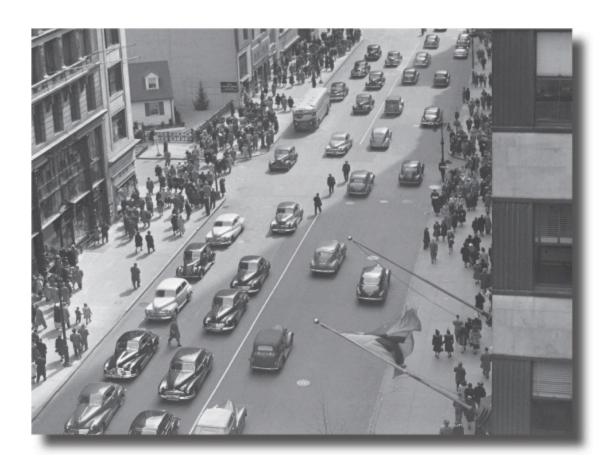
=

Chapter 12. Creating objects

Advanced Object Construction



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,
                                                         With an object literal you type
                                                         out each part of the object
    started: false,
                                                         within curly braces. When you're
                                                         done, the result is an actual
    start: function() { this.started = true;},
                                                         JavaScript object, which you
    stop: function() { this.started = false;},
                                                          typically assign to a variable for
                                                          later use.
    drive: function {
          // drive code here
    }
};
```

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?

- $\hfill\Box$ 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.
- $\hfill\Box$ Who needs taxis when we have Uber?

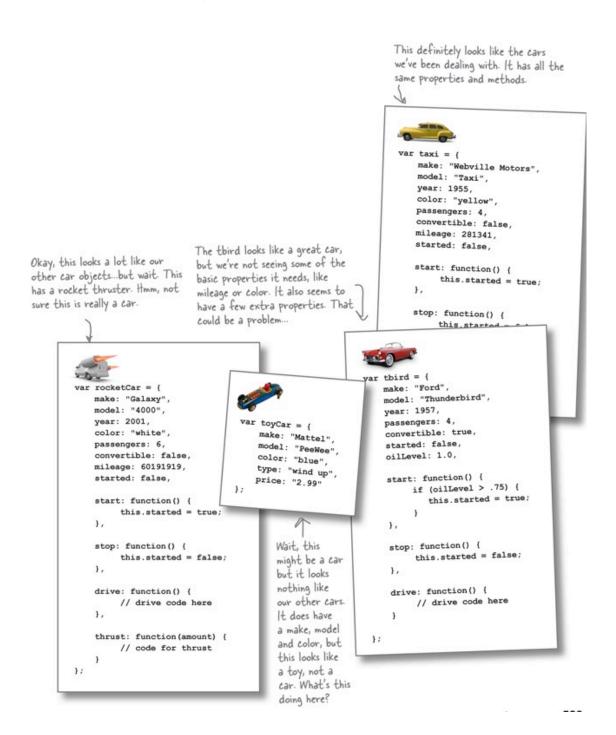
Using conventions for objects

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:



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...



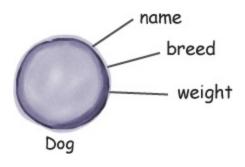
Introducing Object Constructors

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.

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



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.

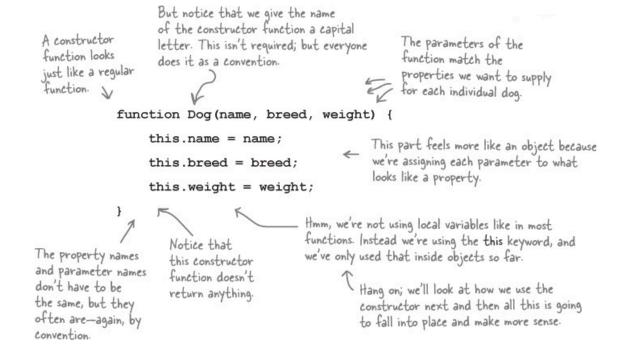


How to create a Constructor



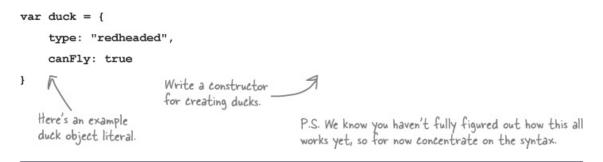
Using constructors is a two-step process: first we define a constructor, and then we use it to create objects. Let's first focus on creating a constructor.

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:



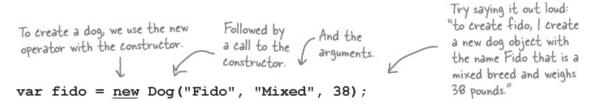
SHARPEN YOUR PENCIL

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:



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:



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.

EXERCISE

```
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:

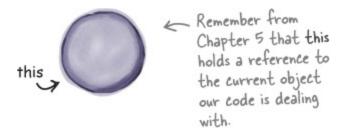
```
var fido = new Dog("Fido", "Mixed", 38);
```

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

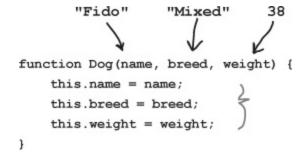
1. The first thing new does is create a new, empty object:



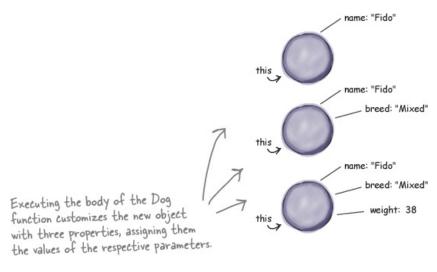
2. Next, new sets this to point to the new object.



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

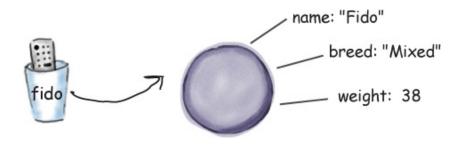


4. Next the body of the function is invoked. Like most constructors, Dog assigns values to properties in the newly created this object.



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:

NOTE

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

```
To add a bark method we simply
                                                  assign a function, in this case
 function Dog(name, breed, weight) {
                                                  an anonymous function, to the
                                                  property this.bark.
    this.name = name;
    this.breed = breed;
    this.weight = weight;
     this.bark = function() {
         if (this.weight > 25) {
              alert(this.name + " says Woof!");
                                                                Now every dog object will
                                                                 also have a bark method
                                                                that you can invoke
              alert(this.name + " says Yip!");
         }
    };
}
               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.
```

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();
}</pre>
```



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

EXERCISE

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;

    Write the two methods for
    this constructor here.
```

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

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

JAVASCRIPT CONSOLE

You've ordered a medium House Blend coffee.

You've ordered a large Dark Roast coffee.

Here's our output; yours should look similar.

THERE ARE NO DUMB QUESTIONS

- *Q*: **Q**: Why do constructor names start with a capital letter?
- A: 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: Q: So, other than setting up the properties of the this object, a constructor's just like a regular function?
- A: 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: Q: Do the parameter names of a constructor function have to match the property names?
- A: 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: Q: Is an object created by a constructor just like an object created with a literal?
- A: A: Yes, until you get into more advanced object design, which we'll do in the next chapter.
- Q: 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: 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: 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: 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: Q: Is it better to create objects with a constructor than with object literals?
- A: 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-timeuse only object, and literals are concise and expressive to use for this.

NOTE

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

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

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

};

Oops we forgot to use new!

Dut maybe that's okay

because Album is a function.

}

var darkside = Album("Dark Side of the Cheese", "Pink Mouse", 1971);

darkside.play();

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.

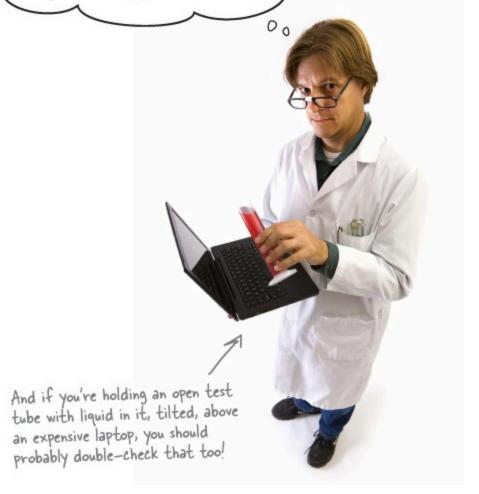
NOTE

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 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.



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.



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 cadi = {
  make: "GM",
  model: "Cadillac",
var chevy = {
                                                                                                    year: 1955,
color: "tan",
     make: "Chevy",
     model: "Bel Air",
                                                                                                    passengers: 5,
convertible: false,
     year: 1957,
                                                                                                    mileage: 12892,
     color: "red"
                                                                                                            false,
     passengers: 2,
                                                                                                             function() {...},
                                                                         var fiat = {
  make: "Fiat",
  model: "500",
                                                                                                            unction() {...},
function() {...}
     convertible: false,
     mileage: 1021,
     started: false,
                                                                             year: 1957,
                                                                             color: "Medium Blue",
                                                                           passengers: 2,
convertible: false,
mileage: 88000,
started: false,
start: function() {
      start: function() {
        this.started = true;
      },
                                                                           stop: function()
drive: function()
      stop: function() {
        this.started = false;
                                                                                                  var taxi = {
                                                                                                      make: "Webville Motors",
model: "Taxi",
      drive: function() {
                                                                                                      year: 1955,
                                                                                                      color: "yellow",
         if (this.started) {
                                                                                                      passengers: 4,
convertible: false,
            console.log(this.make + " " +
                     this.model + " goes zoom zoom!");
                                                                                                      mileage: 281341,
                                                                                                      started: false,
         } else {
                                                                                                      start: function() {...},
            console.log("Start the engine first.");
                                                                                                     stop: function() {...},
drive: function() {...}
         }
                                                                                                 };
      }
};
```

EXERSICE

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() {	
3	
J	
NOTE	
All your work goes here.	
NOTE	
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.

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

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

```
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! →
 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();
 }
```

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();
}</pre>
```

```
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!

Webville Motors Test Car goes zoom zoom!

Webville Motors Test Car goes zoom zoom!
```

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?

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;
                                               And when we write code that calls this
                                               constructor we have to make sure we get the
    this.passengers = passengers;
                                               arguments all in exactly the right order.
    this.convertible = convertible;
    this.mileage = mileage;
    this.started = false;
    this.start = function() {
        this.started = true;
    };
    //rest of the methods here
}
```

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.

NOTE

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.

NOTE

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

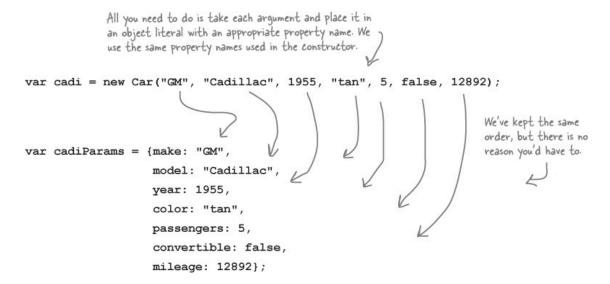
However, there is a common technique that we can use when passing all these arguments that can be used for any function, whether or not it's a constructor. The technique works like this: take all your arguments, throw them in an object literal, and then pass that literal to your function—that way you're 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.

Rewiring the arguments as an object literal



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



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

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",
                                             No changes here, we've just
                     year: 1955,
                                                 reproduced the object literal and
                                                  the call to the Car constructor from the previous page.
                     color: "tan",
                     passengers: 5,
                     convertible: false,
                     mileage: 12892};
                                          First things first. We'll replace the seven
                                          parameters of the Car constructor with one
var cadi = new Car(cadiParams);
                                          parameter, for the object we're passing in.
function Car(params) {
                                                     Then for each reference to a
                                                     parameter, we substitute the
    this.make = params.make;
                                                      corresponding property from the
    this.model = params.model;
                                                      object passed into the function.
    this.year = params.year;
    this.color = params.color;
    this.passengers = params.passengers;
    this.convertible = params.convertible;
    this.mileage = params.mileage;
                                                  In our methods we never use a parameter directly. It
    this.started = false;
                                                  wouldn't make sense to because we always want to use
                                                  the object's properties (which we do using the this
    this.start = function() {
                                                  variable). So, no changes are needed to this code at all.
         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.");
    };
```

TEST DRIVE



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

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

EXERCISE

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.

NOTE

You'll find the Dog constructor in <u>You can put methods into constructors as well</u>.

NOTE

Put the output here.

BRAIN POWER

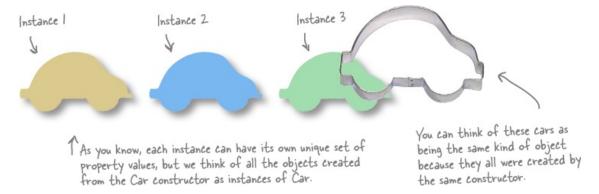
Say someone handed you an object and you wanted to know what type of object it was (is it a Car? a Dog? Superman?), or you wanted to see if it was the same type as another object. Would the typeof operator be helpful?

- Q: Q: Remind me what typeof does again?
- A: 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: Q: So if typeof can't tell me that my object is a dog or car, how do I determine what is what?
- A: 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:

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 instance of uses that to determine if an object is an instance of a certain constructor.

NOTE

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

JAVASCRIPT CONSOLE

Congrats, it's a Car!

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!");
    }
}
```



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:

```
there'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.

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) {

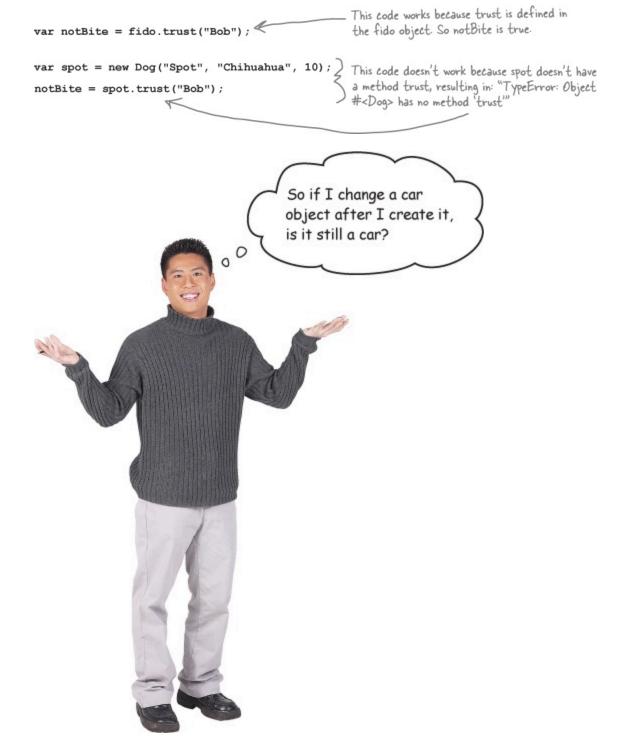
return (person === "Bob");

};

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:



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:

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

NOTE

Evaluates to true.

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.

NOTE

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

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:

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.

Returns a number for the day of the week represented by the date object, like I (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*.

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:

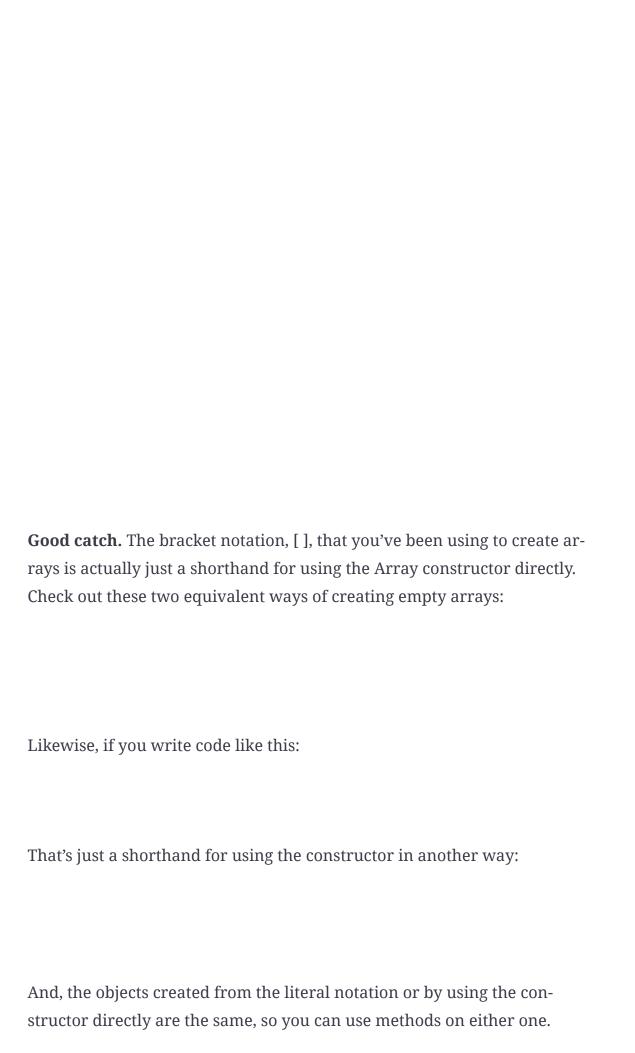
Here, we're creating a new, empty array object. And at any time we can add items to it, like this:

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

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:

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.



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:

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.

MATH

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

REGEXP

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

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: 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 that's a number, it knows that's the size of the array, and if it has more, then those arguments are all initial values.
- Q: Q: Can we do that with our constructors?
- A: 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: 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: 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: 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: A: You can check to see if two objects have the same constructor like this:

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

- Q: 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.

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	Down
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	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.

NOTE

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

EXERCISE SOLUTION

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.

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

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.

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.

JAVASCRIPT CONSOLE

You've ordered a medium House Blend coffee.

You've ordered a large Dark Roast coffee

Here's our output; yours should look similar.

EXERCISE SOLUTION

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.

EXERCISE SOLUTION

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:

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:

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.

JAVASCRIPT CROSS SOLUTION		