about Spot. Spot requested a bigger WOOF! so we need to override the prototype to give him his own custom bark method. Let's update the code:

... } ← The rest of the code goes here. We're just saving trees, or bits, or our carbon footprint, or something...

```
var spot = new Dog("Spot", "Chihuahua", 10);  
  
spot.bark = function() {  
    console.log(this.name + " says WOOF!");  
};  
  
// calls to fido and fluffy are the same
```

spot.bark(); ← We don't need to change how we  
spot.run(); call Spot's bark method at all.  
spot.wag();

← The only change we make to the code is to give Spot his own custom bark method.

### Test drive the custom bark method



Add the new code above and take it for a quick test drive...

Spot gets the WOOF! he wanted. →

JavaScript console

```
Fido says Woof!  
Run!  
Wag!  
Fluffy says Woof!  
Run!  
Wag!  
Spot says WOOF!  
Run!  
Wag!
```



## Exercise

Remember our object diagram for the Robby and Rosie robots? We're going to implement that now. We've already written a Robot constructor for you along with some test code. Your job is to set up the robot prototype and to implement the two robots. Make sure you run them through the test code.

```
function Robot(name, year, owner) {
  this.name = name;
  this.year = year;
  this.owner = owner;
}
```

Here's the basic Robot constructor. You still need to set up its prototype.

```
Robot.prototype.maker =
```

You'll want to set up the robot prototype here.

```
Robot.prototype.speak =
```

```
Robot.prototype.makeCoffee =
```

```
Robot.prototype.blinkLights =
```

```
var robby =
```

```
var rosie =
```

```
robby.onOffSwitch =
```

```
robby.makeCoffee =
```

Write your code to create the Robby and Rosie robots here. Make sure you add any custom properties they have to the instances.

```
rosie.cleanHouse =
```

Use this code to test your instances to make sure they are working properly and inheriting from the prototype.

```
console.log(robby.name + " was made by " + robby.maker +
           " in " + robby.year + " and is owned by " + robby.owner);
robby.makeCoffee();
robby.blinkLights();
```

```
console.log(rosie.name + " was made by " + rosie.maker +
           " in " + rosie.year + " and is owned by " + rosie.owner);
rosie.cleanHouse();
```

I was wondering how `this.name` in the `bark` method still works given that the `bark` method is in the prototype and not in the original object.

**Good question.** When we didn't have prototypes this was easy because we know `this` gets set to the object whose method was called. When we are calling the `bark` method in the prototype, you might think that `this` is now set to the prototype object. Well, that's not how it works.

When you call an object's method, `this` is set to the object whose method was called. If the method is not found in that object, and is found in the prototype, that doesn't change the value of `this`. `this` always refers to the original object—that is, the object whose method was called—even if the method is in the prototype. So, if we find the `bark` method in the prototype, then we call the method, with `this` set to the original dog object, giving us the result we want, like “Fluffy says Woof!”.



# Teaching <sup>all</sup><sub>a</sub> dog<sub>s</sub> a new trick

It's time to teach our dogs a new trick. That's right we said "dogs" plural, not dog. You see, now that we have a prototype, if we add any methods to that prototype, even after we've already created dog objects, all dogs inheriting from the prototype immediately and automatically get this new behavior.

Let's say we want to teach all our dogs to sit. What we do is add a method to the prototype for sitting.

```
var barnaby = new Dog("Barnaby", "Basset Hound", 55); ← Let's create another dog  
to test this on.  
  
Dog.prototype.sit = function() { ← And then let's add the  
    console.log(this.name + " is now sitting");  
}  
} → sit method.
```

We'll give this a try with Barnaby:

barnaby.sit(); ← We first check to see if the barnaby object has a sit method and there isn't one. So we then check the prototype, find the sit method, and invoke it.

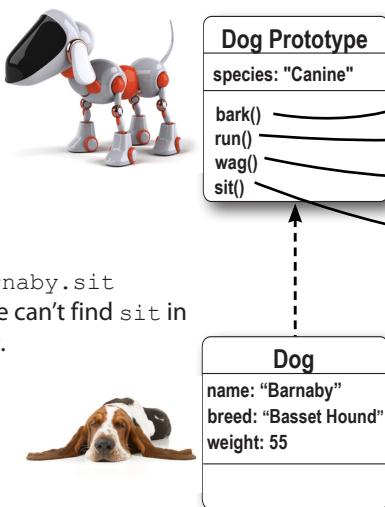
JavaScript console  
Barnaby is now sitting



## A Closer Look

Let's take a closer look at how this works. Make sure you follow the sequence 1, 2, 3, 4.

- 4 But we do find sit in the prototype, and we invoke it.



- 3 We call the barnaby.sit method, but we can't find sit in barnaby object.



- 2 Next we add a new method, sit, to the prototype.

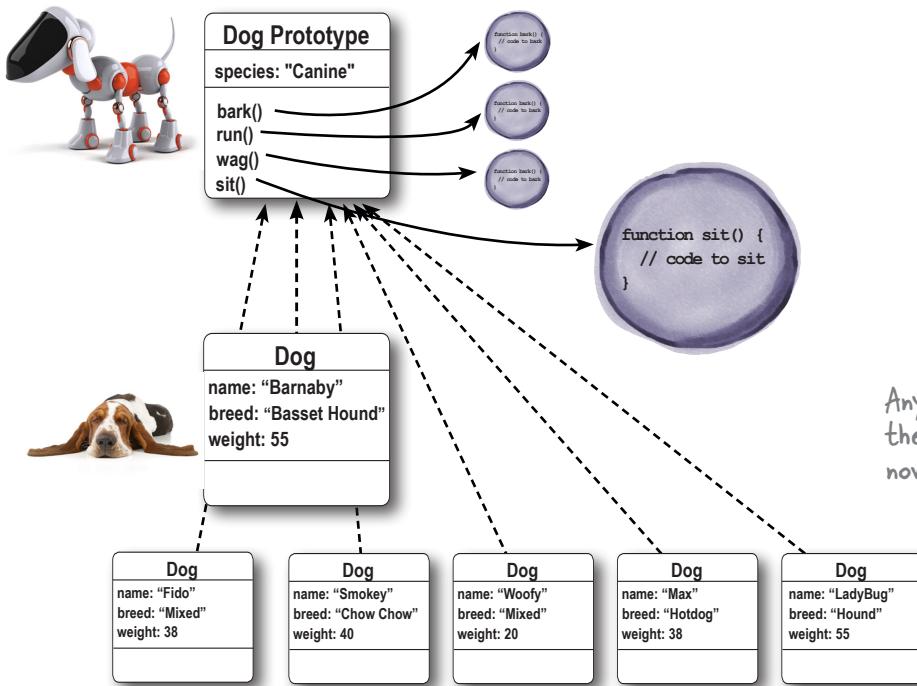


- 1 We create a new dog Barnaby.

## Prototypes are dynamic

We're glad to see Barnaby can now sit. But it turns out that now *all* our dogs can sit, because once you add a method to a prototype, any objects that inherit from that prototype can make use of that method:

This works for properties too, of course.



Any dog object that has the Dog prototype can now use the sit method.

there are no  
Dumb Questions

**Q:** So when I add a new method or property to a prototype, all the object instances that inherit from it immediately see it?

**A:** If by "see it" you mean that they inherit that method or property, you are correct. Notice that this gives you a way to extend or change the behavior of all your instances at runtime by simply changing their prototype.

**Q:** I see how adding a new property to a prototype makes that property available to all the objects that inherit from the prototype. What if I change an existing property in the prototype; does that affect those objects in the same way? Like if I change the property species to "Feline" instead of "Canine", does that mean all existing dogs are now "Feline" species?

**A:** Yes. If you change any property in the prototype, it affects all the objects that inherit from that prototype, unless that object has overridden that property.



Robby and Rosie are being used in a Robot game. You'll find the code for them below. In this game, whenever a player reaches level 42, a new robot capability is unlocked: the laser beam capability. Finish the code below so that at level 42 both Robby and Rosie get their laser beams. Check your answer at the end of the chapter before you go on.

```
function Game() {
    this.level = 0;
}

Game.prototype.play = function() {
    // player plays game here
    this.level++;
    console.log("Welcome to level " + this.level);
    this.unlock();
}

Game.prototype.unlock = function() {

}

function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

var game = new Game();
var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

while (game.level < 42) {
    game.play();
}

robby.deployLaser();
rosie.deployLaser();
```

```
JavaScript console
Welcome to level 1
Welcome to level 2
Welcome to level 3
...
Welcome to level 41
Welcome to level 42
Rosie is blasting you with
laser beams.
```

↗ A sample of our output. When you finish your code, give it a play and see which robot wins and gets to blast its laser beams!



## A more interesting implementation of the sit method

Let's make the `sit` method a little more interesting: dogs will start in a state of not sitting (in other words, standing up). So, when `sit` is called, if a dog isn't sitting, we'll make him sit. Otherwise, we'll let the user know he's already sitting. To do this we're going to need an extra property, `sitting`, to keep track of whether the dog is sitting or not. Let's write the code:

We start with a `sitting` property in the prototype.

```
Dog.prototype.sitting = false;
```

```
Dog.prototype.sit = function() {
  if (this.sitting) {
    console.log(this.name + " is already sitting");
  } else {
    this.sitting = true;
    console.log(this.name + " is now sitting");
  }
};
```

Notice that the instance now has its own local `sitting` property, set to `true`.

By setting `sitting` to `false` in the prototype, all dogs start by not sitting.

Then, in the `sit` method, we check to see if the dog is sitting or not. At first, when we check `this.sitting` we'll be looking at the value in the dog prototype.

If the dog is sitting, we say he's already sitting.

But, if the dog is not sitting, we say he's now sitting and then we set the value of `this.sitting` to `true`. This overrides the prototype property and sets the value in the instance.

The interesting thing about this code is that when a dog instance starts out life, it inherits a default value of `false` for `sitting`. But, as soon as the `sit` method is called, the dog instance adds its own value for `sitting`, which results in a property being created in the instance. This overrides the inherited `sitting` property in the prototype. This gives us a way to have a default for all dogs, and then to specialize each dog if we need to.

### Test drive the new `sit` method

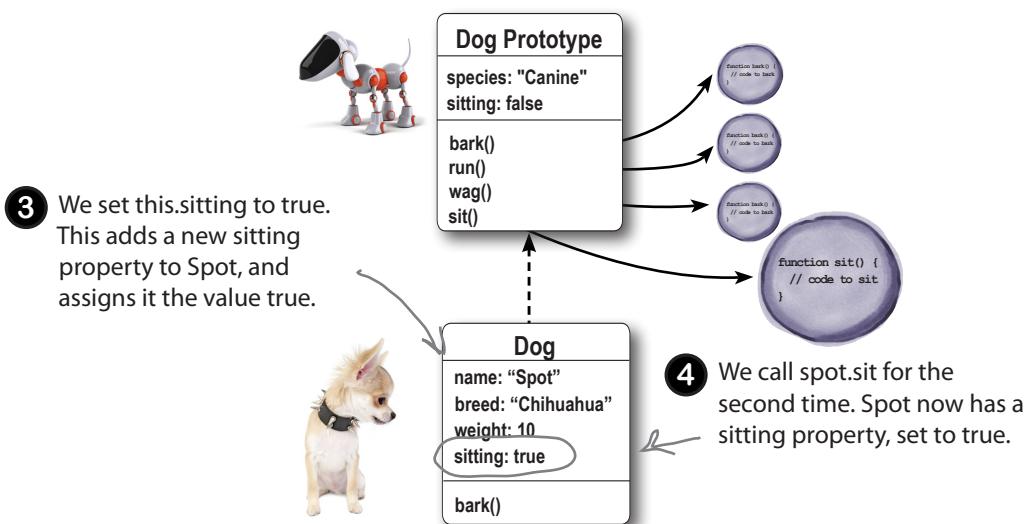
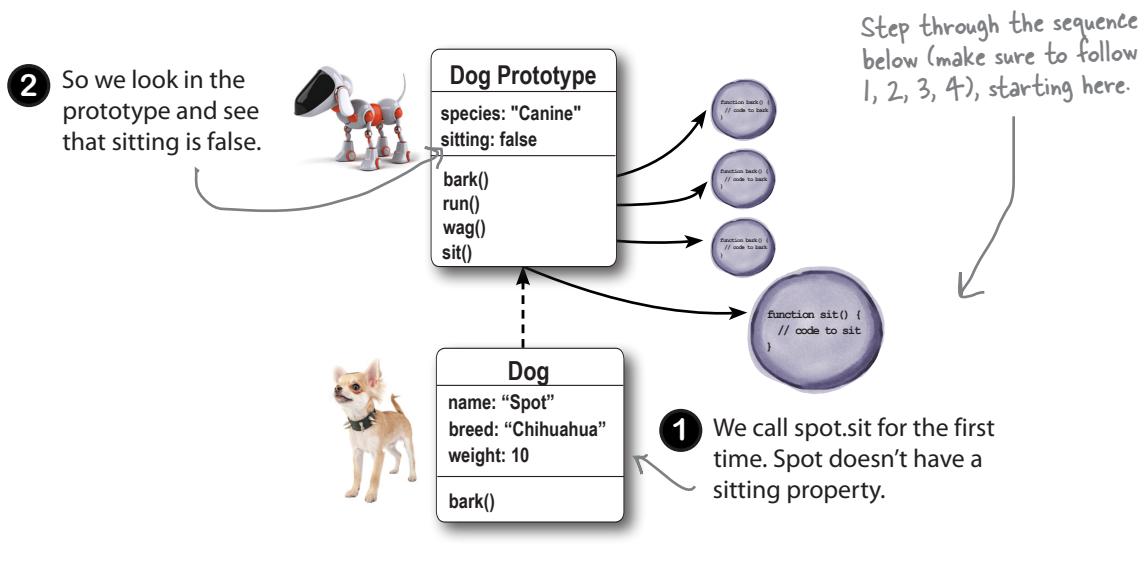
Let's give this a try for real. Go ahead and update your code, adding the new property and implementation of `sit`. Now when we test drive this code, you can see that we can make `barnaby` sit, and then make `spot` sit, and each dog keeps track of whether it is sitting separately:



```
JavaScript console
barnaby.sit() → Barnaby is now sitting
barnaby.sit() → Barnaby is already sitting
spot.sit() → Spot is now sitting
spot.sit() → Spot is already sitting
```

# One more time: how the sitting property works

Let's make sure we've got this down, because if you go too fast on this implementation you might miss the key details. Here's the key: the first time we get the value of `sitting`, we're getting it from the prototype. But then when we set `sitting` to true, that happens in the object instance, not the prototype. And after that property has been added to the object instance, every subsequent time we get the value of `sitting`, we're getting it from the object instance because it is overriding the value in the prototype. Let's step through it one more time:





While we're talking about properties, is there a way in my code to determine if I'm using a property that's in the instance or in the prototype?

**Yes, there is.** You can use the `hasOwnProperty` method that every object has. The `hasOwnProperty` method returns true if a property is defined in an object instance. If it's not, but you can access that property, then you can assume the property must be defined in the object's prototype.

Let's try it on `fido` and `spot`. First, we know that the `species` property is implemented only in the dog prototype, and neither `spot` nor `fido` has overridden this property. So if we call the `hasOwnProperty` method and pass in the property name, "species", as a string, we get back false for both:

```
spot.hasOwnProperty("species");
fido.hasOwnProperty("species");
```

Both of these return the value false because species is defined in the prototype, not the object instances spot and fido.

Now let's try it for the `sitting` property. We know that the `sitting` property is defined in the prototype and initialized to false. So we assign the value true to `spot.sitting`, which overrides the `sitting` property in the prototype and defines `sitting` in the `spot` instance. Then we'll ask both `spot` and `fido` if they have their own `sitting` property defined:

```
When we first check to see if Spot has
his own sitting property we get false.

spot.hasOwnProperty("sitting");
spot.sitting = true; ← Then we set spot.sitting to
                      true, adding this property to
                      the spot instance.

spot.hasOwnProperty("sitting");
fido.hasOwnProperty("sitting"); ← This call to hasOwnProperty
                                returns true, because spot now
                                has his own sitting property.
```

But this call to `hasOwnProperty` returns false, because the `fido` instance does not have a `sitting` property. That means the `sitting` property that `fido` uses is defined only in the prototype, and inherited by `fido`.



We've added a new capability to our robots, Robby and Rosie: they can now report when they have an error through the `reportError` method. Trace the code below, paying particular attention to where this method gets its error information, and to whether it's coming from the prototype or the robot instance.

Below give the output of this code:

```
function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

Robot.prototype.maker = "ObjectsRUs";
Robot.prototype.errorMessage = "All systems go.";
Robot.prototype.reportError = function() {
    console.log(this.name + " says " + this.errorMessage);
};

Robot.prototype.spillWater = function() {
    this.errorMessage = "I appear to have a short circuit!";
};

var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

rosie.reportError();
robby.reportError();
robby.spillWater();
rosie.reportError();
robby.reportError();

console.log(robby.hasOwnProperty("errorMessage")); _____
console.log(rosie.hasOwnProperty("errorMessage")); _____
```

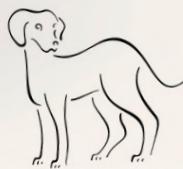
Does Robby have his own  
errorMessage property?

Does Rosie? ↗

# BEST DOG IN SHOW

All your hard work in this chapter has already paid off. The Webville Kennel Club saw your work on the dog objects and they immediately knew they'd found the right person to implement their dog show simulator. The only thing is they need you to update the Dog constructor to make show dogs. After all, show dogs aren't ordinary dogs—they don't just run, they gait. They don't go through the trash, they show a tendency towards scent articles; they don't beg for treats, they show a desire for bait.

More specifically, here's what they're looking for:



Wonderful work on the Dog constructor! We'd love to get you engaged on our dog show simulator. Show dogs are a little different, so they need additional methods (see below).

Thanks! —Webville Kennel Club

stack() — otherwise known as stand at attention.

gait() — this is like running. The method takes a string argument of "walk", "trot", "pace", or "gallop".

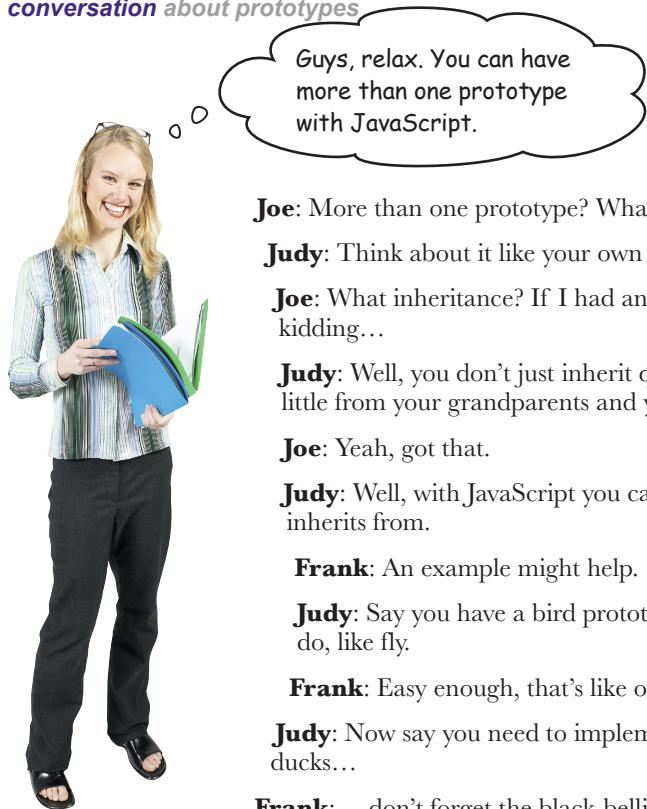
bait() — give the dog a treat.

groom() — doggie shampoo time.

## How to approach the design of the show dogs

So how are we going to design this? Clearly we'd like to make use of our existing dog code. After all, that's why Webville Kennel came to us in the first place. But how? Let's get some thoughts on the ways we could approach this:





**Joe:** More than one prototype? What does that even mean?

**Judy:** Think about it like your own inheritance.

**Joe:** What inheritance? If I had an inheritance I wouldn't be working here! Just kidding...

**Judy:** Well, you don't just inherit qualities from your parents, right? You inherit a little from your grandparents and your great-grandparents and so on.

**Joe:** Yeah, got that.

**Judy:** Well, with JavaScript you can set up a chain of prototypes that your object inherits from.

**Frank:** An example might help.

**Judy:** Say you have a bird prototype that knows how to do all things most birds do, like fly.

**Frank:** Easy enough, that's like our dog prototype.

**Judy:** Now say you need to implement a whole set of ducks—mallards, red-headed ducks...

**Frank:** ...don't forget the black-bellied-whistling duck.

**Judy:** Why, thank you Frank.

**Frank:** No problem. I was just reading about all those ducks in that *Head First Design Patterns* book.

**Judy:** Okay, but ducks are a different kind of bird. They swim, and we don't want to put that into the bird prototype. But with JavaScript we can create a duck prototype that inherits from the bird prototype.

**Joe:** So let me see if I have this right. We'd have a Duck constructor that points to a duck prototype. But that prototype—that is the duck prototype—would itself point to the bird prototype?

**Frank:** Whoa, shift back into first gear.

**Judy:** Think of it like this, Frank. Say you create a duck and you call its `fly` method. What happens if you look in the duck and there's no such method? You look in the duck prototype, still no fly method. So you look at the prototype the duck inherits from, bird, and you find `fly` there.

**Joe:** And, if we call `swim`, then we look in the duck instance, nothing there. We look in the duck prototype, and we find it.

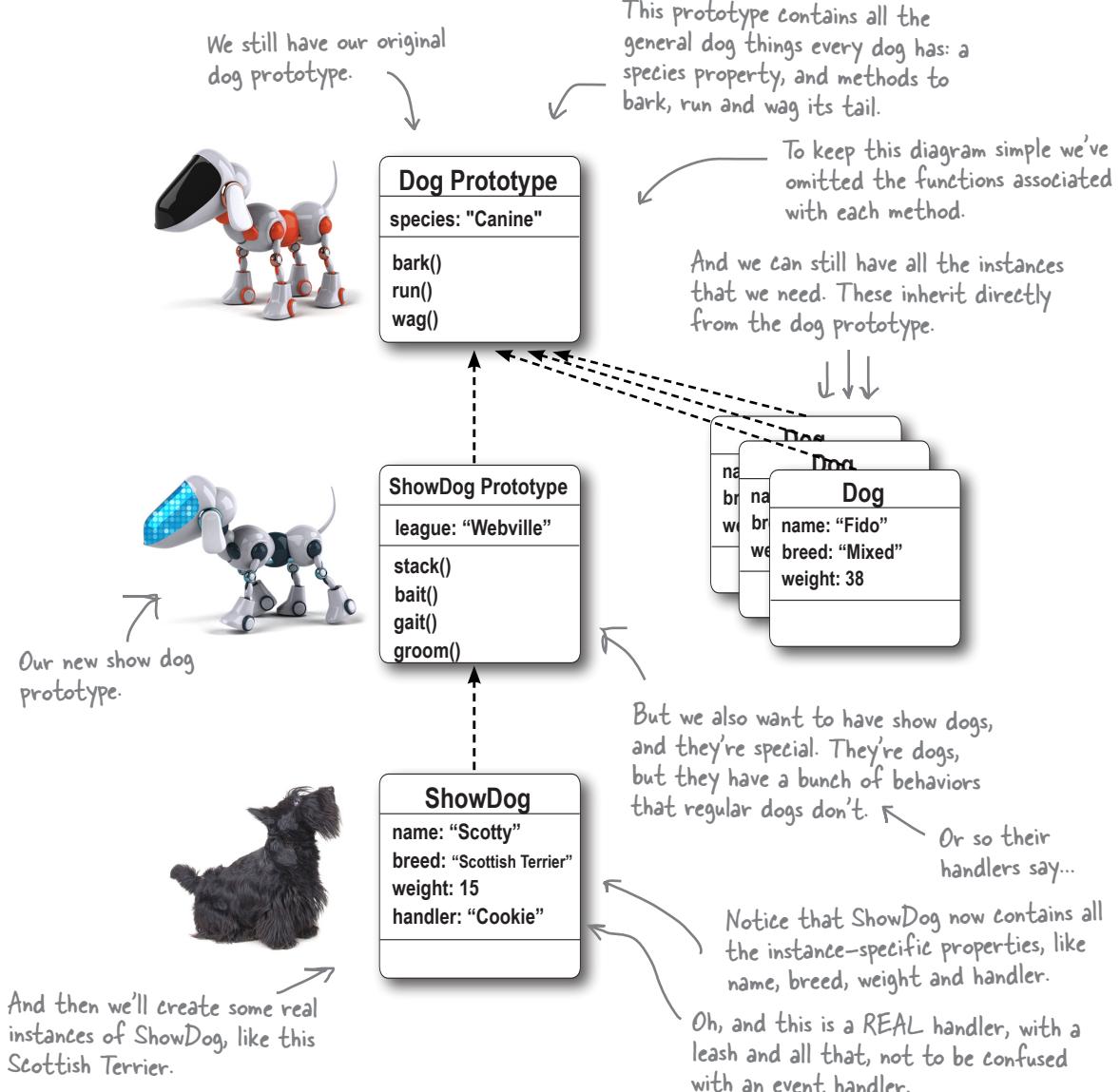
**Judy:** Right... so we're not just reusing the behavior of the duck prototype, we're following a chain up to the bird prototype, when necessary, to use that as well.

**Joe:** That sounds perfect for extending our dog prototype into a show dog. Let see what we can do with this.

# Setting up a chain of prototypes

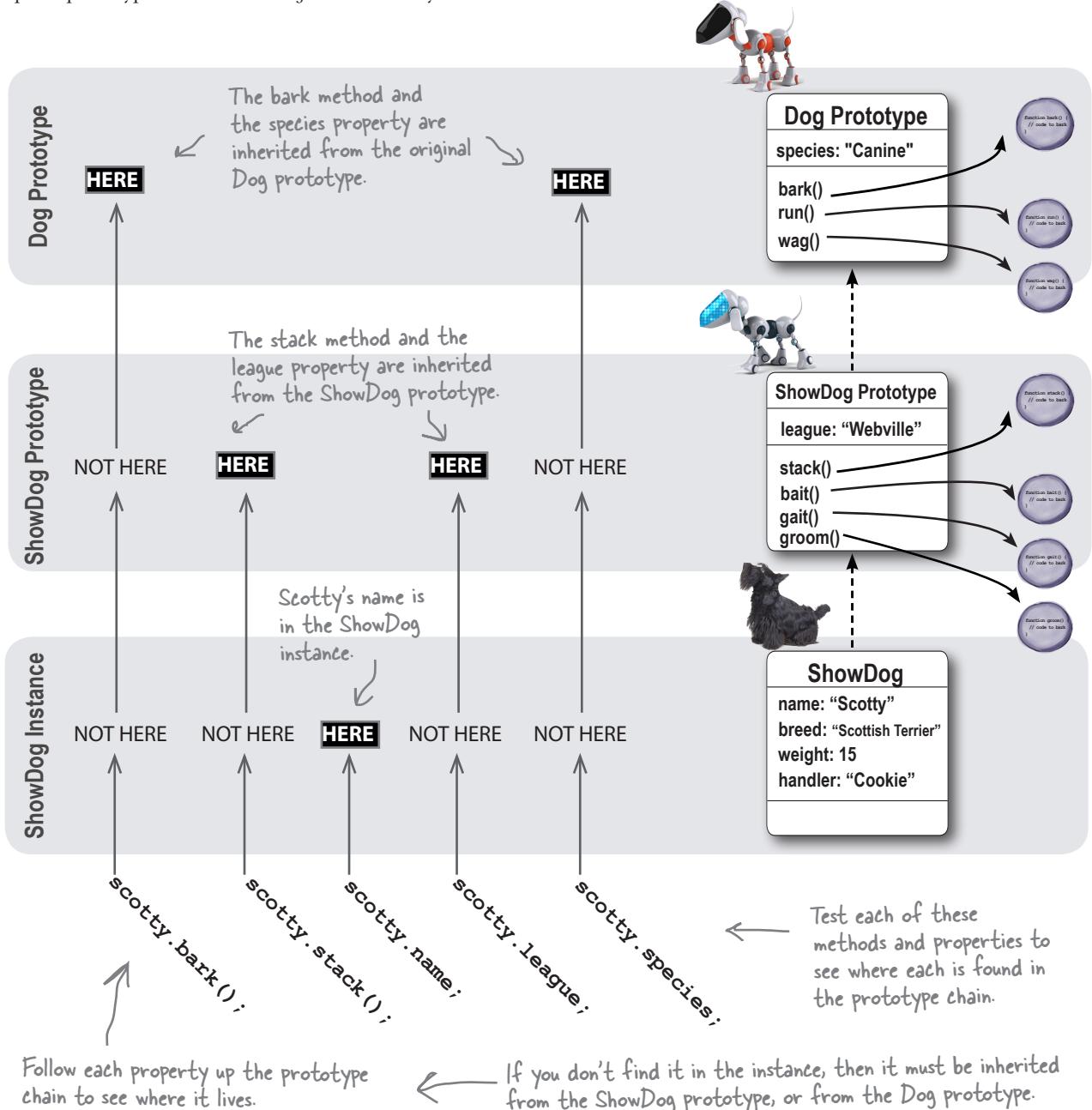
Let's start thinking in terms of a *chain of prototypes*. Rather than having an instance that inherits properties from just one prototype, there might be a chain of one or more prototypes your instance can inherit from. It's not that big a logical step from the way we've been thinking about this already.

Let's say we want a show dog prototype for our show dogs, and we want that prototype to rely on our original dog prototype for the bark, run, and wag methods. Let's set that up to get a feel for how it all works together:



# How inheritance works in a prototype chain

We've set up the prototype chain for the show dogs, so let's see how inheritance works in this context. Check out the properties and methods at the bottom of the page, and then trace them up the prototype chain to the object where they are defined.



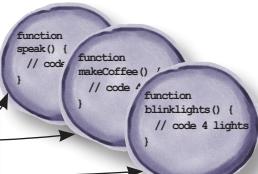
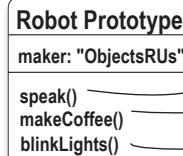


## Code Magnets

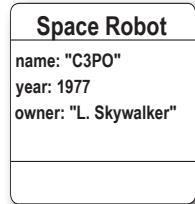
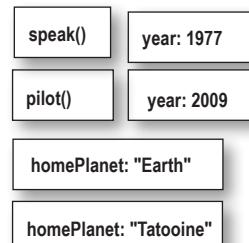
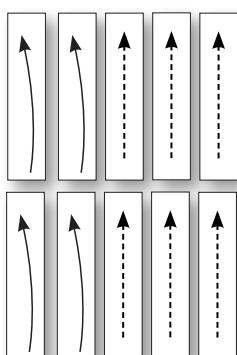
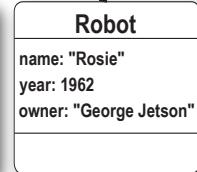
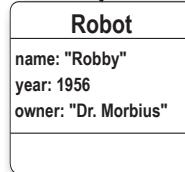
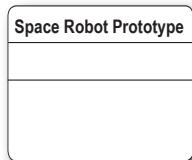
We had another object diagram on the fridge, and then someone came and messed it up. Again!! Can you help put it back together? To reassemble it we need a new line of Space Robots that inherit properties from Robots. These new Space Robots override the Robot's speaking functionality, and extend Robots with piloting functionality and a new property, homePlanet. Good luck (there might be some extra magnets below)!

Build the object diagram here.

Here's the prototype for Robots.



And here's the prototype for the Space Robots.



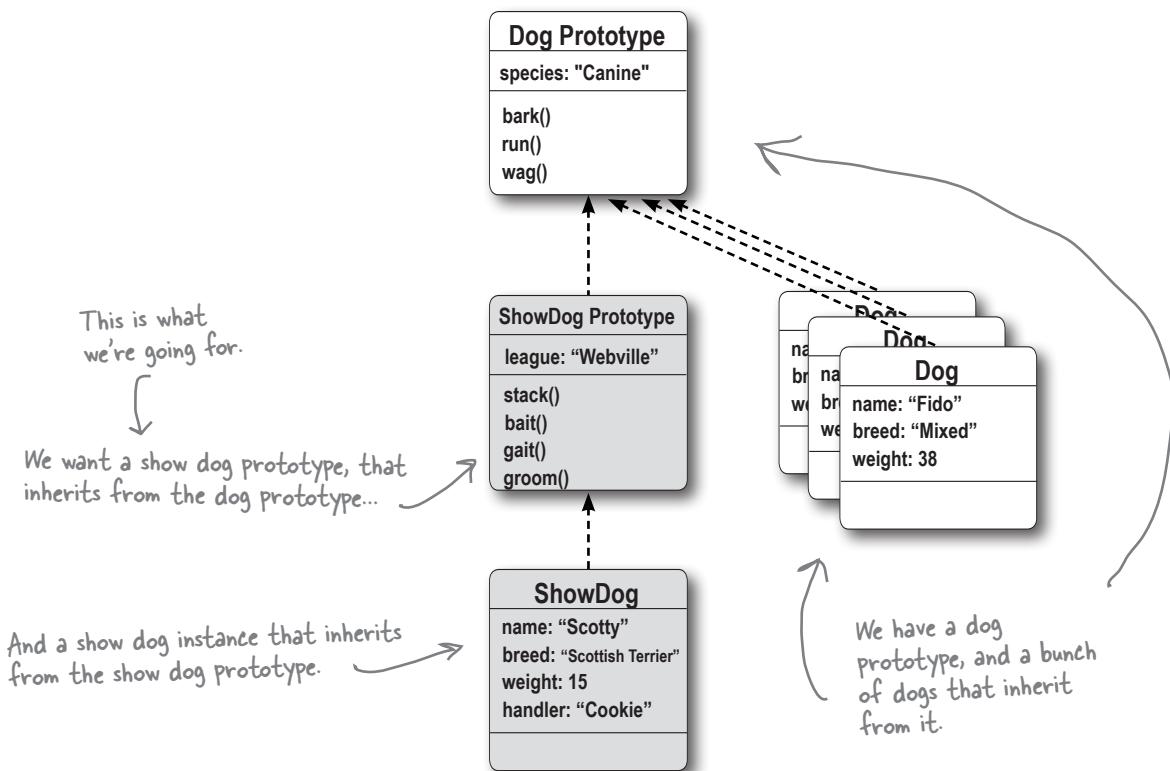
## Creating the show dog prototype

When we created the dog prototype we didn't have to do anything—there was already an empty object supplied by the Dog constructor's `prototype` property. So we took that and added the properties and methods we wanted our dog instances to inherit.

But with the show dog prototype we have more work to do because we need a prototype object that inherits from another prototype (the dog prototype). To do that we're going to have to create an object that inherits from the dog prototype and then explicitly wire things up ourselves.

Right now we have a dog prototype and a bunch of dog instances that inherit from that prototype. And what we want is a show dog prototype (that inherits from dog prototype), and a bunch of show dog instances that inherit from the show dog prototype.

Setting this up will take a few steps, so we'll take it one at a time.



# First, we need an object that inherits from the dog prototype

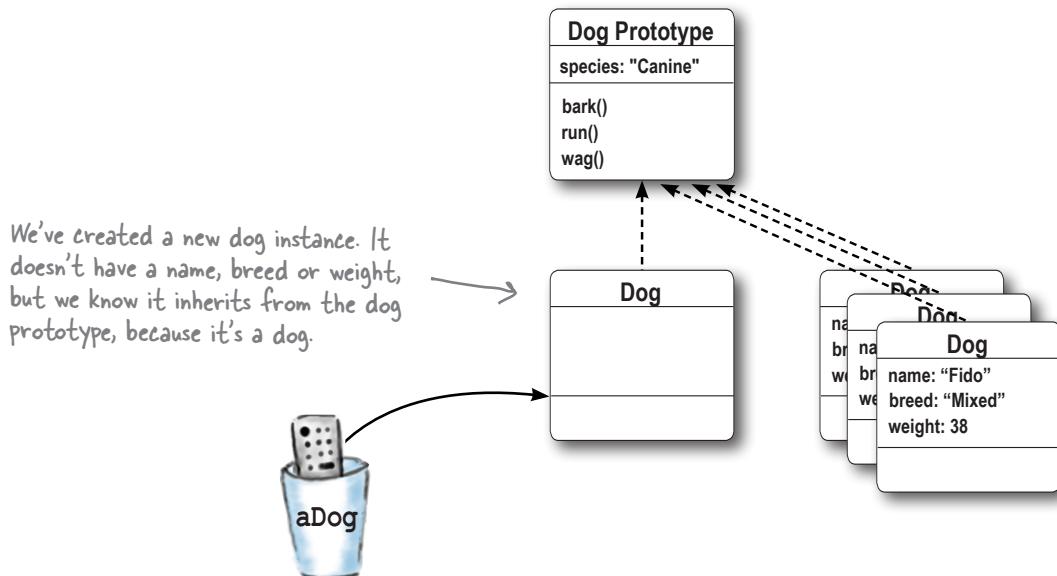
We've established that the show dog prototype is an object that inherits from the dog prototype. But, what's the best way to create an object that inherits from the dog prototype? Well, it's something you've already been doing as you've created instances of dog. Remember? Like this:

To create an object that inherits from the dog prototype, we just use new with the Dog constructor.

```
var aDog = new Dog();
```

→ We'll talk about what happened to the constructor arguments in a minute...

So this code creates an object that inherits from the dog prototype. We know this because it's exactly the same as how we created all our dog instances, except this time, we didn't supply any arguments to the constructor. That's because at the moment, we don't care about the specifics of the dog; we just need the dog to inherit from the dog prototype.



Now, what we really need is a show dog prototype. Like our dog instance, that's just an object that inherits from the dog prototype. So let's see how we can use our empty dog instance to make the show dog prototype we need.

## Next, turning our dog instance into a show dog prototype

Okay, so we have a dog instance, but how do we make that our show dog prototype object? We do this by assigning the dog instance to the `prototype` property of our `ShowDog` constructor. Oh wait; we don't have a `ShowDog` constructor yet... so let's make one:

```
function ShowDog(name, breed, weight, handler) {  
    this.name = name;  
    this.breed = breed;  
    this.weight = weight;  
    this.handler = handler;  
}
```

← This constructor takes everything we need to be a dog (name, breed, weight), and to be a show dog (a handler).

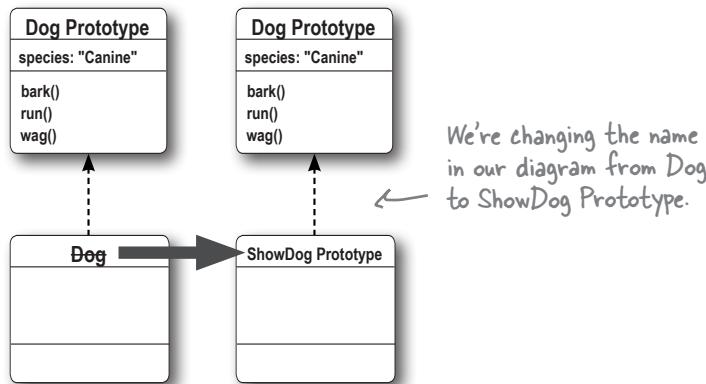
Now that we have a constructor, we can set its `prototype` property to a new dog instance:

```
ShowDog.prototype = new Dog();
```

← We could have used our dog instance created on the previous page, but we can skip the variable assignment and just assign the new dog straight to the `prototype` property instead.

So, let's think about where we are: we have a `ShowDog` constructor, with which we can make show dog instances, and we now have a show dog prototype, which is a dog instance.

Let's make sure our object diagram accurately reflects the roles these objects are playing by changing the label "Dog" to "ShowDog Prototype". But keep in mind, the show dog prototype *is still a dog instance*.



Now that we've got a `ShowDog` constructor and we've set up the show dog prototype object, we need to go back and fill in some details. We'll take a closer look at the constructor, and we've also got some properties and methods to add to the prototype so our show dogs have the additional show dog behavior we want them to have.

# Now it's time to fill in the prototype

We've got the show dog prototype set up (which at the moment is just an empty instance of dog). Now, it's time to fill it with properties and behaviors that will make it look more like a show dog prototype.

Here are some properties and methods that are specific to show dogs we can add:

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed; ←
    this.weight = weight;
    this.handler = handler;
}
```

Remember, the ShowDog constructor looks a lot like the Dog constructor. A show dog needs a name, breed, weight, plus one extra property, a handler (the person who handles the show dog). These will end up being defined in the show dog instance.

```
ShowDog.prototype = new Dog();
```

```
Showdog.prototype.league = "Webville"; ←

ShowDog.prototype.stack = function() {
    console.log("Stack");
};

ShowDog.prototype.bait = function() {
    console.log("Bait");
};

ShowDog.prototype.gait = function(kind) {
    console.log(kind + "ing");
};

ShowDog.prototype.groom = function() {
    console.log("Groom");
};
```

All our show dogs are in the Webville league, so we'll add this property to the prototype.

Here are all the methods we need for show dogs. We'll just keep them simple for now.

← We're adding all these properties to the show dog prototype so all show dogs inherit them.

This is where we're taking the dog instance that is acting as the show dog prototype, and we're adding new properties and methods.

Dog Prototype
species: "Canine"
bark()
run()
wag()

ShowDog Prototype
league: "Webville"
stack()
bait()
gait()
groom()

With these additions our show dog prototype is starting to look like a show dog. Let's update our object diagram again, and then it's probably time to do a big test run of the show dogs. We're guessing Webville Kennel is going to be pretty excited to see these in action.

→ We say that our show dog prototype "extends" the dog prototype. It inherits properties from the dog prototype and extends it with new ones.

## Creating a show dog instance

Now we just have one more thing to do: create an instance of ShowDog. This instance will inherit show dog properties and methods from our show dog prototype, and because our show dog prototype is an instance of Dog, the show dog will also inherit all its doggy behavior and properties from the dog prototype, so he'll be able to bark and run and wag with the rest of the dogs.

Here's all the code so far, and the code to create the instance:

```
function ShowDog(name, breed, weight, handler) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
    this.handler = handler;
}

ShowDog.prototype = new Dog();

ShowDog.prototype.league = "Webville";

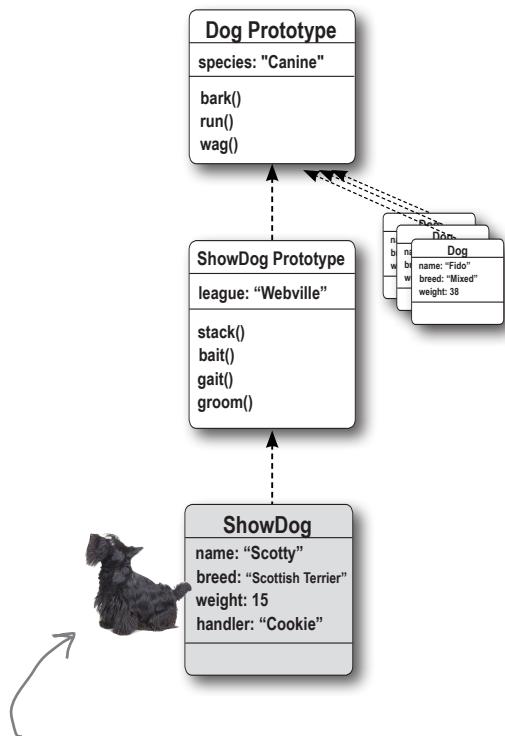
ShowDog.prototype.stack = function() {
    console.log("Stack");
};

ShowDog.prototype.bait = function() {
    console.log("Bait");
};

ShowDog.prototype.gait = function(kind) {
    console.log(kind + "ing");
};

ShowDog.prototype.groom = function() {
    console.log("Groom");
};

var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
```



And here's our show dog instance. It inherits from the show dog prototype, which inherits from the dog prototype. Just what we wanted. If you go back and look at page 592, you'll see we've completed the prototype chain.

Here's our new show dog, scotty.

# Test drive the show dog



Take all your the code on the previous page, and add to it the quality assurance code below, just to give scotty a good testing. Hey, and while you're at it, add a few dogs of your own and test them:

```
scotty.stack();
scotty.bark();
console.log(scotty.league);
console.log(scotty.species);
```

Here's what we got.

JavaScript console

```
Stack
Scotty says Yip!
Webville
Canine
```



## Exercise

Your turn. Add a SpaceRobot line of robots to the ObjectsRUs line of robots. These robots should of course be able to do everything that robots can do, plus some extra behavior for space robots. We've started the code below, so finish it up and then test it. Check your answer at the end of the chapter before moving on.

```
function SpaceRobot(name, year, owner, homePlanet) {
}

SpaceRobot.prototype = new _____;

_____.speak = function() {
  alert(this.name + " says Sir, If I may venture an opinion...");
};

_____.pilot = function() {
  alert(this.name + " says Thrusters? Are they important?");
};

var c3po = new SpaceRobot("C3PO", 1977, "Luke Skywalker", "Tatooine");
c3po.speak();
c3po.pilot();
console.log(c3po.name + " was made by " + c3po.maker);

var simon = new SpaceRobot("Simon", 2009, "Carla Diana", "Earth");
simon.makeCoffee();
simon.blinkLights();
simon.speak();
```



## Exercise

Let's take a closer look at all these dogs we're creating. We've tested Fido before and we know he's truly a dog. But let's see if he's a show dog as well (we don't think he should be). And what about Scotty? We figure he should be a show dog for sure, but is he a dog too? We're not sure. And we'll test Fido and Scotty's constructors while we're at it...

```
var fido = new Dog("Fido", "Mixed", 38);
if (fido instanceof Dog) {
    console.log("Fido is a Dog");
}

if (fido instanceof ShowDog) {
    console.log("Fido is a ShowDog");
}

var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
if (scotty instanceof Dog) {
    console.log("Scotty is a Dog");
}

if (scotty instanceof ShowDog) {
    console.log("Scotty is a ShowDog");
}

console.log("Fido constructor is " + fido.constructor);
console.log("Scotty constructor is " + scotty.constructor);
```

← Run this code and provide  
your output below.

Your output goes here:



JavaScript console

You'll find our output on  
the next page. →

# Examining the exercise results

Here's the output from that last test run:

Fido is a dog, which we expected, and we don't see that Fido is a show dog, so he must not be one. That makes sense too.

And Scotty is both a dog and a show dog, which makes sense. But how does `instanceof` know that?

Hmm, this looks weird. Both Fido and Scotty show they were created by the dog constructor. But we used the show dog constructor to create Scotty...

```
JavaScript console
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```



Let's think about these results for a minute. First, Fido is apparently just a dog and not a show dog—actually, that is totally what we thought would happen; after all, Fido was created with the Dog constructor, which has nothing to do with show dogs.

Next, Scotty is a dog *and* a show dog. That makes sense too, but how did this happen? Well, `instanceof` doesn't just look at what kind of object you are, it also takes into account all the objects you inherit from. So, Scotty was created as a show dog, but a show dog inherits from a dog, so Scotty is a dog too.

Next up, Fido has a Dog constructor, and that makes sense, because that is how we created him.

And finally, Scotty has a Dog constructor too. That doesn't make sense, because Scotty was created by the ShowDog constructor. What's going on here? Well, first let's think about where this constructor comes from: we're looking at the `scotty.constructor` property, and this is something we've never setup. So we must be inheriting it from the dog prototype (again, because we haven't explicitly set it up for a show dog).

So why is this happening? Honestly, it's a loose end that we need to cleanup. You see, if we don't take care of setting the show dog prototype's `constructor` property, no one else will. Now, keep in mind everything is working fine without it; but not setting it could lead to confusion if you or someone else tries to use `scotty.constructor` expecting to get show dog.

But don't worry, we'll fix it.

```
JavaScript console
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```

```
JavaScript console
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```

```
JavaScript console
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```

```
JavaScript console
Fido is a Dog
Scotty is a Dog
Scotty is a ShowDog
Fido constructor is function Dog...
Scotty constructor is function Dog...
```

## A final cleanup of show dogs

Our code is just about ready to ship to Webville Kennel, but we need to make one final pass to polish it. There are two small issues to clean up.

The first, we've already seen: that instances of ShowDog don't have their constructor property set correctly. They're inheriting the Dog constructor property. Now, just to be clear, all our code works fine as is, but setting the right constructor on our objects is a best practice, and some day another developer may end up with your code and be confused when they examine a show dog object.

To fix the constructor property, we need to make sure it is set up correctly in the show dog prototype. That way, when a show dog is constructed it will inherit the right constructor property. Here's how we do that:

```
function ShowDog(name, breed, weight, handler) {  
  this.name = name;  
  this.breed = breed;  
  this.weight = weight;  
  this.handler = handler;  
  
}  
  
ShowDog.prototype = new Dog();  
ShowDog.prototype.constructor = ShowDog;
```

That's all you need to do. When we check Scotty again he should have the correct constructor property, as should all other show dogs.

Here we're taking the show dog prototype and explicitly setting its constructor property to the ShowDog constructor.

Remember this is a best practice, without it your code still works as expected.

Note that we didn't have to do this for the dog prototype because it came with the constructor property set up correctly by default.



### Exercise

Here's what we got. Note that Scotty's constructor is now ShowDog.

Quickly rerun the tests from the previous exercise and make sure your Scotty show dog instance has the correct constructor.

#### JavaScript console

```
Fido is a Dog  
Scotty is a Dog  
Scotty is a ShowDog  
Fido constructor is function Dog...  
Scotty constructor is function ShowDog...
```

## A little more cleanup

There's another place we could use some cleanup: in the ShowDog constructor code. Let's look again at the constructor:

```
function ShowDog(name, breed, weight, handler) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
  this.handler = handler;
}
```

If you didn't notice, this code is replicated from the Dog constructor.

As you've seen in this book, anytime we see duplicated code, the warning bells go off. In this case, the Dog constructor already knows how to do this work, so why not let the constructor do it? Further, while our example has simple code, at times constructors can have complex code to compute initial values for properties, and we don't want to start reproducing code everytime we create a new constructor that inherits from another prototype. So let's fix this. We'll rewrite the code first, and step you through it:

```
function ShowDog(name, breed, weight, handler) {
  Dog.call(this, name, breed, weight);
  this.handler = handler;
}
```

This idea of eliminating duplicate code even has an acronym: DRY. "Don't Repeat Yourself" as all the cool coders say.

This bit of code is going to reuse the Dog constructor code to process the name, breed, and weight.

But we still need to handle the handler in this code because the Dog constructor doesn't know anything about it.

As you can see we've replaced the redundant code in the ShowDog constructor with a call to a method named `Dog.call`. Here's how it works: `call` is a built-in method that you can use on any function (and remember Dog is a function). `Dog.call` invokes the Dog function and passes it the object to use as `this`, along with all the arguments for the Dog function. Let's break this down:

Dog is the function we're going to call.

Whatever is in this is used for this in the body of the Dog function.

The rest of the arguments are just passed to Dog like normal.

With this code we're calling the Dog constructor function but telling it to use our ShowDog instance as this, and so the Dog function will set the name, breed and weight properties in our ShowDog object.

call is the method of Dog we're calling. The call method will cause the Dog function to be called. We use the call method instead of just calling Dog directly so we can control what the value of this is.

## Stepping through Dog.call

Using `Dog.call` to call `Dog` is a bit tricky to wrap your head around so we'll walk through it again, starting with the reworked code.

```
function ShowDog(name, breed, weight, handler) {
  Dog.call(this, name, breed, weight); ←
  this.handler = handler; ←
}

But Dog doesn't know anything about
handler, so we have to take care of
that in ShowDog.
```

We're going to rely on the code from the `Dog` constructor to handle assigning the `name`, `breed`, and `weight` properties.

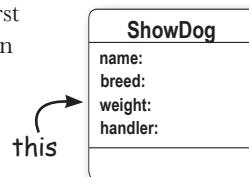
Here's how to think about how this works. First, we call `ShowDog` with the `new` operator. Remember that the `new` operator makes a new, empty object, and assigns it to the variable `this` in the body of `ShowDog`.

```
var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
```

Then, we execute the body of the `ShowDog` constructor function. The first thing we do is call `Dog`, using the `call` method. That calls `Dog`, passing in `this`, and the `name`, `breed`, and `weight` parameters as arguments.

```
function ShowDog(name, breed, weight, handler) {
  Dog.call(this, name, breed, weight);
  ←
  function Dog(name, breed, weight) {
    this.name = name;
    this.breed = breed;
    this.weight = weight;
  }
  ←
  this.handler = handler;
}
```

We execute the body of `Dog` as normal, except that this is a `ShowDog`, not a `Dog` object.



The `this` object created by `new` for `ShowDog` gets used as `this` in the body of `Dog`.

Once the `Dog` function completes (and remember, it is not going to return anything because we didn't call it with `new`), we complete the code in `ShowDog`, assigning the value of the parameter `handler` to the `this.handler` property.

Then, because we used `new` to call `ShowDog`, an instance of `ShowDog` is returned, complete with its `name`, `breed`, `weight`, and `handler`.



These three properties are assigned to `this` by the code in the `Dog` function.

This property is assigned to `this` by the code in the `ShowDog` function.

# The final test drive



Well done, you've created a fantastic design that we're sure Webville Kennel is going to love. Take all your dogs for one final test run so they can show off all their doggy capabilities.

```

function ShowDog(name, breed, weight, handler) {
    Dog.call(this, name, breed, weight);
    this.handler = handler;
}

ShowDog.prototype = new Dog();
ShowDog.prototype.constructor = ShowDog;
ShowDog.prototype.league = "Webville";
ShowDog.prototype.stack = function() {
    console.log("Stack");
};

ShowDog.prototype.bait = function() {
    console.log("Bait");
};

ShowDog.prototype.gait = function(kind) {
    console.log(kind + "ing");
};

ShowDog.prototype.groom = function() {
    console.log("Groom");
};

var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);
var scotty = new ShowDog("Scotty", "Scottish Terrier", 15, "Cookie");
var beatrice = new ShowDog("Beatrice", "Pomeranian", 5, "Hamilton");

fido.bark();
fluffy.bark();
spot.bark();
scotty.bark();
beatrice.bark();
scotty.gait("Walk");
beatrice.groom();

```

Webville Kennel is going to love this!



← We've brought all the ShowDog code together here. Add this to the file with your Dog code to test it.

We've added some test code below.

← Create some dogs and some show dogs.

← Put them through their paces and make sure they're all doing the right thing.

JavaScript console
Fido says Woof!
Fluffy says Woof!
Spot says Yip!
Scotty says Yip!
Beatrice says Yip!
Walking
Groom

## there are no Dumb Questions

**Q:** When we made the dog instance we used for the show dog prototype, we called the Dog constructor with no arguments. Why?

**A:** Because all we need from that dog instance is the fact that it inherits from the dog prototype. That dog instance isn't a specific dog (like Fido or Fluffy); it's simply a generic dog instance that inherits from the dog prototype.

Also, all the dogs that inherit from the show dog prototype define their own name, breed, and weight. So even if that dog instance did have values for those properties, we'd never see them because the show dog instances will always override them.

**Q:** So what happens to those properties in the dog instance we use for the show dog prototype?

**A:** They never get assigned values, so they are all undefined.

**Q:** If we never set the ShowDog's prototype property to a dog instance, what happens?

**A:** Your show dogs will work fine, but they won't inherit any behavior from the dog prototype. That means they won't be able to bark, run, or wag, nor will they have the "Canine" species property. Give it a try yourself. Comment out the line of code where we set ShowDog.prototype to new Dog() and then try making Scotty bark. What happens?

**Q:** Could I create an object literal and use that as the prototype?

**A:** Yes. You can use any object as the prototype for ShowDog. Of course, if you do that, your show dogs won't inherit anything from the dog prototype. They'll inherit the properties and methods you put in your object literal instead.

**Q:** I accidentally put the line of code to assign ShowDog.prototype to the instance of dog below where I created my scotty instance, and my code didn't work. Why?

**A:** Because when you create scotty (an instance of ShowDog), it gets the prototype that's assigned to ShowDog.prototype at the time when you create it. So if you don't assign the dog instance object to the prototype until after you create scotty, then scotty will have a different object as its prototype (the object you get by default with the ShowDog constructor). And that object doesn't have any of the Dog prototype's properties. You should assign the show dog prototype first thing after you create the constructor, but before you add anything to the prototype, or create any instances of ShowDog.

**Q:** If I change a property in the dog prototype, like changing species from "Canine" to "Feline", will that affect the show dogs I've created?

**A:** Yes, anything you change in the prototype will affect any instances that inherit from that prototype in the chain, no matter how many links you have in your chain.

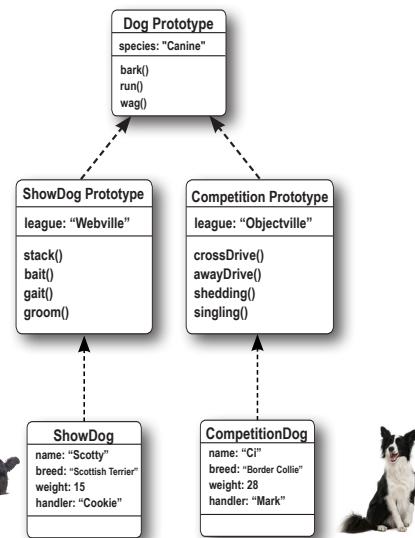
**Q:** Is there a limit to how long my prototype chains can be?

**A:** Theoretically, no, but in practice, maybe. The longer your prototype chain, the more work it is to resolve a method or property. That said, runtime systems are often quite good at optimizing these lookups.

In general, you're not going to need designs that require that many levels of inheritance. If you do, you'll probably want to take another look at your design.

**Q:** What if I have another category of dogs, like competition dogs. Can I create a competition dog prototype that inherits from the same dog prototype as the show dog prototype does?

**A:** Yes, you can. You'll need to create a separate dog instance to act as your competition dog prototype, but once you've done that you'll be good to go. Just follow the same steps we used here to create the show dog prototype.



# The chain doesn't end at dog

You've already seen a couple of prototype chains—we have the original dog prototype that our dog objects inherit from, and we have the more specialized show dog instances that inherit first from the show dog prototype, then from the dog prototype, and finally from Object.

But in both cases, is dog the end of the chain? Actually it isn't, because dog has its own prototype, Object.

In fact, every prototype chain you ever create will end in Object. That's because the default prototype for any instance you create (assuming you don't change it) is Object.

## What is Object?

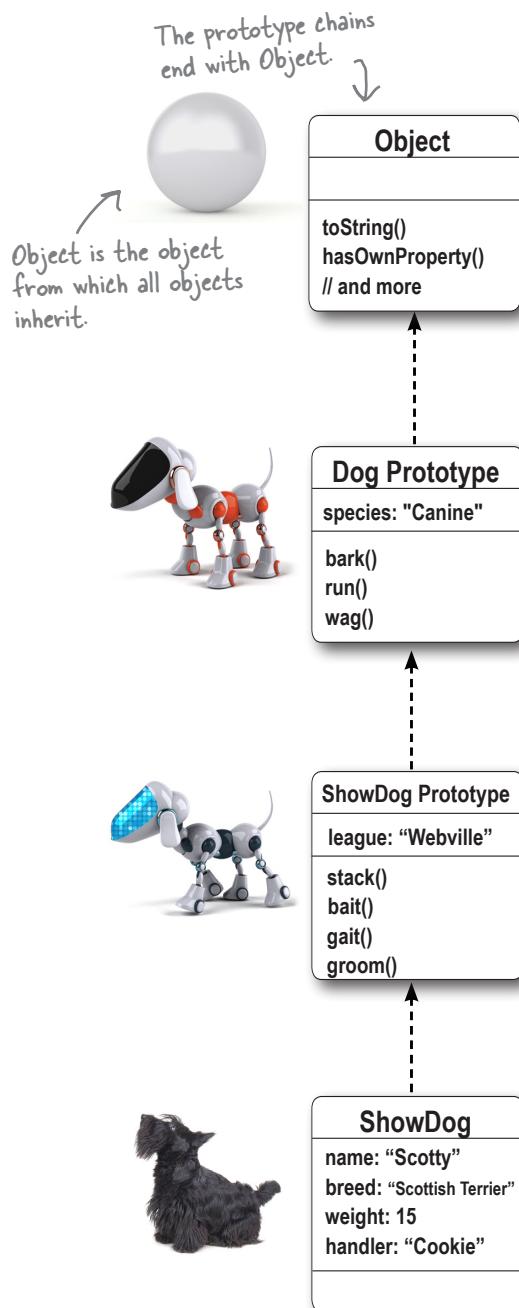
Think of Object like the primordial object. It's the object that all objects initially inherit from. And Object implements a few key methods that are a core part of the JavaScript object system. Many of these you won't use on a daily basis, but there are some methods you'll see commonly used.

One of those you've already seen in this chapter: `hasOwnProperty`, which is inherited by every object (again, because every object ultimately inherits from Object). Remember, `hasOwnProperty` is the method we used earlier to determine if a property is in an object instance or in one of its prototypes.

Another method inherited from Object is the `toString` method, which is commonly overridden by instances. This method returns a String representation of any object. We'll see in a bit how we can override this method to provide a more accurate description of our own objects.

## Object as a prototype

So whether you realized it or not, every object you've ever created has had a prototype, and it's been Object. You can set an object's prototype to another kind of object, like we did with the show dog prototype, but ultimately, all prototype chains eventually lead to Object.



## Using inheritance to your advantage... by overriding built-in behavior

If you're inheriting from a built-in object you can override methods in those objects. One common example is the `toString` method of `Object`. All objects inherit from `Object`, so all objects can use the `toString` method to get a simple string representation of any object. For instance, you might use it with `console.log` to display your object in the console:

```
function Robot(name, year, owner) {  
    this.name = name;  
    this.year = year;  
    this.owner = owner;  
}  
  
var toy = new Robot("Toy", 2013, "Avary");  
  
console.log(toy.toString());
```

As you can see, the `toString` method doesn't do a very good job of converting the toy robot into a string. So we can override the `toString` method and write one that creates a string specifically for `Robot` objects:

```
function Robot(name, year, owner) {  
    // same code here  
}  
  
Robot.prototype.toString = function() {  
    return this.name + " Robot belonging to " + this.owner;  
};  
  
var toy = new Robot("Toy", 2013, "Avary");  
  
console.log(toy.toString());
```

Notice that the `toString` method can be invoked even if you're not calling it directly yourself. For instance, if you use the `+` operator to concatenate a string and an object, JavaScript will use the `toString` method to convert your object to a string before concatenating it with the other string.

```
console.log("Robot is: " + toy);
```

The `toy` object will get converted to a string using `toString` before it's concatenated. If `toy` has overridden `toString`, it will use that method.

JavaScript console  
[Object object]

↑ The `toString` method we're inheriting from `Object` doesn't do a very good job.

JavaScript console  
Toy Robot belonging to Avary

↑ Much better! Now we're using our own `toString` method.

Toy? This Robot's running an Arduino stack and even controllable with JavaScript!





## DANGER ZONE

Once you start overriding properties and methods, it's easy to get a little carried away. It's especially important to be careful when overriding properties and methods in built-in objects, because you don't want to change the behavior of other code that might rely on these properties to do certain things.

So if you're thinking of overriding properties in Object, read this Safety Guide first. Otherwise, you might end up blowing up your code in unexpected ways. (Translation: you'll have bugs that are really hard to track down.)



## DO NOT OVERRIDE

Here are the properties in Object you don't want to override:

`constructor`

The constructor property points to the constructor function connected to the prototype.

`hasOwnProperty`

You know what the hasOwnProperty method does.

`isPrototypeOf`

isPrototypeOf is a method you can use to find out if an object is a prototype of another object.

`propertyIsEnumerable`

The propertyIsEnumerable method checks to see if a property can be accessed by iterating through all the properties of an object.

## OKAY TO OVERRIDE

Here are the properties in Object that you can override now that you know your way around prototypes, and know how to override safely:

`toString`

toLocaleString is a method, like toString, that converts an object to a string. This method is designed to be overridden to provide a localized string (say, for your country/language) about an object.

`toLocaleString`

valueOf is another method designed to be overridden. By default it just gives you the object you call it on. But you can override that to return another value instead if you want.

`valueOf`

# Using inheritance to your advantage... by extending a built-in object

You already know that by adding methods to a prototype, you can add new functionality to all instances of that prototype. This applies not only to your own objects, but also to built-in objects.

Take the String object for instance—you've used String methods like `substring` in your code, but what if you want to add your own method so that any instance of String could make use of it? We can use the same technique of extending objects through the prototype on Strings too.

Let's say we want to extend the String prototype with a method, `cliche`, that returns true if the string contains a known cliché. Here's how we'd do that:

```
String.prototype.cliche = function() {
    var cliche = ["lock and load", "touch base", "open the kimono"];

    for (var i = 0; i < cliche.length; i++) {
        var index = this.indexOf(cliche[i]);
        if (index >= 0) {
            return true;
        }
    }
    return false;
};
```

Note that `this` is the string on which we call the method `cliche`.

Here we're adding a method, `cliche`, to the String prototype.

We define offending phrases to look for.

And then we use the String's `indexOf` function to see if the string matches any of the clichés. If it does we immediately return true.

Now let's write some code to test the method:

To test let's create some sentences, including a couple that use clichés.

```
var sentences = ["I'll send my car around to pick you up.",
    "Let's touch base in the morning and see where we are",
    "We don't want to open the kimono, we just want to inform them."];
```

```
for (var i = 0; i < sentences.length; i++) {
    var phrase = sentences[i];
```

Each sentence is a string, so we can call its `cliche` method.

```
    if (phrase.cliche()) {
```

Notice that we're not creating a string using the `String` constructor and `new`. JavaScript is converting each string to a `String` object behind the scenes for us, when we call the `cliche` method.

```
        console.log("CLICHE ALERT: " + phrase);
```

```
}
```

If true is returned, we know we have a cliché in the string.

# Test driving the cliché machine



Get the code into a HTML file, open your browser and load it up.

Check your console and you should see this output:

Works great. If only we could convince Corporate America to install this code!

JavaScript console

**CLIQUE ALERT:** Let's touch base in the morning and see where we are

**CLIQUE ALERT:** We don't want to open the kimono, we just want to inform them.



## Watch it!

**Be careful when you extend built-in objects like String with your own methods.**

Make sure the name you choose for your method doesn't conflict with an existing method in the object. And if you link to other code, be aware of other custom extensions that code may have (and again, watch for name clashes). And finally, some built-in objects aren't designed to be extended (like Array). So do your homework before you start adding methods to built-in objects.



## Exercise

Your turn. Write a method, palindrome, that returns true if a string reads the same forwards and backwards. (Just one word, don't worry about palindrome phrases.) Add the method to the String.prototype and test. Check your answer at the end of the chapter.

# JavaScript

## Grand Unified Theory of Everything

Congratulations. You've taken on the task of learning an entirely new programming language (maybe your first language) and you've done it. Assuming you've made it this far, you now know more JavaScript than pretty much everyone.

More seriously, if you've made it this far in the book, you are well on your way to becoming a JavaScript expert. Now all you need is more experience designing and coding web applications (or any kind of JavaScript application for that matter).



We're using the logic that about 5.9 billion people don't know JavaScript at all, and so those who do are pretty much a rounding error, which means you know more JavaScript than just about anyone.

## Better living through objects

When you're learning a complex topic like JavaScript, it's hard to see the forest for the trees. But, once you understand most of JavaScript, it's easier to step back and check out the forest.

When you're learning JavaScript, you learn about pieces of it at a time: you learn about primitives (that can, at any moment, be used like an object), arrays (which kinda act like objects at times), functions (which, oddly, have properties and methods like objects), constructors (which feel like part function, part object) and well... objects themselves. It all seems rather complex.

Well, with the knowledge you have now, you can sit back, relax, take a cleansing breath, and meditate on the mantra "everything is an object."

Because you see, everything *is* an object—oh, sure we have a few primitives, like booleans, numbers and strings, but we already know that we can treat those as objects anytime we need to. We have some built-in types too, like Date, Math and RegEx, but those are just objects too. Even arrays are objects, and as you saw, the only reason they look different is because JavaScript provides some nice "syntactic sugar" we can use to make creating and accessing objects easier. And of course we have objects themselves, with the simplicity of object literals and the power of the prototypal object system.

But what about functions? Are they really objects? Let's find out:

```
function meditate() {  
    console.log("Everything is an object...");  
}  
  
alert(meditate instanceof Object);
```

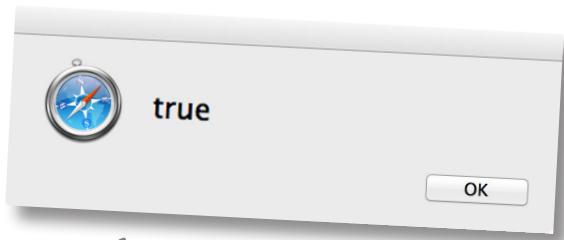


So it's true: functions are just objects. But, really, this shouldn't be a big surprise at this point. After all we can assign functions to variables (like objects), pass them as arguments (like objects), return them from functions (like objects), and we've even seen they have properties, like this one:

**Dog . constructor**

↑  
Remember this  
is a function.

↑  
And this is a  
property.



It's true! Functions are objects too.

And there's nothing stopping you from adding your own properties to a function should that come in handy. And, by the way, just to bring it all full circle, have you considered that a method is just a property in an object that is set to an anonymous function expression?

## Putting it all together

A lot of JavaScript's power and flexibility comes from the interplay between how we use functions and objects, and the fact that we can treat them as first class values. If you think about the powerful programming concepts we've studied—constructors, closures, creating objects with behavior that we can reuse and extend, parameterizing the behavior of functions, and so on—all these techniques have relied on your understanding of advanced objects and functions.

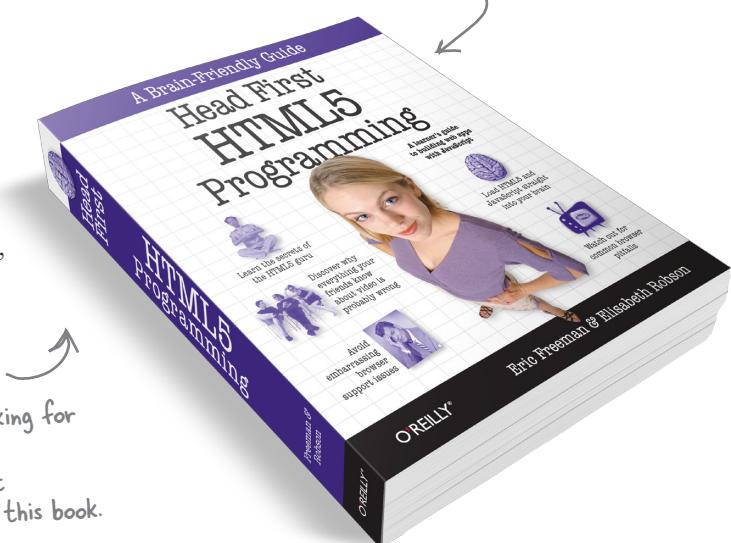
Well, now you're in a position to take this all even further...

## What's next?

Now that you've got all the fundamentals down, it's time to take it all further. Now you're ready to really put your experience to use with the browser and its programming interfaces. You can do that by picking up *Head First HTML5 Programming*, which will take you through how to add geolocation, canvas drawing capabilities, local storage, web workers and more into your applications. But before you put this book down, be sure to read the appendix for a great list of other topics to explore.

This is a rapidly evolving topic, so before you go looking for Head First HTML5 Programming, hit <http://wickedlysmart.com/javascript> for our latest recommendations and any updates and revisions for this book.

Be sure to visit <http://wickedlysmart.com/javascript> for follow-up materials for this book and, as your next mission, should you accept it...





## BULLET POINTS

- JavaScript's object system uses **prototypal inheritance**.
- When you create an instance of an object from a constructor, the instance has its own customized properties and a copy of the methods in the constructor.
- If you add properties to a constructor's prototype, all instances created from that constructor **inherit** those properties.
- Putting properties in a prototype can reduce runtime code duplication in objects.
- To **override** properties in the prototype, simply add the property to an instance.
- A constructor function comes with a default **prototype** that you can access with the function's prototype property.
- You can assign your own object to the prototype property of a constructor function.
- If you use your own prototype object, make sure you set the constructor function correctly to the constructor property for consistency.
- If you add properties to a prototype after you've created instances that inherit from it, all the instances will immediately inherit the new properties.
- Use the **hasOwnProperty** method on an instance to find out if a property is defined in the instance.
- The method **call** can be used to invoke a function and specify the object to be used as **this** in the body of the function.
- **Object** is the object that all prototypes and instances ultimately inherit from.
- Object has properties and methods that all objects inherit, like **toString** and **hasOwnProperty**.
- You can override or add properties to built-in objects like **Object** and **String**, but take care when doing so as your changes can have far-ranging effects.
- In JavaScript, almost everything is an object, including functions, arrays, many built-in objects, and all the custom objects you make yourself.

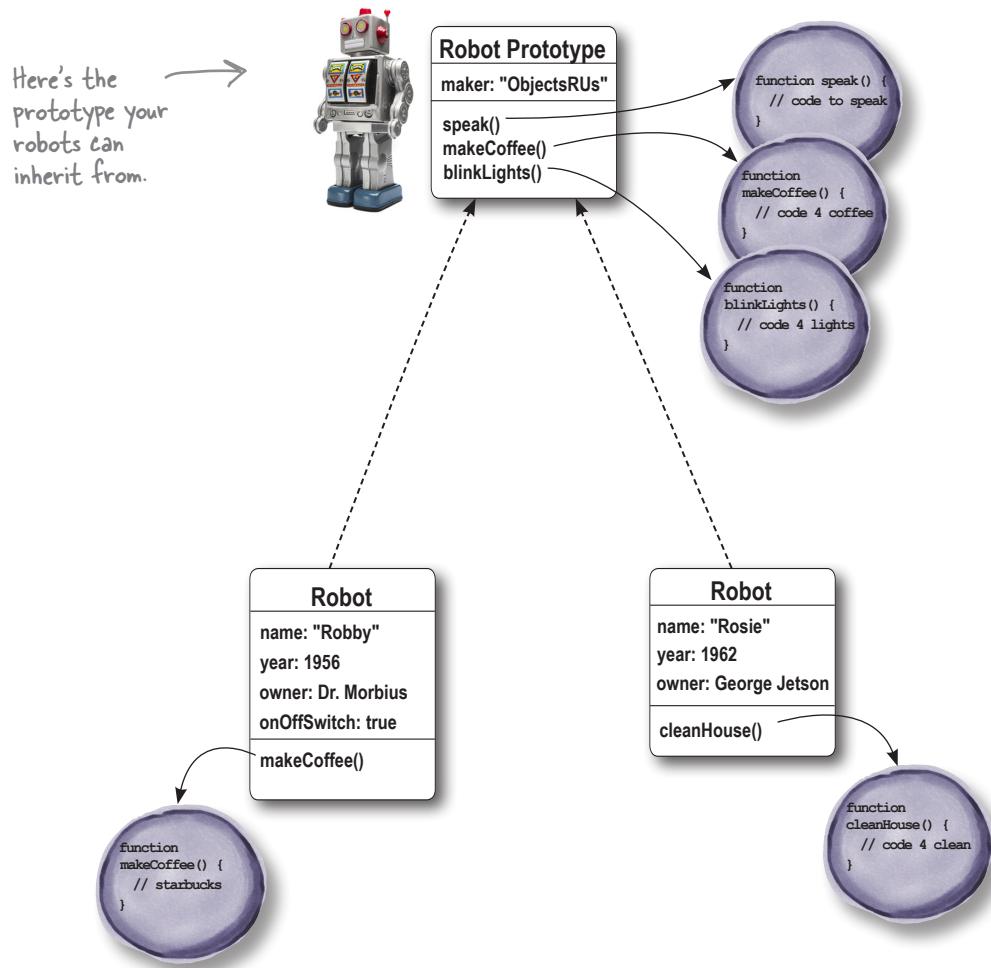




## Code Magnets Solution

We had an object diagram on the fridge, and then someone came and messed it up. Can you help put it back together? To reassemble it, we need two instances of the robot prototype. One is Robby, created in 1956, owned by Dr. Morbius, has an on/off switch and runs to Starbucks for coffee. We've also got Rosie, created in 1962, who cleans house and is owned by George Jetson. Good luck (oh, and there might be some extra magnets below)!

Here's our solution:





## Exercise Solution

Remember our object diagram for the Robby and Rosie robots? We're going to implement that now. We've already written a Robot constructor for you along with some test code. Your job is to set up the robot prototype and to implement the two robots. Make sure you run them through the test code. Here's our solution.

```

function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

Robot.prototype.maker = "ObjectsRUs";
Robot.prototype.speak = function() {
    alert("Warning warning!!!");
};

Robot.prototype.makeCoffee = function() {
    alert("Making coffee");
};

Robot.prototype.blinkLights = function() {
    alert("Blink blink!");
};

var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

robby.onOffSwitch = true;
robby.makeCoffee = function() {
    alert("Fetching a coffee from Starbucks.");
};

rosie.cleanHouse = function() {
    alert("Cleaning! Spic and Span soon...");
};

console.log(robby.name + " was made by " + robby.maker +
            " in " + robby.year + " and is owned by " + robby.owner);
robby.makeCoffee();
robby.blinkLights();
console.log(rosie.name + " was made by " + rosie.maker +
            " in " + rosie.year + " and is owned by " + rosie.owner);
rosie.cleanHouse();

```

Here's the basic Robot constructor.

Here we're setting up the prototype with a maker property...

...and three methods that are shared by all robots.

We create our robots, Robby and Rosie here.

Here, we're adding a custom property to Robby, as well as a custom method for making coffee (by going to Starbucks).

And Rosie also gets a custom method to clean the house (why do the girl robots have to clean?).

Here's our output (plus some alerts we're not showing).

JavaScript console

```

Robby was made by ObjectsRUs in 1956
and is owned by Dr. Morbius
Rosie was made by ObjectsRUs in 1962
and is owned by George Jetson

```



## Exercise Solution

Robby and Rosie are being used in a Robot game. You'll find the code for them below. In this game, whenever a player reaches level 42, a new robot capability is unlocked: the laser beam capability. Finish the code below so that at level 42 both Robby and Rosie get their laser beams. Here's our solution.

```
function Game() {
    this.level = 0;
}

Game.prototype.play = function() {
    // player plays game here
    this.level++;
    console.log("Welcome to level " + this.level);
    this.unlock();   We call unlock each time we play the game but no
                    ↙ power is unlocked until the level reaches 42.
}

Game.prototype.unlock = function() {
    if (this.level === 42) {
        Robot.prototype.deployLaser = function () {
            console.log(this.name + " is blasting you with laser beams.");
        }
    }
}

function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

var game = new Game();
var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

while (game.level < 42) {
    game.play();
}

robby.deployLaser();
rosie.deployLaser();
```

JavaScript console

```
Welcome to level 1
Welcome to level 2
Welcome to level 3
...
Welcome to level 41
Welcome to level 42
Rosie is blasting you with
laser beams.
```

A sample of our output. When you finish your code, give it a play and see which robot wins and gets to blast its laser beams!

Here's the trick to this game: when you reach level 42, a new method is added to the prototype. That means all robots inherit the ability to deploy lasers!





## Exercise Solution

We've added a new capability to our robots, Robby and Rosie: they can now report when they have an error through the `reportError` method. Trace the code below, paying particular attention to where this method gets its error information, and to whether it's coming from the prototype or the robot instance.

Here's our solution.

```

function Robot(name, year, owner) {
    this.name = name;
    this.year = year;
    this.owner = owner;
}

Robot.prototype.maker = "ObjectsRUs";
Robot.prototype.errorMessage = "All systems go.";
Robot.prototype.reportError = function() {
    console.log(this.name + " says " + this.errorMessage);
};

Robot.prototype.spillWater = function() {
    this.errorMessage = "I appear to have a short circuit!";
};

var robby = new Robot("Robby", 1956, "Dr. Morbius");
var rosie = new Robot("Rosie", 1962, "George Jetson");

rosie.reportError();
robby.reportError();
robby.spillWater(); ←
rosie.reportError();
robby.reportError();

console.log(robby.hasOwnProperty("errorMessage"));
console.log(rosie.hasOwnProperty("errorMessage"));

```

The `reportError` method only uses the value of `errorMessage`, so it doesn't override the property.

The `spillWater` method assigns a new value to `this.errorMessage`, which will override the property in the prototype in any robot that calls this method.

We call the `spillWater` method on Robby, so Robby gets his own `errorMessage` property, which overrides the property in the prototype.

true

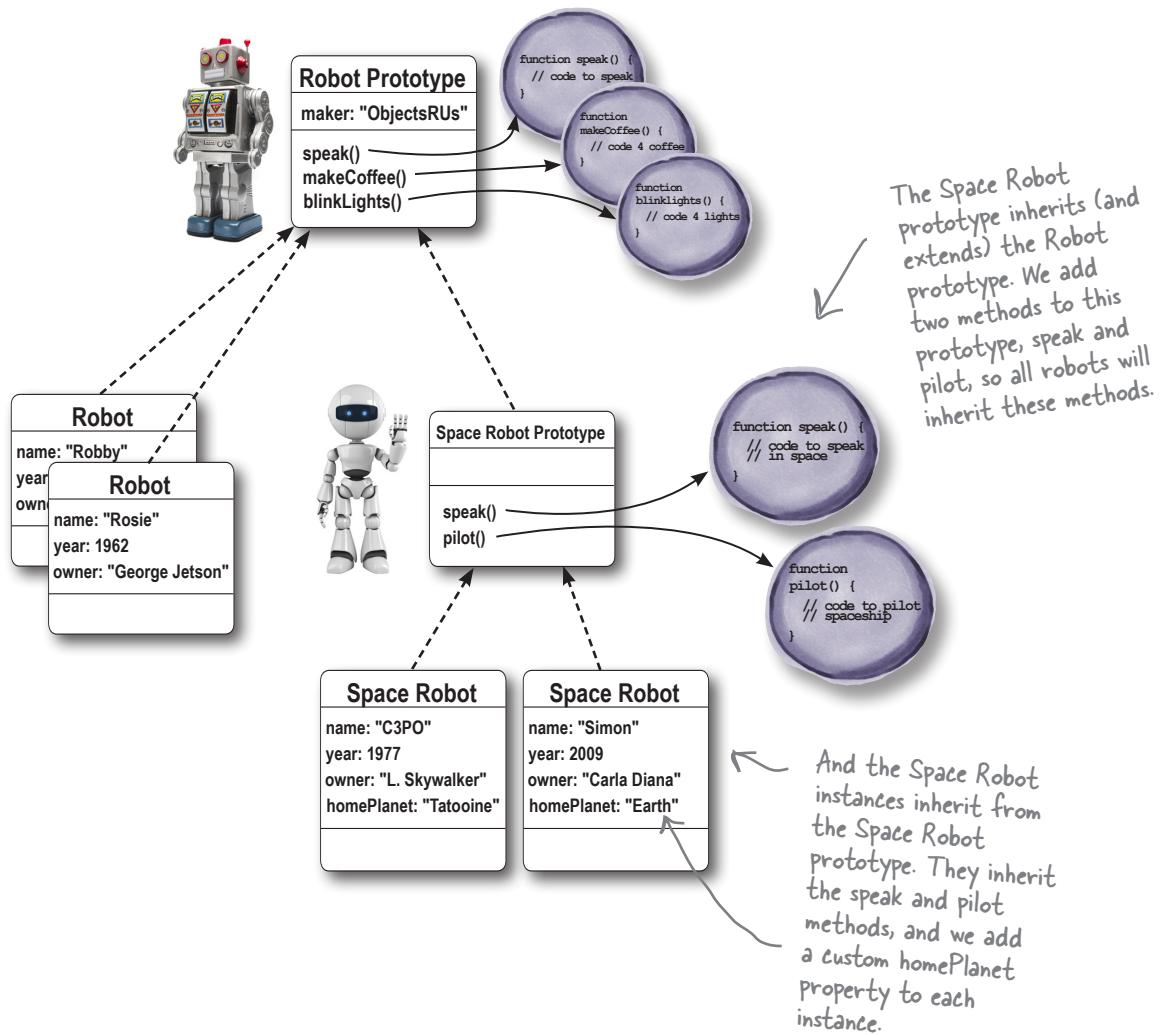
false

But we never call `spillWater` on Rosie, so she inherits the property in the prototype.



## Code Magnets Solution

We had another object diagram on the fridge, and then someone came and messed it up. Again!! Can you help put it back together? To reassemble it we need a new line of Space Robots that inherit properties from Robots. These new Space Robots override the Robot's speaking functionality, and extend Robots with piloting functionality and a new property, homePlanet. Here's our solution.





Your turn. Add a SpaceRobot line of robots to the ObjectsRUs line of robots. These robots should of course be able to do everything that robots can do, plus some extra behavior for space robots. We've started the code below, so finish it up and then test it. Here's our solution.

```
function SpaceRobot(name, year, owner, homePlanet) {
    this.name = name;
    this.year = year;
    this.owner = owner;
    this.homePlanet = homePlanet;
}

SpaceRobot.prototype = new Robot();
SpaceRobot.prototype.speak = function() {
    alert(this.name + " says Sir, If I may venture an opinion...");;
};

SpaceRobot.prototype.pilot = function() {
    alert(this.name + " says Thrusters? Are they important?");;
};

var c3po = new SpaceRobot("C3PO", 1977, "Luke Skywalker", "Tatooine");
c3po.speak();
c3po.pilot();
console.log(c3po.name + " was made by " + c3po.maker);

var simon = new SpaceRobot("Simon", 2009, "Carla Diana", "Earth");
simon.makeCoffee();
simon.blinkLights();
simon.speak();
```

The SpaceRobot constructor is similar to the Robot constructor, except we have an extra homePlanet property for the SpaceRobot instances.

We want the SpaceRobot prototype to inherit from the Robot prototype, so we assign a Robot instance to the SpaceRobot constructor's prototype property.

These two methods are added to the prototype.

Here's our output (plus some alerts we're not showing).

JavaScript console

C3PO was made by ObjectsRUs



## Exercise Solution

Your turn. Write a method, `palindrome`, that returns true if a string reads the same forward and backward. Add the method to the `String.prototype` and test. Here's our solution (for one word palindromes only).

```
String.prototype.palindrome = function() {
    var len = this.length-1;           ← First we get the length of the string.
    for (var i = 0; i <= len; i++) {
        if (this.charAt(i) !== this.charAt(len-i)) { ← Then we iterate over each character
            return false;               ← in the string, and test to see if
        }                                ← the character at i is the same as
        if (i === (len-i)) {           ← the character at len-i (i.e., the
            return true;              ← character at the other end).
        }
    }
    return true;                      ← If we get to where i is in the middle of the
                                    ← string, or we get to the end of the loop, we
                                    ← return true because we've got a palindrome.
};

var phrases = ["eve", "kayak", "mom", "wow", "Not a palindrome"];   ← Here are some words to test.

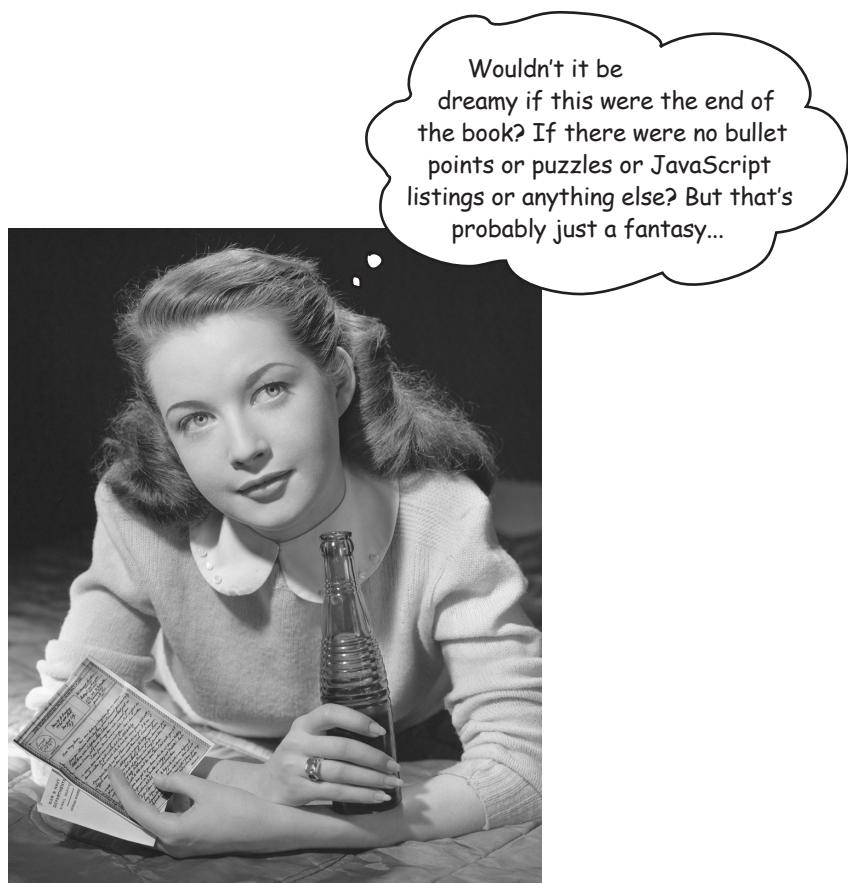
for (var i = 0; i < phrases.length; i++) {
    var phrase = phrases[i];
    if (phrase.palindrome()) { ← We just iterate through
        console.log("'" + phrase + "' is a palindrome");
    } else {                   ← each word in the array
        console.log("'" + phrase + "' is NOT a palindrome");
    }
}
```



## Super Advanced Solution

```
String.prototype.palindrome = function() {
    var r = this.split("").reverse().join("");
    return (r === this.valueOf());
}
```

Here, we first split the string into an array of letters, with each letter being one item in the array. We then reverse the array and join all the letters back up into a string. If the original string's value equals the new string, we've got a palindrome. Note, we have to use `valueOf` here, because this is an object, not a string primitive like `r`, so if we don't, we'd be comparing a string to an object, and they wouldn't be equal even if this is a palindrome.



**Congratulations!**  
**You made it to the end.**

**Of course, there's still an appendix.  
And the index.  
And the colophon.  
And then there's the website...  
There's no escape, really.**

## Appendix: leftovers

# \*The top ten topics (we didn't cover)\*



**We've covered a lot of ground, and you're almost finished with this book.** We'll miss you, but before we let you go, we wouldn't feel right about sending you out into the world without a little more preparation. We can't possibly fit everything you'll need to know into this relatively small chapter. Actually, we *did* originally include everything you need to know about JavaScript Programming (not already covered by the other chapters), by reducing the type point size to .00004. It all fit, but nobody could read it. So we threw most of it away, and kept the best bits for this Top Ten appendix.

This really *is* the end of the book. Except for the index, of course (a must-read!).

## #1 jQuery

jQuery is a JavaScript library that is aimed at reducing and simplifying much of the JavaScript code and syntax that is needed to work with the DOM and add visual effects to your pages. jQuery is an enormously popular library that is widely used and expandable through its plug-in model.

Now, there's nothing you can do in jQuery that you can't do with JavaScript (as we said, jQuery is just a JavaScript library); however, it does have the power to reduce the amount of code you need to write.

jQuery's popularity speaks for itself, although it can take some getting used to if you are new to it. Let's check out a few things you can do in jQuery and we encourage you to take a closer look if you think it might be for you.

For starters, remember all the `window.onload` functions we wrote in this book? Like:

```
window.onload = function() {  
    alert("the page is loaded!");  
}
```

Here's the same thing using jQuery:

```
$(document).ready(function() { ← Just like our version, when the document  
    alert("the page is loaded!"); is ready, invoke my function.  
});
```

Or you can shorten this even more, to:

```
$(function() { ← This is cool, but as you can see  
    alert("the page is loaded!"); it takes a little getting used to  
}); at first. No worries, it becomes second-nature fast.
```

So what about getting elements from the DOM? That's where jQuery shines. Let's say you have an `<a>` element in your page with an id of "buynow" and you want to assign a click handler to the click event on that element (like we've done a few times in this book). Here's how you do that:

```
$(function() { ← So what's going on here? First we're setting up a function  
    $("#buynow").click(function() { that is called when the page is loaded.  
        alert("I want to buy now!");  
    });  
}); ← Next we're grabbing the element with  
      a "buynow" id (notice jQuery uses CSS  
      syntax for selecting elements).  
      ← And then we're calling a jQuery method, click,  
      on the result to set the onclick handler.
```

↖ A working knowledge of  
jQuery is a good skill these  
days on the job front and for  
understanding others' code.

That's really just the beginning; we can just as easily set the click handler on every `<a>` element in the page:

```
$(function() {
    $("a").click(function() {
        alert("I want to buy now!");
    });
});
```

To do that, all we need to do is use the tag name.



Compare this to the code you'd write to do this if we were using JavaScript without jQuery.

Or, we can do things that are much more complex:

```
$(function() {
    $("#playlist > li").addClass("favorite");
});
```

Like find all the `<li>` elements that are children of the element with an id of `playlist`.



And then add the class "favorite" to all the elements.



Actually this is jQuery just getting warmed up; jQuery can do things much more sophisticated than this.

There's a whole 'nother side of jQuery that allows you to do interesting interface transformations on your elements, like this:

```
$(function() {
    $("#specialoffer").click(function() {
        $(this).fadeOut(800, function() {
            $(this).fadeIn(400);
        });
    });
});
```



This makes the element with an id of `specialoffer` fade out and then fade back in at different rates.

As you can see, there's a lot you can do with jQuery, and we haven't even talked about how we can use jQuery to talk to web services, or all the plug-ins that work with jQuery. If you're interested, the best thing you can do is point your browser to <http://jquery.com/> and check out the tutorials and documentation there.



And, check out Head First jQuery too!

## #2 Doing more with the DOM

We've touched on some of the things you can do with the DOM in this book, but there's a lot more to learn. The objects that represent the document in your page—that is, the document object, and the various element objects—are chock full of properties and methods you can use to interact with and manipulate your page.

You already know how to use `document.getElementById` and `document.getElementsByTagName` to get elements from the page. The document object has these other methods you can use to get elements, too:

`document.getElementsByClassName` ← Pass this method the name of a class, and you'll get back all elements that have that class, as a NodeList.

`document.getElementsByName` ← This method retrieves elements that have a name attribute with a value that matches the name you pass it.

`document.querySelector` ← This method takes a selector (just like a CSS selector) and returns the first element that matches.

`document.querySelectorAll` ← This method also takes a selector, but returns all the elements that match, as a NodeList.

Here's how you'd use `document.querySelector` to match a list item element with the class "song" that's nested in a `<ul>` element with the id "playlist":

```
var li = document.querySelector("#playlist .song");
```

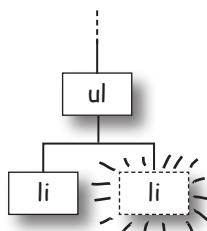
This says match the element with the id `playlist`, and then match the first element with the class `song`.

Notice how this selector is just like one you'd write in CSS?

What if you want to add new elements to your page from your code? You can use a combination of document object methods and element methods to do that, like this:

```
var newItem = document.createElement("li"); ← First, we create a new <li>
newItem.innerHTML = "Your Random Heart"; ← element, and set its content
                                            to a string.
```

```
var ul = document.getElementById("playlist"); ← Then we get the <ul> element
ul.appendChild(newItem); ← we want to add the new <li>
                        to (as a child element), and
                        append the <li> to the <ul>.
```



There's a lot more you can do with the DOM using JavaScript. For a good introduction, check out *Head First HTML5 Programming*.

## #3 The Window Object

You've heard of the DOM, but you should know there's also a BOM, or Browser Object Model. It's not really an official standard, but all browsers support it through the `window` object. You've seen the `window` object in passing when we've used the `window.onload` property, and you'll remember we can assign an event handler to this property that is triggered when the browser has fully loaded a page.

You've also seen the `window` object when we've used the `alert` and `prompt` methods, even though it might not have been obvious. The reason it wasn't obvious is that `window` is the object that acts as the global namespace. When you declare any kind of global variable or define a global function, it is stored as a property in the `window` object. So for every call we made to `alert`, we could have instead called `window.alert`, because it's the same thing.

Another place you've used the `window` object without knowing it is when you've used the `document` object to do things like get elements from the DOM with `document.getElementById`. The `document` object is a property of `window`, so we could write `window.document.getElementById`. But, just like with `alert`, we don't have to, because `window` is the global object, and it is the default object for all the properties and methods we use from it.

In addition to being the global object, and supplying the `onload` property and the `alert` and `prompt` methods, the `window` object supplies other interesting browser-based properties and methods. For instance, it's common to make use of the width and height of the browser window to tailor a web page experience to the size of the browser. You can access these values like this:

`window.innerWidth` ← Use these properties to get the browser window's width and height in pixels. Note that older browsers don't always expose these properties.  
`window.innerHeight`

Check out the W3C documentation\* for more on the `window` object. Here are a few common methods and properties:

`window.close()` ← This method closes the browser window.

`window.setTimeout()` ← You already know these methods;  
`window.setInterval()` they're supplied by the `window` object.

`window.print()` ← Initiates printing the page to your printer.

`window.confirm()` ← This method is similar to `prompt`, only it gives the user the choice of an `Okay` or `Cancel` button.

`window.history` ← This property is an object containing the browsing history.

`window.location` ← This property is the URL of the current page. You can also set this property to direct the browser to load a new page.

\* <http://www.w3.org/html/wg/drafts/html/CR/browsers.html#the-window-object>

## #4 Arguments

An object named `arguments` is available in every function when that function is called. You won't ever see this object in the parameter list, but it is available nevertheless every time a function is called, in the variable `arguments`.

The `arguments` object contains every argument passed to your function, and it can be accessed in an array-like manner. You can use `arguments` to create a function that accepts a variable number of arguments, or create a function that does different things depending on the number of arguments passed to it. Let's see how `arguments` works with this code:

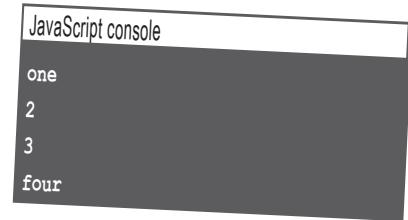
We're not going to define any formal parameters for now. We'll just use the `arguments` object.

```
function printArgs() {  
    for (var i = 0; i < arguments.length; i++) {  
        console.log(arguments[i]);  
    }  
}  
  
printArgs("one", 2, 1+2, "four");
```

Like an array, `arguments` has a `length` property.

And we can access each argument using array notation.

Here we call `printArgs` with four arguments.

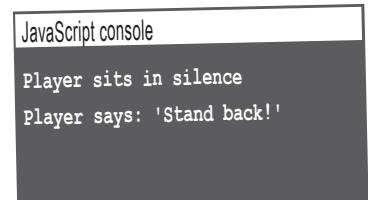


While `arguments` looks just like an array, it is not actually an array; it's an object. It has a `length` property, and you can iterate over it and access items in it using bracket notation, but that's where the similarity with an array ends. Also, note that you can use both parameters and the `arguments` object in the same function. Let's write one more piece of code to see how a function with a variable number of arguments might be written:

We can define parameters like normal. In this case, using a parameter helps indicate how to use this function.

```
function emote(kind) {  
    if (kind === "silence") {  
        console.log("Player sits in silence");  
    } else if (kind === "says") {  
        console.log("Player says: '" + arguments[1] + "'");  
    }  
}  
  
emote("silence");  
emote("says", "Stand back!");
```

If the first argument is "silence" then we don't expect another. If the first argument is "says", then we use `arguments[1]` to get the second argument.





## #5 Handling exceptions

JavaScript is a fairly forgiving language, but now and then things go wrong—wrong enough that the browser can't continue executing your code. When that happens, your page stops working, and if you look in the console you're likely to see an error. Let's take a look at an example of some code that causes an error. Start by creating a simple HTML page with a single element in the body:

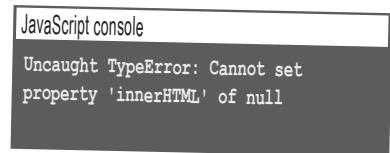
```
<div id="message"></div>
```

Now, add the following JavaScript:

```
window.onload = function() {
    var message = document.getElementById("messge");
    message.innerHTML = "Here's the message!";
};
```

We're making an error in this code.  
Can you see what we did wrong?

Load the page in your browser, make sure the console is open, and you'll get an error. Can you see what went wrong? We mistyped the id of the `<div>` element, so when your code tries to retrieve that `<div>` element it fails, and the variable `message` is null. And you can't access the `innerHTML` property of null.



When you get an error that causes your code to stop executing like this one does, it's called an exception. JavaScript has a mechanism, called `try/catch`, that you can use to watch for exceptions and catch them when they happen. The idea is that if you can catch one of these exceptions, rather than your code just stopping, you can take an alternative action (try something else, offer the user a different experience, etc.).

### Try/catch

The way you use `try/catch` is like this: you put the code you want to try in the `try` block, and then you write a `catch` block that contains code that will be executed in case anything goes wrong with the code in the `try` block. The `catch` keyword is followed by parentheses that contain a variable name (that acts a lot like a function parameter). If something goes wrong and an exception is caught, the variable will be assigned to a value related to the exception, often an `Error` object. Here's how you use a `try/catch` statement:

```
window.onload = function() {
    try {
        var message = document.getElementById("messge");
        message.innerHTML = "Here's the message!";
    } catch (error) {
        console.log("Error! " + error.message);
    }
};
```

Depending on the error, you could do something much smarter here.

We moved our code into a `try` block.

Here, we're trying to set the `innerHTML` property of `message` (which is null) to a string.

If the code in the `try` block causes an exception, then this line of code is executed. All we're doing is displaying the `message` property of the `error` object in the console. Then execution continues with the line following the `try/catch`.

## #6 Adding event handlers with addEventListener

In this book, we used object properties to assign event handlers to events. For instance, when we wanted to handle the load event, we assigned an event handler to the `window.onload` property. And when we wanted to handle a button click, we assigned an event handler to that button's `onclick` property.

This is a convenient way of assigning event handlers. But sometimes, you might need a more general way of assigning event handlers. For instance, if you want to assign multiple handlers for one event type, you can't do that if you use a property like `onload`. But you can with a method named `addEventListener`:

We call `addEventListener` on `window` to register a handler for the `load` event.

```
window.addEventListener("load", init, false);
function init() {
  // page has loaded
}
```

And we pass it three arguments: the name of the event, "load", as a string...

...a reference to the handler function for the event...

...and a flag indicating if we want to bubble the event up (we'll explain bubble in a moment).

So the `init` function is called when the `load` event happens.

You can assign a second `load` event handler to `window` simply by calling `addEventListener` again, passing a different event handler function reference as the second argument. This is handy if you want to split your initialization code into two separate functions, but remember—you won't know which handler will be called first, so keep that in mind as you're designing your code.

The third argument to `addEventListener` determines if the event is “bubbled up” to parent elements. This doesn't make a difference for the `load` event (because the `window` object is at the top level), but if you have, say, a `<span>` element nested inside a `<div>` element, and you click on the `<span>` but want the `<div>` to receive the event, then you can set `bubble` to `true` instead of `false`.

It's totally fine to mix and match using the event properties, like `onload`, with `addEventListener`. Also, if you add an event handler with `addEventListener`, you can remove it later with `removeEventListener`, like this:

```
window.onload = function() {
  var div = document.getElementById("clickme");
  div.addEventListener("click", handleClick, false);
};

function handleClick(e) {
  var target = e.target;
  alert("You clicked on " + target.id);
  target.removeEventListener("click", handleClick, false);
}
```

We're using the `onload` property to assign the `load` event handler for `window`.

And using `addEventListener` to assign the event handler for the `<div>`'s `click` event.

When you click the `<div>`, we remove the event handler from the `div` with `removeEventListener`.

## Event handling in IE8 and older

We've handled a few different kinds of events in this book—mouse clicks, load events, key presses, and more—and hopefully you've been using a modern browser and the code has worked for you. However, if you are writing a web page that handles events (and what web page doesn't?) and you're concerned that some of the people in your audience may be using versions of Internet Explorer (IE) that are version 8 or older, you need to be aware of an issue with event handling.

Unfortunately, IE handled events differently from other browsers until IE9. You could use properties like `onclick` and `onload` to set event handlers across all browsers, however, the way that older IE browsers handle the event object is different. In addition, if you happen to be using the standardized `addEventListener` method, IE didn't support this method until IE9 and later. Here are the main issues to be aware of:

- ❑ IE8 and older browsers do support most of the “on” properties you can use to assign event handlers.
- ❑ IE8 and older browsers use a method named `attachEvent` instead of the `addEventListener` method.
- ❑ When an event is triggered and your event handler is called, instead of passing an event object to the handler, IE8 and older store the event object in the `window` object.

So, if you want to be sure that your code works across all browsers, including IE8 and older browsers, then you can manage these differences like this:

```
window.onload = function() {
    var div = document.getElementById("clickme");
    if (div.addEventListener) {
        div.addEventListener("click", handleClick, false);
    } else if (div.attachEvent) {
        div.attachEvent("onclick", handleClick);
    }
};

function handleClick(e) {
    var evt = e || window.event;
    var target;
    if (evt.target) {
        target = evt.target;
    } else {
        target = evt.srcElement;
    }
    alert("You clicked on " + target.id);
}
```

IE8 supports the `onload` property for the load event so this is okay.

If you use the `addEventListener` method to add an event handler, you need to check to make sure the method exists...

...and if it doesn't, use the `attachEvent` method instead. Notice `attachEvent` doesn't have a third argument, and uses “`onclick`” for the event name.

If the event object is passed, then you know you're dealing with IE9+ or another browser. Otherwise, you have to get the event object from the window.

If the event object is the modern one, the element that triggered the event will be in the `target` property, like normal. But if this is IE8 or older, this element will be in the `srcElement` property.

## #7 Regular Expressions

You've seen the `RegExp` object in passing in this book—"RegEx" stands for *regular expression*, which is, formally, a grammar for describing patterns in text. For instance, using a regular expression, you could write an expression that matches all text that starts with "t" and ends with "e", with at least one "a" and no more than two "u"s in between.

Regular expressions can get complex fast. In fact, regular expressions can almost seem like an alien language when you first try to read them. But you can get started with simple regular expressions fairly quickly, and if you like them, check out a good reference on the topic.

### The `RegExp` constructor

Let's take a look at a couple of regular expressions. To create a regular expression, pass a search pattern to the `RegExp` constructor, between two slashes, like this:

```
var areaCode = new RegExp(/ [0-9]{3} /);  
var phoneNumber = new RegExp(/^ \d{3}-? \d{4} $/);
```

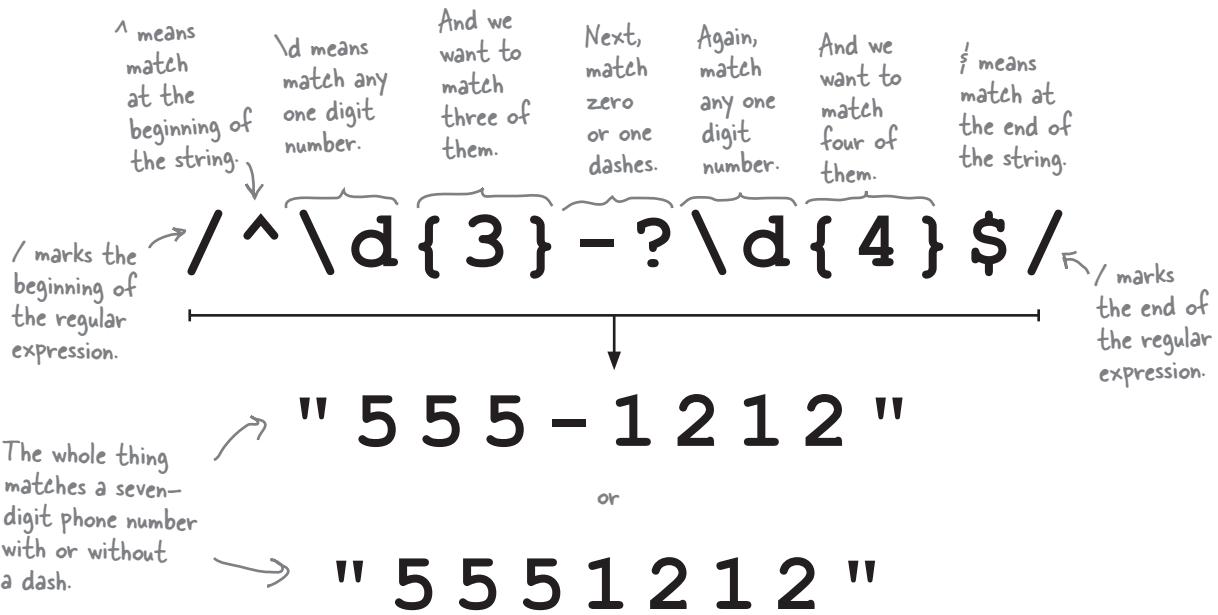
The argument to the `RegExp` constructor is the search pattern. How do you read these two search patterns?  
Like this...  
Remember this from Chair Wars back in Chapter 7?  
This was Amy's winning code.

The key to understanding regular expressions is learning how to read the search patterns. These search patterns are the most complex part of regular expressions, so we'll work through the two examples here, and you'll have to explore the rest on your own.

This says match any of the numbers in the range 0-9. [ ] mean you're specifying a range of letters or numbers.  
/ marks the beginning of the regular expression.  
The whole thing matches any three digit string, like "201" or "503".

/ [ 0 - 9 ] { 3 } /  
And this says that we want to match three of the previous character. In other words we want to match three numbers in the range 0-9.  
/ marks the end of the regular expression.

" 2 0 1 "



## Using a RegExp object

To use a regular expression, you first need a string to search:

```
var amyHome = "555-1212";
```

Then, you match the regular expression to the string by calling the `match` method on the string, and passing the regular expression object as an argument:

```
var result = amyHome.match(phoneNumber);
```

The result is an array containing any parts of the string that matched. If the result is null, then nothing in the string matched the regular expression:

```
var invalid = "5556-1212";
```

```
var result2 = invalid.match(phoneNumber);
```

The value in `result` is ["555-1212"], because in this case, the entire string in the variable `amyHome` matched.

The value in `result2` is null because no part of the string in the variable `invalid` matched our regular expression search pattern.

Once you've got a regular expression, like `phoneNumber`, you can just keep using it to match as many strings as you like.

## #8 Recursion

When you give a function a name it allows you to do something quite interesting: call that function from within the function. We call this *recursion*, or a recursive function call.

Now why would you need such a thing? Well, some problems are inherently recursive. Here's one from mathematics: an algorithm to compute the Fibonacci number series. The Fibonacci number series is:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144... and so on.

To compute a Fibonacci number we start by assuming:

Fibonacci of 0 is 1

Fibonacci of 1 is 1

and then to compute any other number in the series we simply add together the two previous numbers in the series. So:

Fibonacci of 2 is Fibonacci of 1 + Fibonacci of 0 = 2

Fibonacci of 3 is Fibonacci of 2 + Fibonacci of 1 = 3

Fibonacci of 4 is Fibonacci of 3 + Fibonacci of 2 = 5

and so on... The algorithm to compute Fibonacci numbers is inherently recursive because you compute the next number using the results of the previous two Fibonacci numbers.

We can make a recursive function to compute Fibonacci numbers like this: to compute the Fibonacci of the number n, we call the Fibonacci function with the argument n-1 and call the Fibonacci function with the argument n-2, and then add the results together.

Let's do that in code. We'll start by handling the cases of 0 and 1:

```
function fibonacci(n) {  
    if (n === 0) return 1; ← We start with a function that accepts n,  
    if (n === 1) return 1; ← the number in the series we're after.  
}  
  
Then we know that if the number is either 0  
or 1, we return 1. This is known as the base  
case of the function, because it doesn't make  
any recursive calls.
```

These are the *base cases*—that is, the cases that don't rely on previous Fibonacci numbers to compute—and it is usually good to write them first. From there you can think like this: “To compute a Fibonacci number n, I return the result of adding the Fibonacci of n-1 and the Fibonacci of n-2.”

Let's do that...

```
function fibonacci(n) {
    if (n === 0) return 1;
    if (n === 1) return 1;
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

Now if n isn't 0 or 1, we just need to compute the Fibonacci by adding together the Fibonacci of n-1 and n-2.

This looks a little like magic if you've never seen recursion before, but this does compute Fibonacci numbers. Let's clean the code up a little, and test it:

```
function fibonacci(n) {           ← Same code, just written a little better.
    if (n === 0 || n === 1) {
        return 1;
    } else {
        return (fibonacci(n-1) + fibonacci(n-2));
    }
}

for (var i = 0; i < 10; i++) {
    console.log("The fibonacci of " + i + " is " + fibonacci(i));
}
```

← And some test code.



**Watch it!**

**Make sure you have a base case.**  
If recursive code never reaches a base case where the computation ends, it will run forever, like an infinite loop. In other words, the function will continue calling itself over and over, consuming resources until your browser can't take it anymore. So if you write recursive code and your page isn't responding, figure out how to make sure you're getting to the base case.

JavaScript console

```
The fibonacci of 0 is 1
The fibonacci of 1 is 1
The fibonacci of 2 is 2
The fibonacci of 3 is 3
The fibonacci of 4 is 5
The fibonacci of 5 is 8
The fibonacci of 6 is 13
The fibonacci of 7 is 21
The fibonacci of 8 is 34
The fibonacci of 9 is 55
```

## #9 JSON

Not only is JavaScript the programming language of the Web, it's becoming a common interchange format for storing and transmitting objects. JSON is an acronym for "JavaScript Object Notation" and is a format that allows you to represent a JavaScript object as a string—a string that can be stored and transmitted:

A JSON string.

```
var fidoString = '{ "name": "Fido", "breed": "Mixed", "weight": 38 }';
```

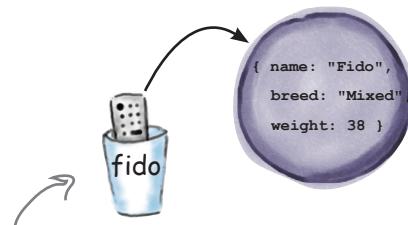
Notice that we're using single quotes around the JSON string. We have to use single quotes because the string contains double quotes, so JavaScript will get confused otherwise. This way, JavaScript knows this is one long string that contains other strings.

Look familiar? It should. This string looks a lot like the `fido` object we worked with earlier in the book...

Now, the cool thing about JSON is we can take strings like this and turn them into objects. The way we do it is with a couple of methods supplied by JavaScript JSON object: `JSON.parse` and `JSON.stringify`. We'll use the `parse` method to parse the `fidoString` above and turn it into a real dog (well, a JavaScript object anyway):

```
var fido = JSON.parse(fidoString);
```

We call the `parse` method of the `JSON` object, passing the string above, and we get back...



...a real JavaScript object. We store the reference to the object in the variable `fido`.

Notice that we're using the `JSON` object here. `JSON` is both the name of a string format and an object in JavaScript.

And you can go the other way, too. If you have an object, `fido`, and you want to turn it into a string, you just call the `JSON.stringify` method, like this:

```
var fido = {  
  name: "Fido",  
  breed: "Mixed",  
  weight: 38  
};  
var fidoString = JSON.stringify(fido);
```

Here, we're taking a JavaScript object...

...and turning it into a string.

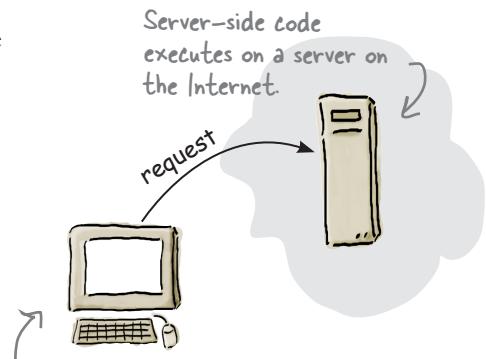
Note that the JSON format doesn't work with methods (so you can't include, say, a `bark` method in your JSON string), but it does work with all the primitive types, as well as objects and arrays.

## #10 Server-side JavaScript

In this book we've focused on the browser and client-side programming, but there's a whole world of server-side programming where you can now use your JavaScript skills. Server-side programming is typically required for the kinds of web and cloud services you use on the Internet. If you want to create Webville Taco's new online order system, or you think the next big idea is the anti-social network, you'll need to write code that lives and runs in the cloud (on a server on the Internet).

Node.js is the JavaScript server-side technology of choice these days, and it includes its own runtime environment and set of libraries (in the same way client-side JavaScript uses the browser's libraries). And like the browser, Node.js runs JavaScript in a single-threaded model where only one thread of execution can happen at a time. This leads to a programming model similar to the browser that is based on asynchronous events and an event loop.

As an example, the method below starts up a web server listening for incoming web requests. It takes a handler that is responsible for handling those requests when they occur. Notice that the convention for setting up the event handler for incoming requests is to pass an anonymous function to the `createServer` method.



Client-side code executes on the client—that is, on your computer.

The `http.createServer` Node.js library method takes a handler in the form of an anonymous function as an argument.

```
http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

The anonymous function is responsible for taking care of requests. It responds to incoming requests by sending back the string "Hello World".

Of course, there is much more to explain and to work through to understand how Node.js works. But, given your knowledge of objects and functions you are well positioned to take this on. Also, explaining Node.js requires at least an entire book of its own, but you'll also find many online tutorials, articles and demonstrations at <http://nodejs.org>.



# Colophon



## All interior layouts were designed by Eric Freeman and Elisabeth Robson.

Kathy Sierra and Bert Bates created the look & feel of the Head First series. The book was produced using Adobe InDesign CS5.5 and Adobe Photoshop CS5.5. The book was typeset using Uncle Stinky, Mister Frisky (you think we're kidding), Ann Satellite, Baskerville, Comic Sans, Myriad Pro, Skippy Sharp, Savoye LET, Jokerman LET, Courier New and Woodrow typefaces.

Interior design and production was done exclusively on Apple Macintoshes—a Mac Pro, an iMac, a Macbook Pro, and two MacBook Airs to be precise.

Writing locations were primarily Bainbridge Island, Washington; Austin, Texas; Port of Ness, Scotland; Seaside, Florida.

Sonic environment during writing included BT, Daft Punk, Muse, The Fixx, Depeche Mode, Adam & the Ants, Men without Hats, Sleep Research Lab, Dousk, Uh Huh Her, Art of Noise, deadmau5 & Kaskade, David Bowie, Cheap Trick, The Who, Blank & Jones, Chris Isaak, Roy Orbison, Elvis, John Lennon, George Harrison, Amy Macdonald, Schiller, Sia, Sigur Ros, Tom Waits, OMD, Phillip Glass, Muse, Eno, Krishna Das, Mike Oldfield, Devo, Steve Roach, Beyman Brothers, Harry and the Potters, and the soundtracks for Frozen, Harry Potter, Back to the Future and Pleasantville.



# Symbols

\$ (dollar sign)  
beginning JavaScript variable names 13  
function in jQuery 624

0 (zero), as falsey value 292

&& (AND) operator 55, 62–63, 74

\* (asterisk) operator, as multiplication arithmetic operator 15, 286

: (colon), separating property name and property value 179

, (comma), separating object properties 177, 179

{ } (curly braces)  
enclosing object properties 177, 179  
in body of function 97  
in code block 17  
matching in code 59  
using with object literals 177, 522

. (dot notation)  
accessing object properties 181, 209, 230  
using with reference variables 186  
using with this object 202

= (equal sign) operator, assigning values to variables using 11, 16, 275

== (equality) operator  
as comparison operator 16, 55, 275–285, 311, 459  
vs. === operator 289

=== (strict equality) operator  
as comparison operator 55, 280–285, 311  
vs. == operator 289

// (forward slashes), beginning comments in JavaScript 13

/ (forward slash) operator, as division arithmetic operator 15, 286

> (greater than) operator 16, 55, 459

>= (greater than or equal to) operator 16, 55

# Index

< (less than) operator 55, 459

<= (less than or equal to) operator 55

- (minus sign) operator  
as unary operator 287  
using as arithmetic operator with string and number 286–287, 312

!= (not equal to) operator 16, 55

! (NOT) operator 55

|| (OR) operator 54, 55, 62–63, 74

() (parentheses)  
in calling functions 68, 430, 439  
in parameters 97

+ (plus sign)  
as arithmetic operator 286–287, 312  
in concatenating strings 15, 133, 142, 354

-- (post-decrement operator) 146–147

++ (post-increment operator) 146–147

“ “ (quotation marks, double)  
surrounding character strings in JavaScript 13  
using around property name 179

; (semicolon), ending statements in JavaScript 11, 13

[] (square brackets)  
accessing properties using 209  
in arrays 127, 129, 550

!== (strict not equal to) operator 281

\_ (underscore), beginning JavaScript variable names 13

## A

action attributes 328

activities, about doing xxxiii

addEventListener method 630

alert function  
communicating with users using 25–26, 42, 46  
determining hits and misses in simplified Battleship game 59–60, 76

- alt attribute 256
  - AND (`&&`) operator 55, 62–63, 74
  - anonymous functions
    - about , 475–476, 482, xx–xxi
    - accessing properties using 509, 518
    - assigning to method in constructors 530–532, 557
    - creating 477–478, 512–513
    - making code tighter 479
    - passing functions to functions 482, 486, 514
    - readability of 480
  - API-specific events 413
  - applications. *See also* Battleship game, advanced; Battleship game, simplified
    - coding JavaScript 29–35
    - creating interactive 319
    - JavaScript in 5
    - web pages as 9
  - arguments, function
    - about 85
    - identifying 91, 105, 119, 120
    - mixing up order of 97
    - objects as 192
    - passing 88–89, 92–94
    - using pass-by-value 92–93
    - vs. parameters 90
  - arguments object 628
  - array constructor object 549–551
  - array literal syntax 550
  - arrays
    - about , xiii–xv
    - abstracting code into functions 157–162
    - accessing item in 129
    - arranging code exercise 139, 168
    - Auto-O-Matic app (example) 195–197
    - creating
      - empty 151
      - most effective bubble solution code 164–165, 171
      - with values 128
    - Cubicle Conversation on 148–149, 170
    - declaring variables in 152
    - empty 134, 153–154, 365, 367, 549–550
    - for determining hits in advanced Battleship game 344–345
    - for hits and ship locations in advanced Battleship game 338
  - for loop in 140–142, 144–145, 147, 169
  - indices in 129, 134, 152, 163
  - initializing counter 140
  - iterating over 138, 140
  - length property in 130
  - literals in 151–152
  - number of levels deep to nest objects 348
  - number of things in 134
  - order of items in 134
  - Phrase-o-Matic app (example) 131–133
  - populating playlist items using 253, 262
  - reusing code in 156
  - sparse 152
  - test drive 143, 147, 155
  - undefined values and 268
  - updating value in 129
  - value types in 134
  - while loop in 17–21
- array sort method 457–463, 472–473
- asterisk (\*) operator, as multiplication arithmetic operator 15, 286
- asynchronous coding
  - about events 383
  - alt attribute 256
  - event handlers
    - about 383
    - adding using addEventListener 630
    - assigning to properties 407
    - callbacks and 250
    - creating 385–386
    - in advanced Battleship game 358–359, 361
    - kinds of 252
    - onload 249
    - timerHandler 407
    - using setInterval function 410, 425
    - using setTimeout function 407–413
  - Event object in DOM 399–402, 423
  - events 404
  - exercise on notification of events 382, 421
  - interview with browser about events 403
  - reacting to events
    - about 387
    - adding images to image guessing game 393–397
    - assigning click handlers 396–397
    - assigning handler to onclick property 390–391, 393–398

- creating image guessing game 388–392, 411–414, 422, 426
  - exercise on 415, 427
  - asynchronous coding event handlers
    - event handlers
      - creating closure with 503–507
  - attributes
    - action 328
    - alt 256
    - class 255
    - getting with getAttribute 256
    - not in element 256
    - setting with code 254
    - setting with setAttribute 255
  - Auto-O-Matic app (example) 195–197
- ## B
- background CSS property 326
  - background image
    - cells in 322
    - creating HTML page 320–321
  - Battleship game, advanced
    - about , xvii–xix
    - cheating at 369
    - controller object
      - about 329
      - counting guesses and firing shot 355–356
      - determining end of game 356
      - developing parseGuess function 351–354
      - implementing 349–357
      - passing input to 360
      - processing player’s guess 350–354
      - testing 357
    - creating HTML page
      - about 320
      - adding CSS 324–326
      - background 320–321
      - placing hits and misses 326
      - table 322
      - using hit and miss classes 327, 375
    - designing game 329–330, 376
    - event handler to fire button 359
    - generating random locations for ships 362–368, 380
    - getting players guess 358–361
  - keeping track of ships 337
  - model object
    - about 329, 336
    - determining hits 344–345
    - determining if ship is sunk 346
    - fire method in 342
    - implementing 341–348
    - interacting with view 336
    - looking for hits 343
    - notifying view of hits or misses 347
    - parseGuess function asking about size of board 353
    - representation of ships in 338–340, 378–379
    - representing state of ships in 337, 377
    - testing 348
  - QA in 370–371
  - starting location of ships 366
  - testing 370
  - toolkit for building 319
  - view object
    - about 329
    - implementing 329, 331–334
    - interacting with model 336
    - notifying of hits or misses 347
    - testing 335
  - Battleship game, simplified
    - about 44
    - adding hit detection code 56–57, 77
    - adding HTML to link code 45–46
    - assigning random ship locations 66–68
    - checking if ship sank 57
    - checking users guess 54
    - designing 45–46
    - displaying stats to user 58
    - doing quality assurance 61, 69–70
    - finding errors in code 68
    - procedural design in 329
    - pseudocode
      - determining hits and misses 59–60, 76
      - determining variables using 50
      - implementing loop using 51
      - translating into JavaScript code 52
      - working through 47–48, 73
    - behavior and state, in objects 210–212, 226
    - block of code 17, 23

<body> element  
about 4  
adding code on HTML page to 32  
replacing element content with innerHTML property 245

body, function  
declaring variables within 97  
parameters in 84–85

BOM (Browser Object Model) 627

boolean expressions  
true and false values 13, 15–16, 291–293, 313  
using to make decisions 22

boolean operators  
being verbose in specifying conditionals using 65, 74  
comparison vs. logical 71  
guide to 55  
writing complex statements using 62–64, 74–75

booleans  
about 11, 23  
as objects 296  
as primitive types 266  
using == operator with other types 277–278, 281

Browser Object Model (BOM) 627

browsers  
conditional statements handled by 434  
events 404  
executing code 433  
function declarations handled by 431–433, 436, 465, 483  
function expressions handled by 434–436, 465  
handling events 383, 403. *See also* handling events  
loading and executing JavaScript in 3  
loading code into 31  
objects provided by 214  
opening console 28  
parsing HTML and building DOM 236, 261  
recommended xxvi  
running JavaScript in xxxii  
setting up Event object in IE8 and older 399  
setting up event objects in IE8 and older 631  
tailoring size of window to web page 627

browser wars 6

built-in functions 71, 91, 119

built-in objects 548–551, 608–610

# C

callbacks. *See also* event handlers 250

calling (invoking) functions  
about 85–86, 492–493  
browsers handling function declarations and expressions 431–437  
exercise for 117  
parentheses in 68, 430, 439  
recursive functions 634–635  
variables referring to functions in 436  
with arguments 90, 430

call method 603–604

camel case in multi-word variable names 13

Cascading Style Sheets (CSS). *See* CSS (Cascading Style Sheets)

case sensitivity in JavaScript 12

catch/try, handling exceptions using 629

cells, in table of advanced Battleship game 322

chaining 345, 348

chain of prototypes  
inheritance in 592  
setting up 591

characters, in JavaScript 297

character strings, quoting in JavaScript 13

charAt method 297

classes  
class attribute 255  
hit and miss classes for advanced Battleship game 326–327, 375  
labeling set of elements with 243

classic object-oriented programming 564

clear code, writing 12

cliche method 610

click events 407, 419, 624

click handlers, assigning 396–397

close() method 627

closures  
about 493, 495–497  
actual environment in 502

- creating with event handler 503–507
- exercise for creating 500, 514–515
- passing function expression as argument to create 501
- using to implement magic counters 498–499
- Coda, as HTML editor 31
- code. *See also* Battleship game, advanced; Battleship game, simplified; writing JavaScript
  - analyzing 80, 116
  - browser executing 433
  - compiling 5
  - duplicating code in objects 208
  - finding errors in
    - exercise 14, 39, 203, 224
    - using console 68
    - using console.log function 25, 27
  - for JavaScript application 29–35
  - functions and reusing 83–88
  - guide to hygiene of 111
  - loading into browser 31
  - moving JavaScript from HTML page to file 33–34
  - recursive 635
  - refactoring 156, 159
  - reusing , 71, 79, xii
- code block 17, 23
- code file
  - linking external file from HTML page <body> element 32
  - linking from HTML page <head> element 32
- collision method 364, 368
- colon (:), separating property name and property value 179
- comma (,), separating object properties 177, 179
- comments in JavaScript 13
- comparison operators 55, 71
- compiling code 5
- concatenating strings 15, 133, 142, 286–287, 312, 354
- concat, method 300
- conditionals
  - as variable or string 23
  - being verbose in specifying 65, 74
  - combining using boolean operators 62–63, 74
  - handled by browsers 434
  - test in arrays 140
- using boolean expressions to make decisions with 22
- using in while statement 17–21
- confirm method 627
- console
  - about 214
  - cheating at advanced Battleship game using 370
  - finding errors using 68
  - opening browser 28
- console.log function
  - communicating with users using 25–27, 42
  - displaying object in console using 185, 608
- constructor object, array 549–551
- constructors, object
  - about , 521, 525, xxi
  - creating 525–526, 536–538, 555, 558, 566–567
  - creating objects
    - by convention 523
    - with object literals 522
  - finding errors in code exercise 529, 556
  - independent properties of 546–547
  - naming 532
  - parameters names in 532
  - putting methods into 530–532, 557
  - real world 548
  - understanding object instances 543–545, 560
  - updating constructor
    - chain of prototypes in 591–593, 606, 619
    - cleaning up code 600–605
    - creating prototype that inherits from another prototype 594–599, 620
    - design for 588–590
  - using 527, 555
  - using new keyword with 528, 532–535, 543, 548
  - vs. literals 532, 539–542, 559
  - workings of 528
- controller object, advanced Battleship game
  - about 329
  - counting guesses and firing shot 355–356
  - determining end of game 356
  - developing parseGuess function 351–354
  - implementing 349–357
  - passing input to 360
  - processing player's guess 350–354
  - testing 357
- counter, initializing, in array 140

CSS (Cascading Style Sheets)  
about 2  
background property 326  
creating interactive web page using 319  
marking up text with 10  
positioning 324–326, 328  
writing 254, 263

curly braces ({} )  
enclosing object properties 177, 179  
in body of function 97  
in code block 17  
matching in code 59  
using with object literals 177, 522

## D

Date object 214, 548, 551

declarations, function  
assigning to variables 439  
evaluating code 438, 466  
handled by browser 431–433, 436, 465, 483  
in anonymous functions 483–485  
parsing 431  
vs. function expressions 431, 437

defining functions, with parameters 90

delete keyword 184

designing  
simplified Battleship game game 45–46

designing, advanced Battleship game game 329–330, 376

diagrams, object 565, 570

displayHit method 331–334

displayMessage method 331–334

displayMiss method 331–334

<div> elements  
giving unique id 243  
positioning using CSS 324–325

document object  
about 214, 240  
getElementById method  
accessing images using 388–392  
as case sensitive 230  
as document object 237  
getting element with 231, 240, 247–248

getting reference to fire button in advanced Battleship game 359  
passing id that does not exist 245  
returning null 256, 270–271  
using to locate element and change its content 238–239

getElementsByClassName method 245, 626  
getElementsByName method 626  
getElementsByTagName method 245, 397  
in DOM 235, 626  
write function, communicating with users using 25–26, 42

dollar sign (\$)  
beginning JavaScript variable names 13  
function in jQuery 624

DOM (Document Object Model)  
about , xv  
accessing images in 389–390  
changing 244  
communicating with users using 25–26, 42  
creating 234–235  
creating, adding, or removing elements getting 258  
document object in 235, 626  
DOM structure as tree 235  
elements grabbed from 241  
Event object in 399–402, 423  
events 413  
getting element by id from 245  
getting elements from, using jQuery 624  
interaction of JavaScript with 233  
NodeList in 397  
parsing HTML and building 236, 261  
setting attributes  
with code 254  
with setAttribute method 255, 333  
updating 247–248  
with secret message 246

“Don’t Repeat Yourself” (DRY) 603

dot notation (.)  
accessing object properties 181, 209, 230  
using with reference variables 186  
using with this object 202

do while loop 364, 373

dragstart event 419

Dreamweaver, as HTML editor 31

drop event 419  
 DRY (“Don’t Repeat Yourself”) 603

# E

ECMAScript 5 6

element objects  
 about 231, 245  
 returning list of 397

elements. *See also* getElementById method  
 accessing with getElementById 238, 240  
 attribute does not exist in 256  
 classes used to label set of 243  
 getting by id from DOM 245  
 getting, creating, adding, or removing 258  
 grabbed from DOM 241  
 identifying elements 243  
 labeling with classes set of 243  
 target set to 398–399, 401  
 using CSS to position 324–326

else if statements  
 making decisions using 22–23  
 writing 56

else statements, using as catch-all with if/else statements 23

empty arrays 134, 153–154, 365, 367, 549–550

encapsulation 200

equal sign (=) operator, assigning values to variables using 11, 16, 275

equality (==) operator  
 as comparison operator 16, 55, 275–285, 311, 459  
 vs. === operator 289

strict equality (===) operator  
 as comparison operator 55, 280–285, 311  
 vs. == operator 289

Error object 551, 629

errors in code, finding  
 exercise 14, 39, 203, 224  
 finding errors using console 68  
 handling exceptions 629  
 using console.log function 25, 27

event handlers  
 about 384  
 adding using addEventListener 630

assigning to properties 407  
 asynchronous coding  
 assigning to properties 407  
 callbacks and 250  
 creating 385–386  
 in advanced Battleship game 358–359, 361  
 kinds of 252  
 onload 249  
 timerHandler 407  
 using setInterval function 410, 425  
 using setTimeout function 407–413  
 callbacks and 250  
 creating 385–386  
 creating closure with 503–507  
 event objects 399–402, 423  
 in advanced Battleship game 358–359, 361  
 kinds of 252  
 onload 249  
 timerHandler 407  
 using setInterval function 410, 425  
 using setTimeout function 407–413

Event object  
 event handlers and 399–402, 423  
 for properties 401  
 setting up in IE8 and older browsers 399, 631

event objects  
 setting up in IE8 and older browsers 631

events  
 about 383, 419  
 API-specific 413  
 click 407, 419, 624  
 creating game that reacts to 388–392  
 DOM 413  
 dragstart 419  
 drop 419  
 exercise on notification of 382, 421  
 exercise on reacting to events 415, 427  
 image game and  
 adding images to 393–397  
 assigning click handlers 396–397  
 assigning handler to onclick property 390–391, 393–398  
 creating 411–414, 422, 426  
 interview with browser about events 403

I/O 413

keypress 419

list of 419  
load 419. *See also* load events  
mousemove 405–406, 419, 424  
mouseout 419  
mouseover 419  
onclick property 390–391, 393–398  
pause 419  
queues and 404  
reacting to 387  
resize 419  
timer 407–409  
touchend 419  
touchstart 419  
types of 407  
unload 419  
using setInterval function 410, 425

example files, downloading xxxiii

exceptions, handling 629

exercises, about doing xxxiii

expressions 15–16, 40

expressions, function

assigning to variables 439  
evaluating code 438, 466  
handled by browser 434–436, 465  
substituting 481  
vs. declarations, function 431, 437

# F

false (boolean value) 13

falsey and truthy values 291–293, 313

fire method, advanced Battleship game 342, 346

first class values, functions as

about 442–443  
extreme JavaScript challenge 486, 508, 517  
passing functions to functions 443–448, 468, 482, 486, 514  
returning functions from 450–456, 470, 472  
substituting function expressions 481  
using array sort method 457–463, 472–473

Flash, creating dynamic web pages using 5

flowchart, for designing program 45

for/in iterator 209

for loop

in adding new ship locations in advanced Battleship game 365, 367  
in guessing location of ships in advanced Battleship game 343  
iterating arrays using 140–142, 169  
redoing 147  
vs. while loop 144

<form> element

action attribute in 328  
adding inputs to 323  
using CSS to position 325

forward slashes (//), beginning JavaScript comments 13

forward slash (/) operator, as division arithmetic operator 15, 286

free variables 495–496, 501, 516

functions

\$ (dollar sign), in jQuery 624  
about , 79, 88, xii  
abstracting code from array for 157–162  
alert  
communicating with users using 25–26, 42, 46  
determining hits and misses in simplified Battleship game 59, 76  
anatomy of 97  
anonymous  
about , 475–476, 482, xx–xxi  
assigning to method in constructors 530–532, 557  
brain twister involving 509, 518  
creating 477–478, 512–513  
making code tighter 479  
passing functions to functions 482, 486, 514  
readability of 480

arguments in

about 85  
identifying 91, 105  
mixing up order of 97  
passing 88–89, 92–94  
using pass-by-value 92–93  
vs. parameters 90

arguments object in 628

as first class values

about 442–443  
extreme JavaScript challenge 487, 508, 517

- passing functions to functions 443–448, 468, 482, 486, 514
- returning functions from 450–456, 470, 472
- substituting function expressions 481
- using array sort method 457–463, 472–473
- as objects 612–613
- assigning to variables 439, 449, 469
- as values 439–440, 467
- body of
  - declaring variables within 97
  - parameters in 84–85
- built-in 71, 91, 119
- calling 85–86, 117
- calling recursively 634–635
- closures and
  - about 493, 495–497
  - actual environment in closures 502
  - creating closure with event handlers 503–507
  - exercise creating 500, 514–515
  - passing function expression as argument to create 501
  - using to implement counters 498–499
- console.log
  - communicating with users using 25, 42
  - displaying object in console using 185
- creating 83–87
- declarations
  - about 430
  - assigning to variables 439
  - evaluating code 438, 466
  - handled by browser 431–433, 436, 465, 483
  - in anonymous functions 483–485
  - parsing 431
  - vs. expressions 431, 437
- defining 132
- exercise identifying 91, 119
- expressions
  - assigning to variables 439
  - evaluating code 438, 466
  - handled by browser 434–436, 465
  - substituting 481
  - vs. declarations 431, 437
- extreme JavaScript challenge 487, 508, 517
- guide to code hygiene 111
- init 249, 251, 359, 361, 369, 389–391
- isNaN 273, 353
- keyword 430
- life span of variables 102
- location in JavaScript 109
- Math.floor 68, 131, 365–367
- Math.random 67–68, 88, 131, 365–367
- naming 97
- nesting 485–486, 488–490, 494
- No Dumb Questions 97, 108
- onload handler 249–250, 359, 389–390
- order in defining 389
- parameters in
  - about 84, 88
  - assigning argument values to 85
  - identifying 91, 105
  - naming 97
  - none used in function 94
  - passing arguments to 89
  - vs. arguments 90
- parameters vs. arguments 90
- parentheses () in calling 68, 430, 439
- passing functions to 409, 443–448, 468, 482, 486, 514
- passing objects to 192–194, 198
- prompt 46, 53, 55
- reference
  - about 430, 476
  - action in function 491, 494
  - assigning 477
  - in calling function 491
  - in passing function argument to another function 486
  - passing 479
  - substituting function expressions and 481
- returning functions from 450–456, 470, 472
- return statement in 95–97
- scope
  - lexical scope of variables 488–490, 494
  - of local and global variables 101
- setInterval 410, 413, 425, 627
- setTimeout 407–413, 410, 480, 501, 627
- standard declaration for 430
- timerHandler 407
- tracing through 96
- variables in
  - declaring inside and outside of functions 97–99
  - declaring local 103, 108
  - default value of 50

lexical scope of 488–490, 494  
local vs. global 99, 106–107  
naming 12–13, 100, 103, 108  
nesting of 486, 488–490, 494  
passing into arguments 89  
reloading page and 108  
scope of local and global 101  
shadowing 103  
var keyword in declaring 97, 99  
vs methods 206  
weird 94  
write 25–26, 42

## G

generateShipLocations method 364, 367, 369  
generateShip method 364–365, 367  
getAttribute method 256, 391  
getDay method 548  
getElementById method  
  accessing images using 388–392  
  as case sensitive 230  
  as document object 237  
  getting element with 231, 240, 247–248  
  getting reference to fire button in advanced Battleship game 359  
  passing id that does not exist 245  
  returning null 256, 270–271  
  using to locate element and change its content 238–239  
getElementsByClassName method 245, 626  
getElementsByName method 626  
getElementsByTagName method 245, 396, 397  
getFullYear method 548  
getSize method 531, 557  
global variables  
  guide to code hygiene 111  
  identifying 105, 120  
  lexical scope of 488–490, 494  
  life cycle of 102  
  nesting affecting 486, 488–490, 494  
  overuse in JavaScript 108  
  overuse of 391  
  reasons for sparing use of 108

scope of 101  
shadowing 104  
vs. local variables 99, 106–107  
greater than ( $>$ ) operator 16, 55, 459  
greater than or equal to ( $\geq$ ) operator 16, 55  
guesses property, advanced Battleship game 349

## H

handleFireButton function 359  
handleKeyPress function 361  
handling events  
  asynchronous coding  
    about 383  
    assigning to properties 407  
    creating 385–386  
    creating closure with 503–507  
    in advanced Battleship game 358–359, 361  
    kinds of 253  
    onload 249  
    timerHandler 407  
    using setInterval function 410, 425  
    using setTimeout function 407–413  
  event handlers  
    about 383  
    adding using addEventListener 630  
    assigning to properties 407  
    callbacks and 250  
    creating 385–386  
    creating closure with 503–507  
    in advanced Battleship game 358–359, 361  
    kinds of 252  
    onload 249  
    timerHandler 407  
    using setInterval function 410, 425  
    using setTimeout function 407–413  
  event objects 399–402, 423  
  exercise on notification of events 382, 383, 421  
  interview with browser about events 403  
  reacting to events  
    about 387  
    adding images to image guessing game 393–397  
    assigning click handlers 396–397  
    assigning handler to onclick property 390–391, 393–398

creating image guessing game 388–392, 411–414, 422, 426  
 exercise on 415, 427  
 using setInterval function 410, 425  
 using setTimeout function 407–413

hasOwnProperty method 586, 607

<head> element  
 about 4  
 adding code on HTML page to 32  
 linking code on HTML page file from 32

Head First HTML5 Programming 413, 613, 626

Head First HTML and CSS 320, 328

history method 627

hits, in advanced Battleship game  
 classes for misses and 326–327, 375  
 determining 344–345  
 determining if ship is sunk 346  
 looking for 343  
 notifying view of 347

HTML  
 about 2  
 creating interactive web page using  
 about 319–320  
 adding CSS 324–326  
 background 320–321  
 placing hits and misses 326  
 player interaction 322  
 table 322  
 using hit and miss classes 327, 375

editors 31

identifying elements with ids 243

linking code in simplified Battleship game 49

marking up text with 10

parsing and building DOM from 236, 261

HTML5  
 standard doctype 4  
 using string of numbers for id attributes in 328  
 using with JavaScript 5

<html> element 4

HTML page  
 JavaScript interacting with 233  
 loading code into 32  
 with code in external file 230–232

HTML wrapper for examples xxxiv

**I**

id attributes  
 accessing elements by 238  
 identifying elements with 243  
 using string of numbers for 328

identity equality operator 284. *See also* strict equality (==) operator

IE8 (Internet Explorer 8)  
 setting up event objects in browsers older than 399, 631

if/else statements  
 making decisions using 23  
 writing 56

if statement, making decisions using 22

image guessing game  
 adding images to  
 about 393–397  
 assigning click handlers 396–397  
 assigning handler to onclick property 390–391, 393–398  
 creating 388–392, 411–414, 422, 426

indexOf method 298, 343, 345, 352, 368, 610

indices, in arrays 129, 134, 152, 163

infinite loop 55

Infinity (-infinity), JavaScript 274

inheritance  
 extending built-in object 610  
 in chain of prototypes 592–593, 619  
 in objects 569–572  
 of prototype from another prototype 594–599, 606, 620  
 overriding built-in behavior 608–609

init function 249, 251, 359, 361, 369, 389–391, 630

innerHeight property 627

innerHTML property  
 about 245  
 changing DOM using 244  
 replacing element content in <body> element with 245  
 setting text using 331  
 using to change element content 239, 241–242, 247–248

innerWidth property 627  
<input> elements  
  adding to <form> 323  
  using CSS to style 325  
instanceof operator 305, 315, 543, 547  
interface transformations on elements, using jQuery 625  
Internet Explorer 8 (IE8)  
  setting up event objects in browsers older than 399, 631  
interpreted languages 5  
in variables 186–187  
invoking (calling) functions  
  about 85–86, 117, 492–493  
  browsers handling function declarations and expressions 431–437  
  exercise for 117  
  parentheses () in 68, 430, 439  
  recursive functions 634–635  
  variables referring to functions in 436  
  with arguments 90, 430  
I/O events 413  
isNaN function 273, 353  
isSunk method, advanced Battleship game 346

## J

Java  
  introduction of 6  
  vs. JavaScript 5  
Java, classic object-oriented programming 564  
JavaScript  
  about , 1–2, x–xi  
  API-specific events 413  
  arrays 127, 132–134  
  case insensitivity 12  
  getElementById as wormhole to 237  
  getting on web page 4–5  
  grand unified theory of 612  
  handling exceptions 629  
  history of 6  
  importance of 5  
  in applications 5  
  Infinity in 274  
  interacting with DOM 233

interaction with web page through DOM 237  
in web pages 2–3  
jQuery 624  
learning 9  
node.js library 637  
objects provided by 214  
reserved words 12  
server-side 637  
syntax 13–14, 39  
thinking about xxix  
values in 272–274, 281, 292  
vs. Java 5  
writing. *See* writing JavaScript

JavaScript: The Definitive Guide 296

jQuery  
  about 624–625  
  online documentation and tutorials 625

JScript 6

JSON object 214, 636

## K

keypress event 419  
keywords 12

## L

lastIndexOf, method 300  
learning  
  JavaScript 9  
  tips for xxxi  
learning principles xxviii  
length property 297, 301, 303  
length property, in arrays 130  
less than (<) operator 55, 459  
less than or equal to (≤) operator 55  
lexical scope of variables 488, 494  
life cycle of variables 102  
list  
  adding songs to playlist with JavaScript (example) 253, 262  
  NodeList 397  
  returning element objects in 397

listener 384. *See also* event handlers

list of event 419

lists

- adding songs to playlist with JavaScript (example) 253, 262
- finding all child `<li>` elements of element with id of playlist, using JQuery 625

literals

- creating objects with 522
- vs. constructors 532, 539–542, 559

literals, in arrays 151–152

LiveScript 6

load event

- about 419
- onload property
  - assigning handler to 385–386, 389–390, 410
  - in anonymous functions 476–477
  - setting to function 249, 359, 392, 394–395, 406–407, 422

local variables

- declaring 103, 108
- guide to code hygiene 111
- identifying 105, 120
- in methods 206
- lexical scope of 488–489, 494
- life cycle of 102
- nesting affecting 486, 488–490, 494
- scope of 101
- shadowing global variable 104
- vs. global variables 99, 106–107

location method 627

logical operators 55, 71

loop

- do while 364, 373
- for
  - in adding new ship locations in advanced Battleship game 365, 367
  - in guessing location of ships in advanced Battleship game 343
  - iterating arrays using 140–142, 169
  - redoing 147
  - vs. while loop 144–145
- implementing using pseudocode 51
- infinite 55

translating pseudocode into JavaScript code 52

using inner and outer 368

while

- about 17
- executing code using 18–21
- vs. for loop 144–145

## M

match method 300, 633

Math.floor function 68, 131, 365–367

Math object 214, 551

Math.random function 67–68, 88, 131, 365–367

metacognition xxix

methods

- about 198
- adding to prototypes 581–582, 584
- available in JavaScript 296
- designing advanced Battleship game 330–331, 376
- duplicating 568
- extending String prototype with 610–611, 621
- in chain of prototypes 592–593, 619
- putting in constructors 530–532, 557
- tour of string 297–300
- using this object in 202, 204–206, 219
- vs. functions 206
- writing 199–200, 207, 225

Microsoft, introduction of JScript 6

Mighty Gumball, Inc. 135–136, 148, 150

minus sign (-) operator

- as unary operator 287
- using as arithmetic operator with string and number 286–287, 312

misses, in advanced Battleship game

- classes for hits and 326–327, 375
- notifying view of 347

model object, advanced Battleship game

- about 329, 336
- determining hits 344–345
- determining if ship is sunk 346
- fire method in 342–343, 346
- implementing 341–348
- interacting with view 336
- keeping track of ships 337

notifying view of hits or misses 347  
parseGuess function asking about size of board 353  
representation of ships in 338–340, 378–379  
representing state of ships in 337, 377  
testing 348  
mousemove event 405–406, 419, 424  
mouseout event 419  
mouseover event 419

## N

naming  
constructors 532  
functions 97  
local and global variables with same name 104  
properties 179  
variables 12–13, 100, 103, 108  
NaN (Not a Number) values 272–274, 281, 292  
nesting functions 485–486, 488–490, 494  
Netscape 6  
new keyword, using with constructors 528, 532–535, 543, 548  
node.js library 637  
NodeList 397  
Notepad (Windows), as text editor 31  
not equal to (!=) operator 16, 55  
NOT (!) operator 55  
null 55, 256, 270–271, 274, 278, 292  
numbers  
as objects 296  
as primitive types 266  
comparing to strings 275–277, 281  
NaN (Not a Number) values and 273–274, 281, 292  
prompt function returning strings for 55  
using + sign with strings and 286–287

## O

Object, as object  
about 551, 607  
as prototype 607  
overriding built-in behavior 608–609

object constructors  
about , 521, 525, xxi  
creating 525–526, 536–538, 555, 558, 566–567  
creating objects, by convention 523  
finding errors in code exercise 529, 556  
independent properties of 546–547  
naming 532  
parameters names in 532  
putting methods into 530–532, 557  
real world 548  
understanding object instances 543–545, 560  
updating constructor  
chain of prototypes in 591–593, 606, 619  
cleaning up code 600–605  
creating prototype that inherits from another prototype 594–599, 620  
design for 588–590  
using 527, 555  
using new keyword with 528, 532–535, 543, 548  
vs. literals 532, 539–542, 559  
workings of 528  
object instances 543–545, 560  
object models, building 3  
object-oriented programming 180, 564  
object prototype model  
about , xxii–xxiii  
better living through objects 612–613  
chain of prototypes  
inheritance in 592–593, 619  
setting up 591  
determining if properties are in instance or in prototype 586–587, 592, 618  
duplicating methods and 568  
extending String prototype with method 610–611, 621  
implementing code 579, 583, 616–617  
inheritance  
extending built-in object 610  
in chain of prototypes 592–593, 619  
in objects 569–572  
of prototype from another prototype 594–599, 606, 620  
overriding built-in behavior 608–609  
Object  
about 607

- as prototype 607
- overriding built-in behavior 608–609
- object diagram exercise 574, 615
- prototypes in
  - about 569
  - adding methods to 581–582, 584
  - changing properties in 582, 585
  - dynamic 582
  - getting 575
  - overriding 573, 578
  - setting up 575
- updating constructor
  - chain of prototypes in 591–593, 606, 619
  - cleaning up code 600–605
  - creating prototype that inherits from another prototype 594–599, 620
  - design for 588–590
- object references 187
- objects
  - about , 173–174, 215, xiv–xv
  - adding behavior to 198–201
  - arguments 628
  - array 549–551
  - array constructor 549–551
  - Auto-O-Matic app (example) 195–197
  - behavior 568
  - better living through 612–613
  - booleans as 296
  - built-in 548–551, 608
  - callback 250
  - comparing 459
  - console 214
  - controller (advanced Battleship game)
    - about 329
    - counting guesses and firing shot 355–356
    - determining end of game 356
    - developing parseGuess function 351–354
    - implementing 349–357
    - passing input to 360
    - processing player's guess 350–354
    - testing 357
  - creating 177–179, 183, 185, 188–191, 220–222
  - creating with object literals 522, 532, 539–542, 559
  - Date 214, 548
  - diagramming 565, 570
  - document
    - about 214, 240
    - getElementById method as 238. *See also* getElementById method
    - in DOM 235, 626
    - write function, communicating with users using 25–26, 42
  - duplicating code in 208
  - element 231
  - equality of 288–290, 314
  - Error 551
  - Event
    - event handlers and 399–402, 423
    - for properties 401
    - setting up in IE8 and older browsers 399, 631
  - finding errors in code (exercise) 203
  - functions as 612–613
  - inheriting 569–572
  - iterating through properties of 209
  - JavaScript provided 214
  - JSON 214
  - Math 214, 551
  - methods 198
  - methods vs. functions 206
  - model (advanced Battleship game). *See* model object, advanced Battleship game
  - models, building 3
  - No Dumb Questions 185
  - number of levels deep in arrays to nest 348
  - numbers as 296
  - passing to functions 192, 192–194, 198
  - properties in
    - about 175
    - accessing 181
    - adding new 182
    - adding or deleting at any time 184
    - changing 182
    - computing with 182
    - iterating through 209
    - undefined values and 268
    - values of 176, 219
  - putting methods into, by convention 523
  - RegExp 214, 551, 632–633
  - rules of road for creating 179
  - state and behavior in 210–212, 226
  - strings as primitives and 294–296

this  
  in constructors 528–529, 532  
  prototypes and 580  
  using 202  
    using with objects 204–206, 219  
undefined values and 268  
understanding object instances 543–545, 560  
using null 270–271, 274  
using this 202, 204–206, 219  
variable declaration for 177  
in variables 186  
vs. primitives 187  
window  
  about 214  
  creating onload event handler for 249, 402  
  history method 627  
  innerHeight property 627  
  innerWidth property 627  
  location method 627  
  methods and properties available for 627  
  onload property. *See* onload property  
  onresize property 415, 427  
  print method 627  
  setInterval method 627  
  setTimeout method 407–413, 627  
  support of BOM in 627  
onclick property 390–391, 393–398, 504–507, 630  
onload event handler 249, 359, 402  
onload handler function 249–250, 359, 389–390  
onload property  
  assigning handler to 385–386, 389–390, 410, 630  
  in anonymous functions 476–477  
  setting to function 249, 359, 392, 394–395, 406–407,  
    422  
  support of BOM using 627  
onmousemove property 405–406, 414, 424, 426  
onmouseout property 414, 426  
onresize property 415, 427  
OR (||) operator 54, 55, 62–63, 74

# P

parameters, function  
  about 84, 88  
  assigning argument values to 85  
  identifying 91, 105, 119, 120  
  names in constructors 532  
  naming 97  
  none used in function 94  
  passing arguments to 89  
  vs. arguments 90  
parentheses ()  
  in calling functions 68, 430, 439  
  in parameters 97  
parseGuess function 351–354  
parse method, in JSON 636  
parsing function declaration 432  
pass-by-values 92–93  
passing arguments 88–89  
passing by reference 192  
passing by value 192  
pause event 419  
Phrase-o-Matic application (example) 131–133  
playlists  
  in jQuery 625  
  populating with song titles using JavaScript array 253,  
    262  
  using document objects in 626  
plus sign (+)  
  as arithmetic operator 286–287, 312  
  in concatenating strings 15, 133, 142, 354  
position: absolute 324–325, 328  
positioning, CSS 324–326, 328  
position: relative 324, 328  
post-decrement operator (--) 146–147  
post-increment operator (++) 146–147  
<p> (paragraph) elements  
  changing using JavaScript 247  
  innerHTML property changing 242

- primitive types
  - about 23, 266
  - identifying undefined values 267–268, 308
  - strings as objects and 294–296
  - vs. objects 187
- print method 627
- procedural design 329
- processGuess method 349–350, 357
- programming languages, learning 9
- prompt function 46, 53, 55
- properties
  - about 175
  - accessing 181
  - adding new 182
  - adding or deleting at any time 184
  - available in JavaScript 296
  - changing 182
  - changing in prototypes 582, 585
  - computing with 182
  - determining if in instance or in prototype 586–587, 592, 618
  - finding errors in code (exercise) 203
  - in chain of prototypes 592–593, 619
  - independent constructor 546–547
  - inheriting 571
  - iterating through object 209
  - methods vs. functions 206
  - naming 179
  - No Dumb Questions 185
  - objects as collections of 174
  - tour of string 297–300
  - undefined values and 268
  - using this object 202, 204–206, 219
  - values and 176, 219
- prototypal inheritance 569–572
- prototype model, object
  - about , xxii–xxiii
  - chain of prototypes
    - inheritance in 592–593, 619
    - setting up 591
  - determining if properties are in instance or in prototype 586–587, 592, 618
  - duplicating methods and 568
  - extending String prototype with method 610–611, 621
- implementing code 579, 583, 616–617
- inheritance
  - extending built-in object 610
  - in chain of prototypes 592–593, 619
  - in objects 569–572
  - of prototype from another prototype 594–599, 606, 620
  - overriding built-in behavior 608–609
- Object
  - about 607
  - as prototype 607
  - overriding built-in behavior 608–609
- object diagram exercise 574, 615
- prototypes in
  - about 569
  - adding methods to 581–582, 584
  - changing properties in 582, 585
  - dynamic 582
  - getting 575
  - overriding 573, 578
  - setting up 575
- updating constructor
  - chain of prototypes in 591–593, 606, 619
  - cleaning up code 600–605
  - creating prototype that inherits from another prototype 594–599, 620
  - design for 588–590
- pseudocode
  - determining hits and misses in simplified Battleship game 59–60, 76
  - determining variables using 50
  - implementing loop using 51
  - translating into JavaScript code 52, 164–165, 171
  - working through 47–48, 73
- push method 152

## Q

### quality assurance (QA)

- about 61
- doing 61, 69–70
  - in advanced Battleship game 370–371
- querySelectorAll method 626
- querySelector method 626

queues, events and 404

quotation marks, double (“ “)

surrounding character strings in JavaScript 13

using around property name 179

# R

random locations for ships, generating in advanced Battleship game 362–368, 380

random numbers, generating 67–68

reader as learner xxviii

recursion 634–635

refactoring code 156, 159

reference, function

- about 430, 476
- action in function 491, 494
- assigning 477
- in calling function 491
- in passing function argument to another function 486
- passing 479
- substituting function expressions and 481

references, object 192

RegExp object 214, 551, 632

removeEventListener 630

replace, method 300

reportError method 587, 618

reserved words 12

resize event 419

return statement, function 95–97

rules of road, for creating objects 179

# S

sample files xxxiv

scope, variables 101

<script> element

- about 4
- anatomy of 35
- src attribute of 34–35

scripting languages 5

<script> tags

- in <head> or <body> element of HTML page 32

opening and closing 35

semicolon (;), ending statements in JavaScript 11, 13

server-side JavaScript 637

setAttribute method 255, 333, 391

setInterval function 410, 413, 425

setInterval method 627

setTimeout function 410, 480, 501

setTimeout method 407–413, 627

shadowing variables 104

showAnswer handler 413

slashes (//), beginning JavaScript comments 13

slash (/) operator, as division arithmetic operator 15, 286

slice, method 300

sort method, array 457–463, 472

sparse Array 152

split method 299

square brackets ([ ])

- accessing properties using 209

- in arrays 127, 129, 550

src attribute, of <script> element 34–35

src property 390, 391

state and behavior, in objects 210–212, 226

statements

- ending with semicolon 11, 13

- variables in 11–13

- writing 10

strict equality (====) operator

- as comparison operator 55, 280–285, 311

- vs. == operator 289

String prototype, extending with method 610–611, 621

strings

- in arrays 132

- as primitives and objects 294–296

- as primitive types 266

- comparing to numbers 275–277, 281

- concatenating 15, 133, 142, 286–287, 312, 354

- conditional as 23

- falsey value are empty 292

- operators to sort 459

- prompt function returning for numbers 55

- properties available in JavaScript 296

secret life of 294–296  
 tour of methods and properties 297  
 substring method 299–300, 302, 304  
 Sun Microsystems 6  
 support material xxxiv  
 syntax, JavaScript  
   about 13  
   finding errors in 14, 39

## T

<table> element  
   creating HTML page 322  
   using CSS to position 325  
 target 398–399, 401  
 <td> element 322, 325, 327–328, 333  
 TextEdit (Mac), as text editor 31  
 text editors 31  
 this object  
   in constructors 528–529, 532  
   prototypes and 580  
   using 202, 204–206, 219  
 timer events 407–409  
 timerHandler function 407–409  
 toLowerCase, method 300  
 toString method 531, 548, 557, 607–608  
 touchend event 419  
 touchstart event 419  
 toUpperCase, method 300  
 tracing flow of execution (tracing) 96  
 treasure map game 405–406, 424  
 tree, DOM structure as 235  
 trim, method 300  
 true and false (boolean values) 13, 15–16, 291–293, 313  
 truthy and falsey values 291–293, 313  
 try/catch, handling exceptions using 629  
 typeof operator 269, 305, 309, 315, 542  
 types  
   about , xvi–xvii  
   conversion 277–280

equality of object 288–290, 314  
 instanceof 305, 315, 543, 547  
 knowing 301–304  
 typeof exercise and 269  
 using == and === operators on different 275–285,  
   311  
 using arithmetic operators for conversion of 286–287,  
   312  
 using null 55, 270–271, 274, 310  
 values and 272–274, 281, 292

## U

undefined, as default value of variables 50  
 undefined values  
   about 267–268, 308  
   as falsey 292  
   comparing to null 278  
   returned by functions without return statement 97  
   vs. null 270, 310  
 underscore (\_), beginning variable names 13  
 Unicode values 281  
 unload event 419  
 users  
   communicating with  
     using alert function 25–26, 46  
     using console.log function 25–27  
     using document object model 25  
     using prompt function 46, 53

## V

values  
   equality and 276  
   functions as 439–440, 467  
   functions as first class  
     about 442–443  
     extreme JavaScript challenge 487, 508, 517  
     passing functions to functions 443–448, 468, 482,  
       486, 514  
     returning functions from 450–456, 470, 472  
     substituting function expressions 481  
     using array sort method 457–463, 472–473  
   identifying undefined 267–268, 308  
   NaN (Not a Number) 272–274, 281, 292  
   null 270

object properties and 176, 219

objects 266

operators and 286

operators to sort 459

passing values to functions 92–93, 192

primitive 266

strict equality 280

strings 291–297

truthy and falsey 291–293

type conversion and 276–281

types in arrays 134

undefined 268

variables and 11

variable declaration, object 177

variables

about 11

assigning functions to 439, 449, 469

conditional as 23

declaring in arrays 152

declaring inside and outside of functions 98

declaring local 103, 108

default value of 50

finding errors in 39

free 495–496, 501, 516

global

guide to code hygiene 111

identifying 105, 120

lexical scope of 488–490, 494

nesting affecting 486, 488–490, 494

overuse of 391

scope of 101

vs. local variables 99, 106–107

guide to code hygiene 111

identifying 91, 119

in arrays 127, 129

lexical scope of 488–490, 494

local

declaring 103, 108

guide to code hygiene 111

identifying 105, 120

in methods 206

lexical scope of 488–490, 494

nesting affecting 486, 488–490, 494

scope of 101

vs. global variables 99, 106–107

naming 12–13, 100, 103, 108

objects assigned to 192

objects in 186

passing into arguments 89

referring to functions to invoke functions 436

reloading page and 108

scope of 101

setting to null 270

shadowing 104

short life of 102

undefined values and 268

using pseudocode in determining 50

var keyword in declaring 97, 99

var keyword

about 11

in declaring variables 97, 99

view object, advanced Battleship game

about 329

implementing 331–334

interacting with model 336

notifying of hits or misses 347

testing 335

## W

W3C documentation website 627

web browsers

conditional statements handled by 434

events 404

executing code 433

function declarations handled by 431–433, 436, 465, 483

function expressions handled by 434–436, 465

handling events 383, 403. *See also* handling events

loading and executing JavaScript in 3

loading code into 31

objects provided by 214

opening console 28

parsing HTML and building DOM from it 261

recommended xxvi

running JavaScript in xxxii

setting up Event Object in IE8 and older 399

setting up event objects in IE8 and older 631

tailoring size of window to web page 627

web browser wars 6

web pages

adding code to HTML page 32

as applications 9

- creating dynamic 5–6
- creating interactive 319
- JavaScript in 2–3
- JavaScript interacting with 233
- tailoring size of browser window to 627
- using <script> element in 4
- while loop
  - about 17
  - executing code using 18–21
  - vs. for loop 144–145
- white space in JavaScript code 13
- wickedlysmart.com 640
- window object
  - about 214
  - addEventListener method 630
  - confirm method 627
  - creating onload event handler for 249, 359, 402
  - history method 627
  - innerHeight property 627
  - innerWidth property 627
  - location method 627
  - methods and properties available for 627
  - onload property
    - assigning handler to 385–386, 389–390, 410, 630
    - in anonymous functions 476–477
    - setting to function 249, 359, 392, 394–395, 406–407, 422
    - support of BOM using 627
  - onresize property 415, 427
  - print method 627
  - setInterval method 627
  - setTimeout function 407–413
  - setTimeout method 627
  - support of BOM in 627
- write function 25–26, 42
- writing JavaScript. *See also* Battleship game, advanced; Battleship game, simplified; code
  - about 3, 7–8
  - analyzing code 80–81, 116
  - being verbose in specifying conditionals 65, 74
  - chaining 345, 348
  - comparing to numbers to strings 275–277, 281
  - doing quality assurance 61, 69–70
- duplicating code in objects 208
- else if statements, making decisions using 22–23
- expressions in 15–16, 40
- finding errors in code
  - exercise 14, 39, 203, 224
  - using console 68
- flowchart for designing program 45
- functions and reusing code 83–88
- generating random numbers 67–68
- guide to code hygiene 111
- null in 256, 270–271, 274, 278
- prompt function in 46, 53
- pseudocode
  - determining hits and misses in simplified Battleship game 59–60, 76
  - determining variables using 50
  - implementing loop using 51
  - translating into JavaScript code 52, 164–165, 171
  - working through 47–48, 73
- refactoring code 156, 159
- reusing code , 71, 79, xii
- statements 10
  - undefined as default value of variables 50
  - using = operator, assigning values to variables using 11, 16, 275
- using == operator
  - as comparison operator 16, 55, 275–285, 311, 459
  - vs. === operator 289
- using === operator
  - as comparison operator 55, 280–285, 311
  - vs. == operator 289
- using variables 11–13
- values in 272–274, 281
- while statement, executing code using 18–21

## Z

zero (0), as falsey value 292



I can't believe the book is almost over.  
Before you go, you really should read the  
index. It's great stuff. And after that you've  
always got the website. So I'm sure we'll see  
you again soon...

## Don't worry, this isn't goodbye.

Nor is it the end. Now that you've got an amazingly solid foundation in JavaScript, it's time to become a master. Point your browser to <http://wickedlysmart.com/hfjs> to explore what's next!

What's next? So much more! Join us at <http://wickedlysmart.com/hfjs> to continue your journey.



