Chapter 5. Understanding Objects: *A trip to Objectville*

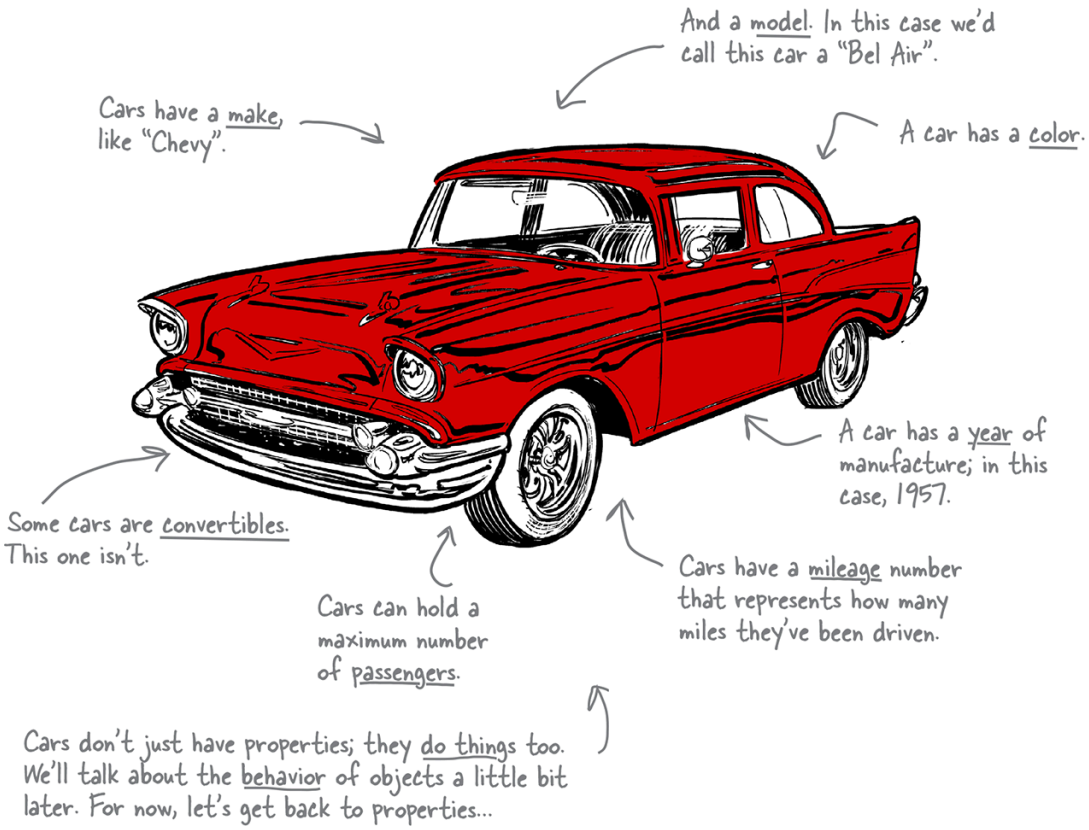
So far, you've been using primitives and arrays in your code. And you've approached coding in quite a **procedural manner**, using simple statements, conditionals, and for/while loops with functions—that's not exactly **object-oriented**. In fact, it's not object-oriented *at all!* You did use a few objects here and there without really knowing it, but you haven't written any of your own objects yet. Well, the time has come to leave this boring procedural town behind and create some **objects** of your own. In this chapter, you're going to find out why using objects is going to make your life so much better—well, better in a **programming sense** (we can't really help you with your fashion sense *and* your JavaScript skills all in

one book). Just a warning: once you've discovered objects, you'll never want to come back. Send us a postcard when you get there.

Did someone say "objects"?!

Ah, our favorite topic! Objects are going to take your JavaScript programming skills to the next level—they're the key to managing complex code, to understanding the browser's document model (which we'll get to in the next chapter), to organizing your data...and they're even the fundamental way many JavaScript libraries are packaged up. So objects must be a difficult topic, right? Hah! We're going to jump in head first, and you'll be using them in no time.

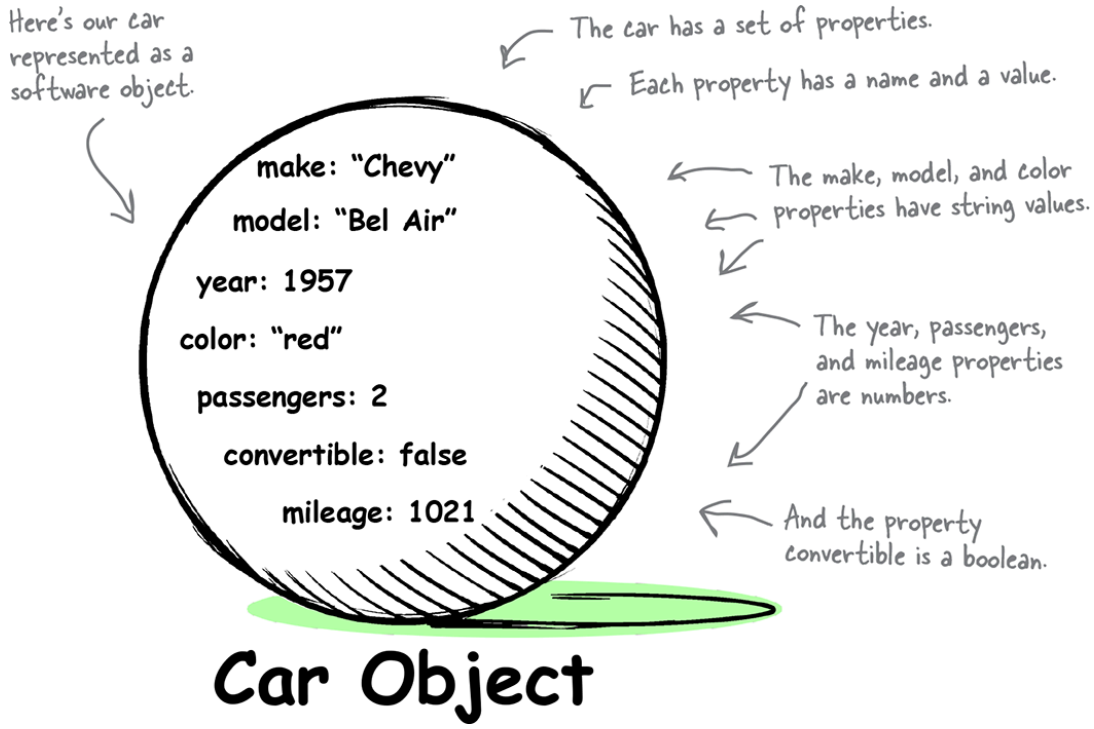
Here's the secret to JavaScript objects: they're just collections of properties. Let's take an example—say, a car. A car has various properties:



Thinking about properties...

Of course, there's a lot more to a real car than just a few properties, but for the purposes of coding, these are the properties we want to capture in

software. Let's think about these properties in terms of JavaScript data types:



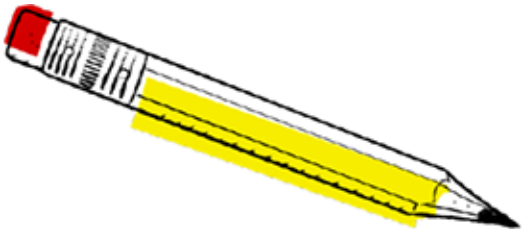
BRAIN POWER

Are there other properties you'd want to have in a car object? Go ahead and think through all the properties you might come up with for a car and write them below. Remember, only some real-world properties are going to be useful in software.

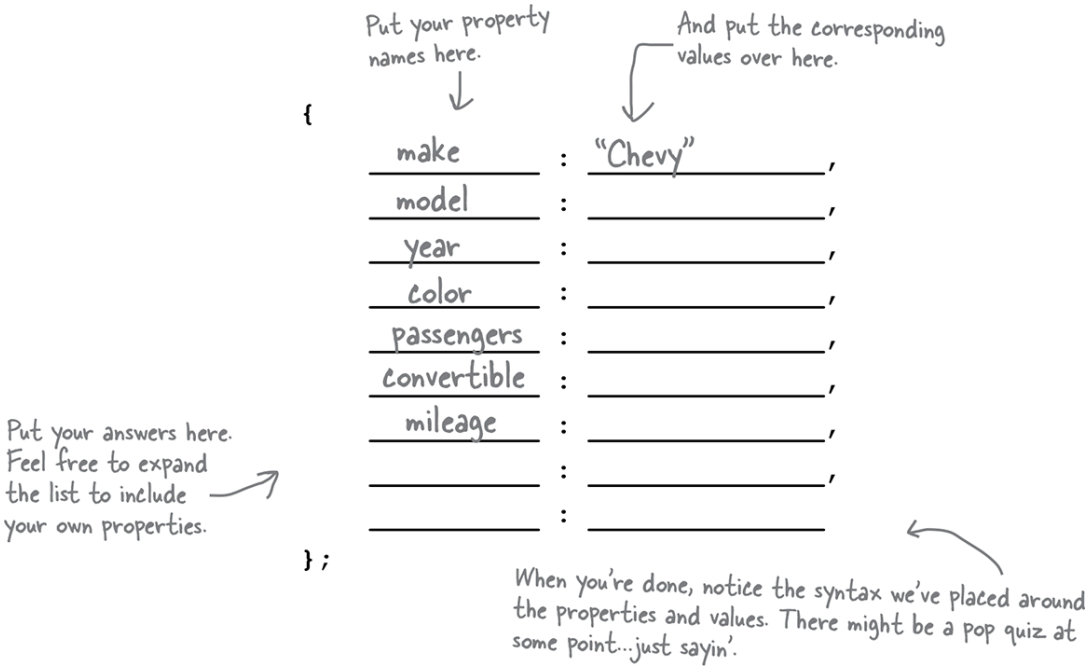
NOTE

Those fuzzy dice may look nice, but would they really be useful in an object?

SHARPEN YOUR PENCIL



We've started making a table of property names and values for a car. Can you help complete it? Make sure you compare your answers with ours before moving on!

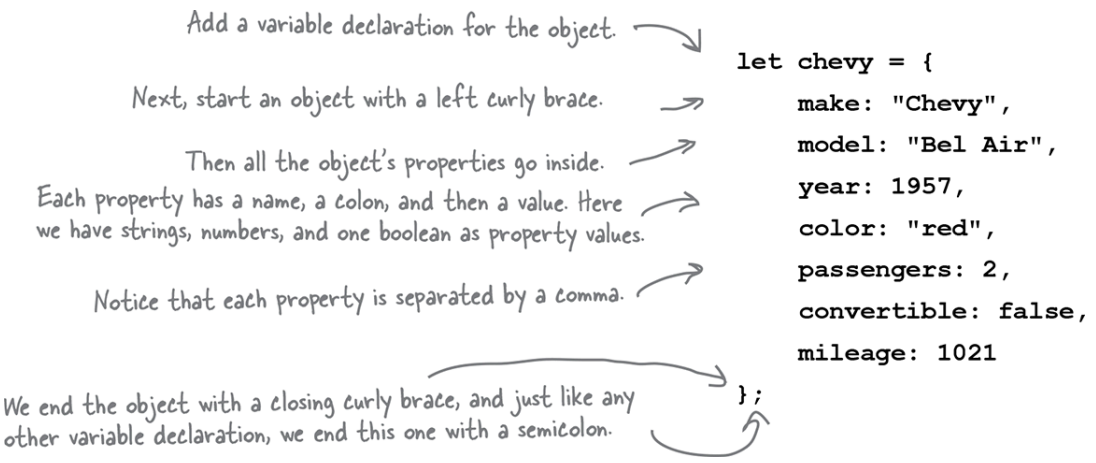


→ **Solution in "Sharpen your pencil Solution"**

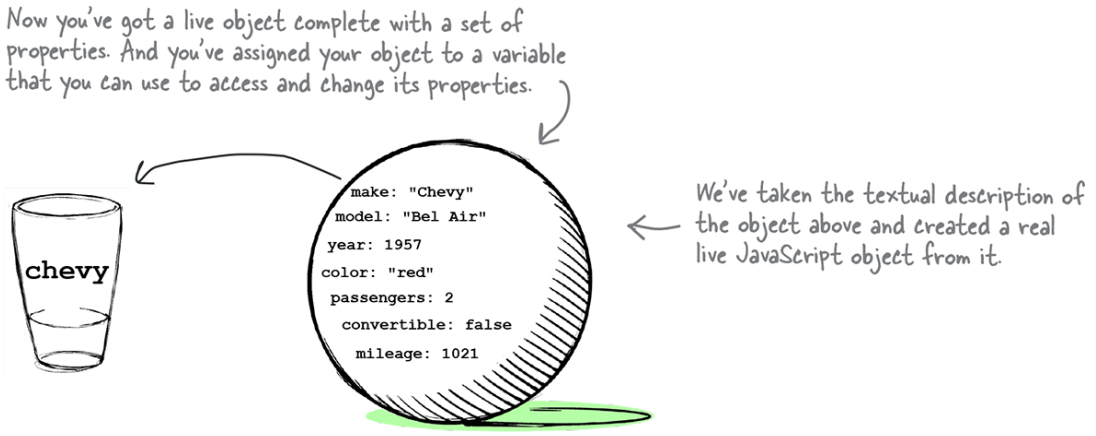
What if the car is a taxi? What properties and values would it share with your ’57 Chevy? How might they differ? What additional properties might it have (or not have)?

How to create an object

Here’s the good news: after the last Sharpen your Pencil exercise, you’re already most of the way to creating an object. All you really need to do is assign what you wrote on the previous page to a variable (so you can do things with your object after you’ve created it). Like this:



The result of all this? A brand new object, of course. Think of the object as something that holds all your names and values (in other words, your properties) together.



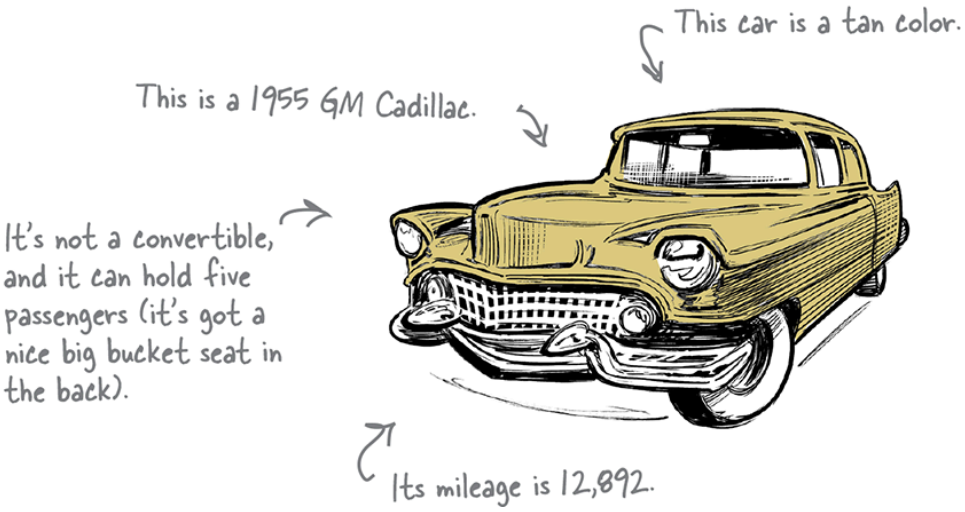
You can now take your object, pass it around, get values from it, change it, add properties to it, or take them away. We'll get to how to do all that in a second. For now, let's create some more objects to play with...

EXERCISE

You don’t have to be stuck with just one object. The real power of objects (as you’ll see soon enough) is having lots of objects and writing code that can operate on whatever object you give it. Try your hand at creating another object from scratch...another car object. Go ahead and work out the code for your second object. Check your answer at the end of the chapter before you go on.

```
let cadi = {

};
```

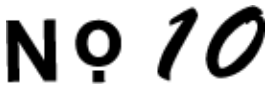
Put the properties for your Cadillac object here.



Solution in "Exercise Solution"

SPEEDING TICKET

Issued by the *Webville Police Dept*.

WARNING CITATION 

Don't worry, you're getting off easy this time; rather than issuing a ticket, we ask that you please review the following "rules of the road" for creating objects.

Make sure you enclose your object in curly braces:

```
let cat = {
   name: "fluffy"
};
```

Separate the property name and property value with a colon:

```
let planet = {
    diameter: 49528
};
```

A property name can be any string, but we usually stick with valid variable names:

```
let widget = {
    cost$: 3.14,
    "on sale": true
};
```

NOTE

Notice that if you use a string with a space in it for a property name, you need to use quotes around the name.

No two properties in an object can have the same name:

```
let forecast = {
    highTemp: 82,
    highTemp: 56
};
```

NOTE

WRONG!

Separate each property name and value pair with a comma:

```
let gadget = {
   name: "anvil",
   isHeavy: true
};
```

Don't use a comma after the last property value:

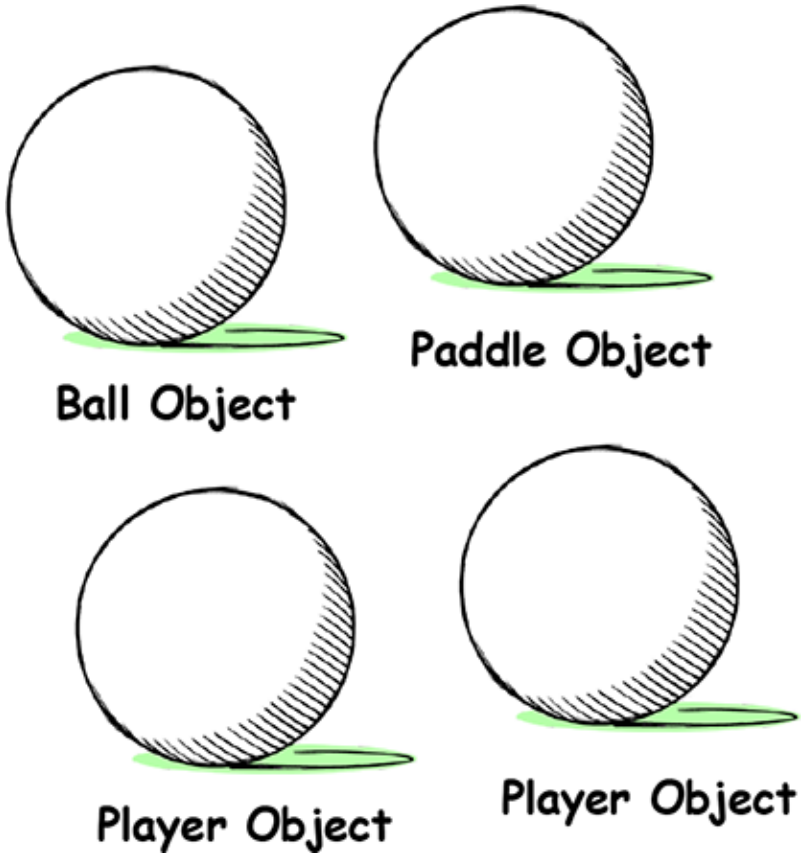
```
 let superhero = {
   name: "Batman",
   alias: "Caped Crusader"
 };
```

NOTE

No comma needed here!

What is "object-oriented" anyway?

Up until now, we've been thinking of a problem as a set of variable declarations, conditionals, for/while statements, and function calls. That's thinking *procedurally*: first do this, then do this, and so on. With *object-oriented* programming, we think about a problem in terms of objects. Objects that have state (like a car might have oil and fuel levels) and behavior (like a car can be started, driven, braked, and stopped).



What's the point? Well, object-oriented programming (or OOP for short) allows you to free your mind to think at a higher level. It's the difference between having to toast your bread from first principles (create a heating coil out of wire, hook it to electricity, turn the electricity on and then hold your bread close enough to toast it, not to mention watch long enough for it to toast and then unhook the heating coil), and just using a toaster (place bread in toaster and push down on the toast button). The first way is procedural, while the second way is object-oriented: you have a toaster object that supports an easy method of inserting bread and toasting it.

WHAT DO YOU LIKE ABOUT OOP?

“It helps me design in a more natural way. Things have a way of evolving.”

-Joy, 27, software architect

“Not messing around with code I’ve already tested, just to add a new feature.”

-Brad, 32, programmer

“I like that the data and the methods that operate on that data are together in one object.” -Josh, 22, beer drinker

“Reusing code in other apps. When I write a new object, I can make it flexible enough to be used in something new, later.”

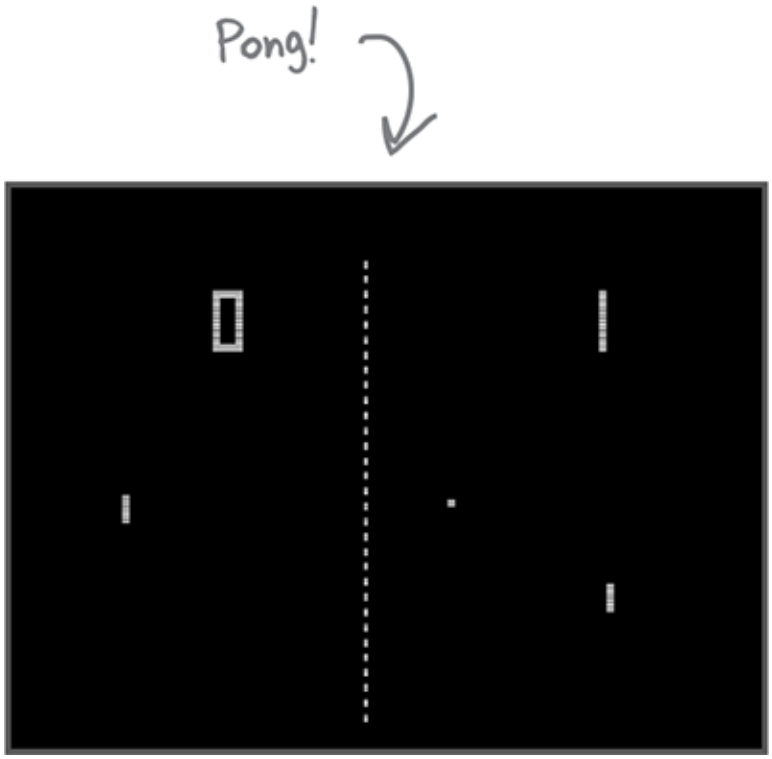
-Chris, 39, project manager

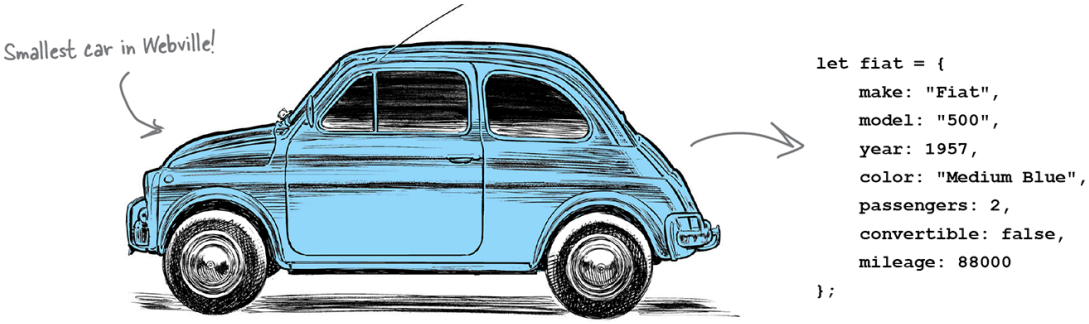
“I can’t believe Chris just said that. He hasn’t written a line of code in five years.”

-Daryl, 44, works for Chris

BRAIN POWER

Say you were implementing a classic Ping-Pong style video arcade game. What would you choose as objects? What state and behavior do you think they'd have?

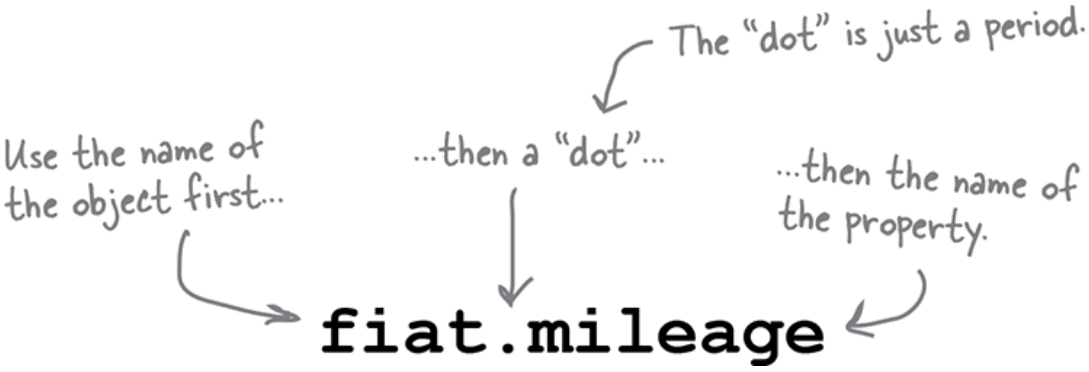




How properties work

So you've got all your properties packaged up in an object. Now what? Well, you can examine the values of those properties, change them, add new properties, take away properties, and in general, compute using them. Let's try a few of these things out, using JavaScript of course.

How to access a property. To access a property of an object, start with the object name, follow it with a period (otherwise known as a "dot"), and then use the property name. We often call that "dot" notation, and it looks like this:



And then we can use a property in any expression, like this:

```
let miles = fiat.mileage;
if (miles < 2000) {
   buyIt();
}
```

NOTE

Start with the variable that holds your object and add a period (otherwise known as a dot) and then your property name.

DOT NOTATION

- Dot notation (.) gives you access to an object's properties.
- For example, `fiat.color` is a property of `fiat` with the name `color` and the value "Medium Blue".

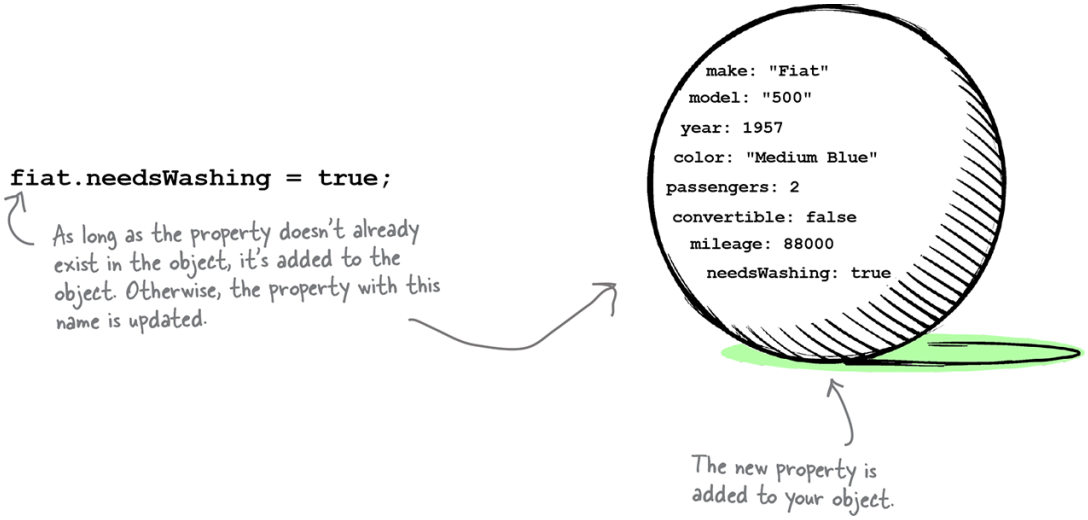
How to change a property. You can change the value of a property at any time. All you need to do is assign the property to a new value. Like, let's say you wanted to set the mileage of your nifty Fiat to an even 10,000. You'd do that like this:

```
fiat.mileage = 10000;
```

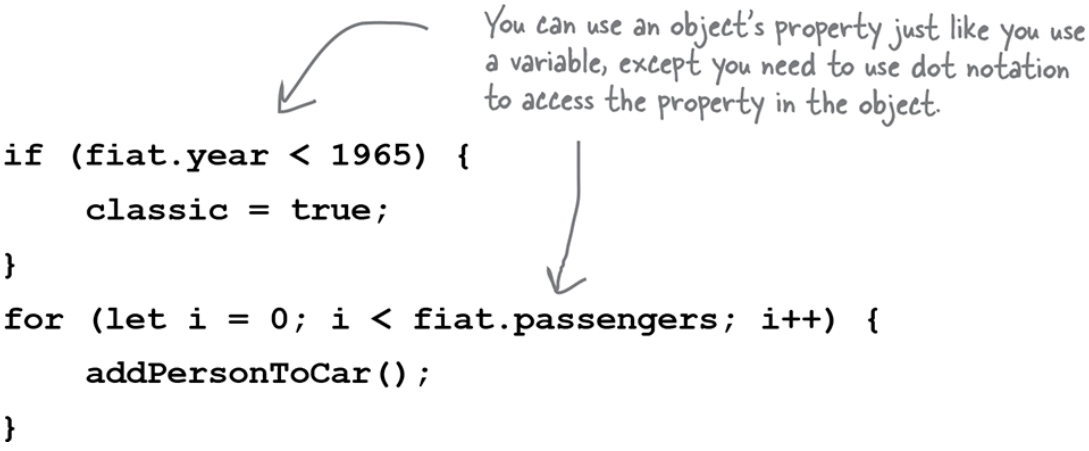
NOTE

Just specify the property you want to change and then give it a new value. Note: in some states this may be illegal!

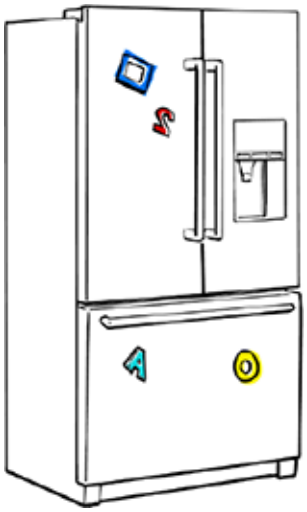
You can extend your object at any time with new properties. To do this, you just specify the new property and give it a value. For instance, say you wanted to add a boolean that indicates when the Fiat needs to be washed:



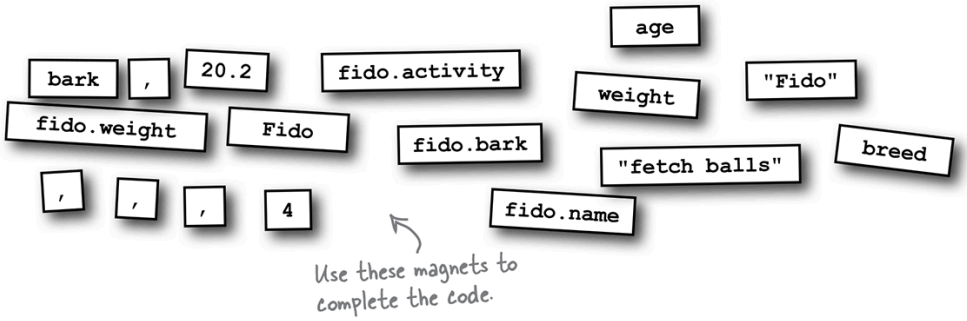
How to compute with properties. Computing with properties is simple: just use a property like you would any variable (or any value). Here are a few examples:



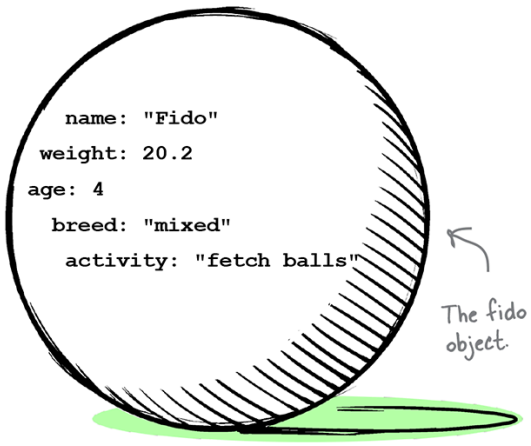
OBJECT MAGNETS



This code got all scrambled up on the fridge. Practice your object creating and dot notation skills by putting it all back together. Be careful, some extra magnets might have gotten mixed in!




```
let fido = {
    name: _________
    _________: 20.2
    age: _________
    _________: "mixed",
    activity: _____________
};
let bark;
if (_____________ > 20) {
   bark = "WOOF WOOF";
} else {
   bark = "woof woof";
}
let speak = __________ + " says " + _________ + " when he wants to " + _____________;
console.log(speak);
```



Solution in [“Object Magnets Solution”]

Yes, you can add or delete properties at any time. As you know, to add a property to an object you simply assign a value to a new property, like this:

```
fido.dogYears = 35;
```

and from that point on `fido` will have a new property, `dogYears`. Easy enough.

To delete a property, we use a special keyword...wait for it... `delete.` You use the `delete` keyword like this:

```
delete fido.dogYears;
```

When you delete a property, you're not just deleting the value of the property, you're deleting the property itself. And if you try to use `fido.dogYears` after deleting it, it will evaluate to `undefined`.

The `delete` expression returns `true` if the property was deleted successfully. `delete` will return `false` only if it can't delete a property (which could happen for, say, a protected object that belongs to the browser). It will return `true` even if the property you're trying to delete doesn't exist in the object.

THERE ARE NO DUMB QUESTIONS

Q: How many properties can an object have?

A: As few or as many as you want. You can have an object with no properties, or you can have an object with hundreds of properties. It's really up to you.

Q: How can I create an object with no properties?

A: Just like you create any object, only leave out all the properties. Like this:

```
let lookMaNoProps = { };
```

Q: I know I just asked how to create an object with no properties, but why would I want to do this?

A: Well, you might want to start with an entirely empty object and then add your own properties dynamically, depending on the logic of your code. Here's an example:

```
let lookMaNoProps = { };
lookMaNoProps.age = 10;
if (lookMaNoProps.age > 5) {
  lookMaNoProps.school = "Elementary";
}
```

Q: What's better about an object than just using a bunch of variables? After all, each of the properties in the fiat object could just be its own variable, right?

A: Objects package up your data so that you can focus on the high-level design of your code, not the nitty-gritty details. Say you want to write a traffic simulator with dozens of cars; you'll want to focus on cars and streetlights and road objects and not hundreds of little variables. Objects also make your life easier because they encapsulate, or hide, the complexity of the state and behavior of your objects so you don't have to worry about them. How all that works will become much clearer as you gain experience with objects.

Q: If I try to add a new property to my object, and the object already has a property with that name, what happens?

A: If you try to add a new property, like needsWashing, to fiat and fiat already has a property needsWashing, then you'll be changing the existing value of the property. So if you say:

```
fiat.needsWashing = true;
```

but fiat already contains a property needsWashing with the value false, then you're changing the value to true.

Q: What happens if I try to access a property that doesn't exist? Like if I said:

```
if (fiat.make) { ... }
```

but fiat didn't have a property make?

A: The result of the expression fiat.make will be undefined if fiat doesn't have a property named make.

Q: What happens if I put a comma after the last property?

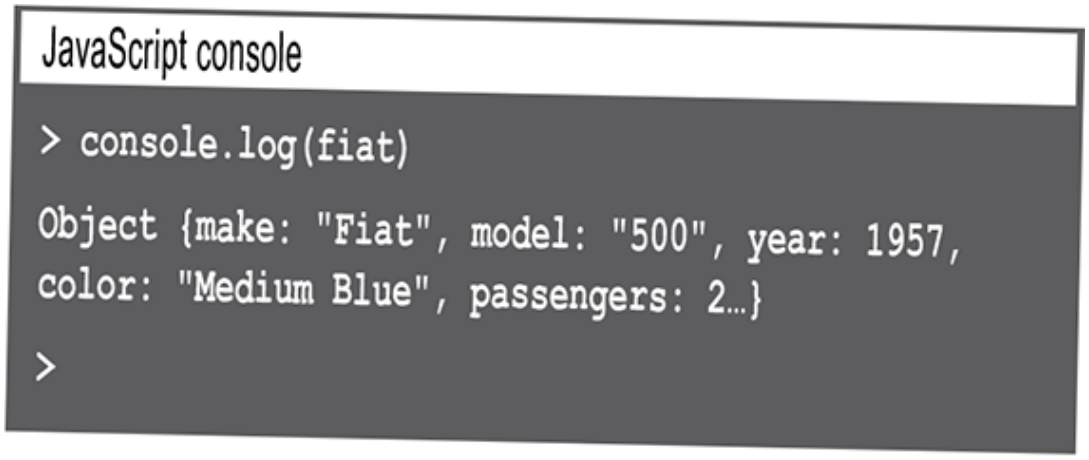
A: In modern browsers it won't cause an error. However, in older versions of some browsers this will cause your JavaScript to halt execution.

Q: Can I use console.log to display an object in the console?

A: You can. Just write:

```
console.log(fiat);
```

in your code, and when you load the page with the console open, you'll see information about the object displayed in the console.



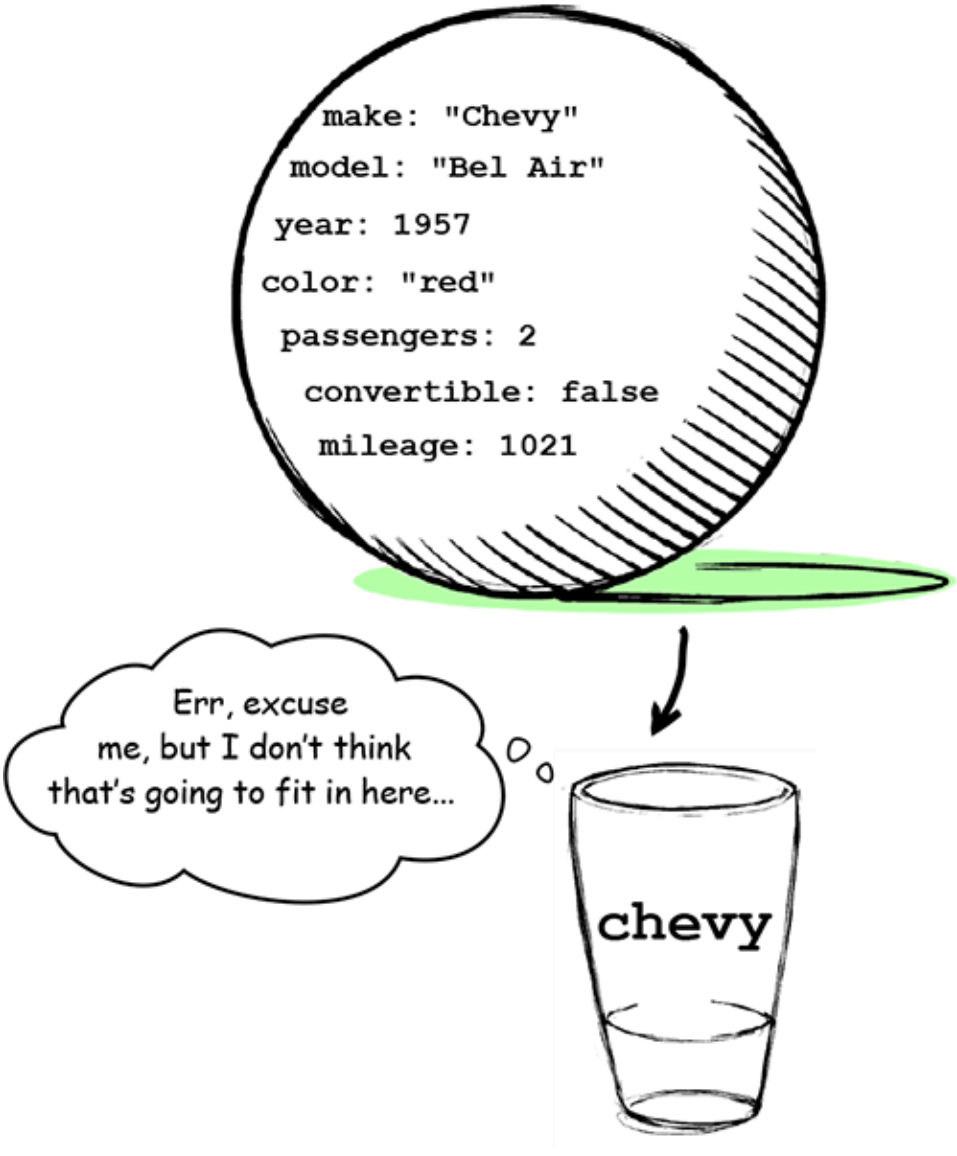
You can also use console.table(fiat)...try it!

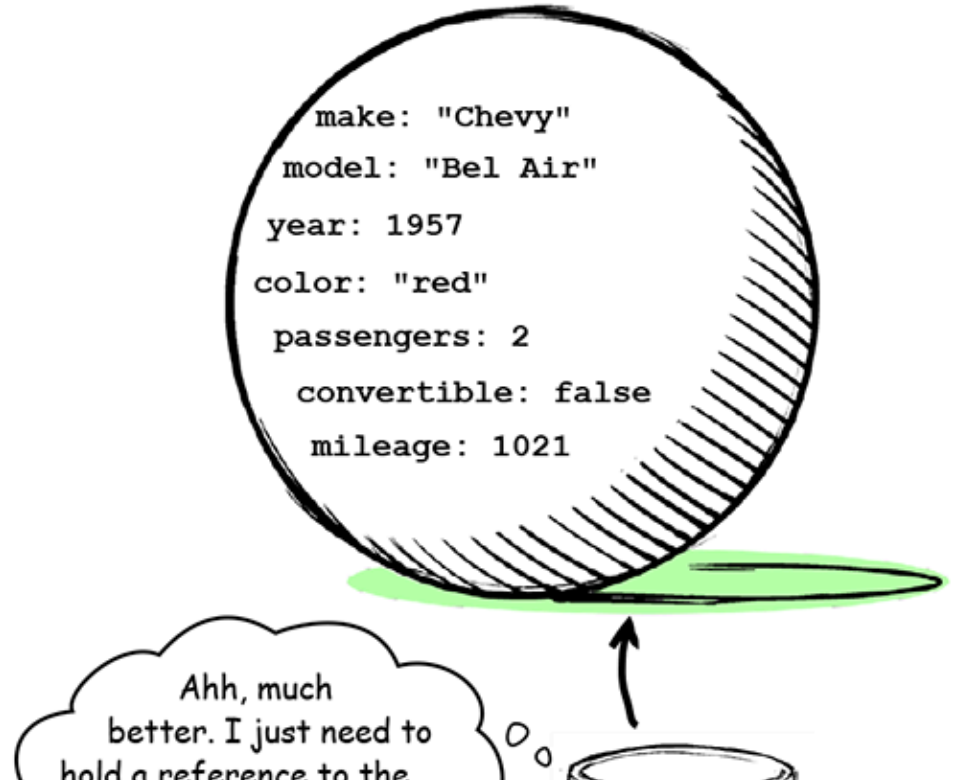
How does a variable hold an object? Inquiring minds want to know...

You've already seen that a variable is like a container, and it holds a value. But numbers, strings, and booleans are pretty small values. What about objects? Can a variable hold any sized object, no matter how many properties you put in it?

- **Variables don't actually hold objects.**
- **Instead, they hold a** reference **to an object.**
- **The reference is like a pointer to the actual object.**
- **In other words, a variable doesn't hold the object itself, but it holds something like a pointer to the object. And, in JavaScript, we don't really know what is inside a reference variable. We do know that whatever it is, it points to our object.**
- **When we use dot notation, the JavaScript interpreter takes care of using the reference to get the object and then accesses its properties for us.**

Behind the Scenes


make: "Chevy"
model: "Bel Air"
year: 1957
color: "red"
passengers: 2
convertible: false
mileage: 1021
Err, excuse me, but I don't think that's going to fit in here...
chevy


make: "Chevy"
model: "Bel Air"
year: 1957
color: "red"
passengers: 2
convertible: false
mileage: 1021
Ahh, much better. I just need to

So, you can't stuff an object into a variable. We often think of it that way, but that's not what happens; variables aren't giant expandable cups that can grow to the size of any object. Instead, an object variable just holds a reference to the object.

Here's another way to look at it: a primitive variable represents the actual *value* of the variable, while an object variable represents *a way to get to the object*. In practice you'll only need to think of objects as, well, objects, like dogs and cars, not as references, but knowing variables contain *references* to objects will come in handy later (and we'll see that in just a few pages).

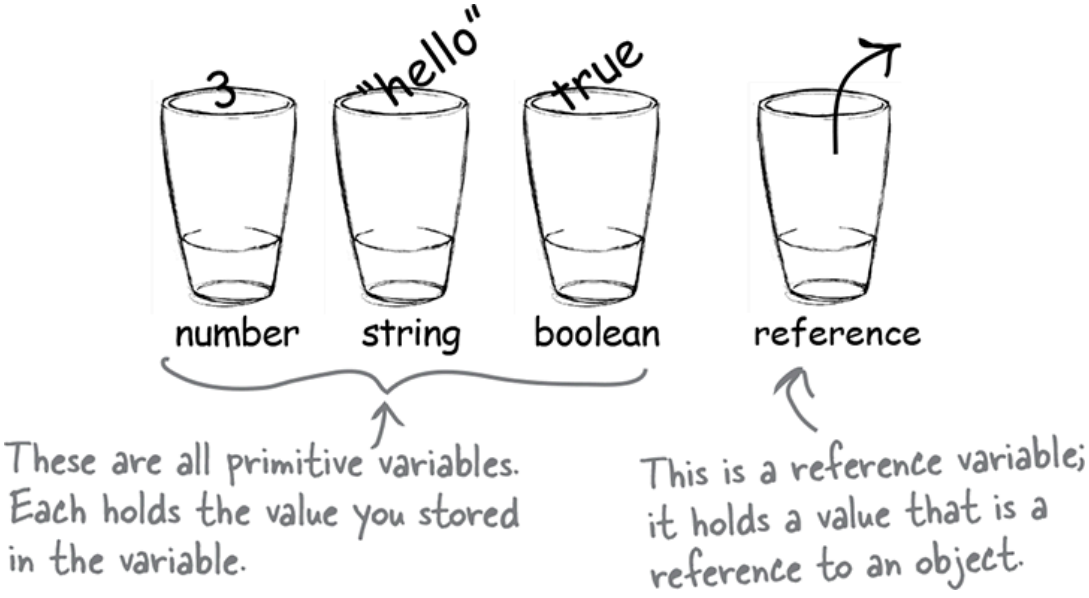
And also think about this: you use dot notation (.) on a reference variable to say "use the reference *before* the dot to get me the object that has the property *after* the dot." (Read that sentence a few times and let it sink in.) For example:

```
car.color;
```

means "use the object referenced by the variable `car` to access the `color` property."

Comparing primitives and objects

Think of an object reference as just another variable value, which means that we can put that reference in a cup, just like we can primitive values. With primitive values, the value of a variable is... the *value*, like 5, –26.7, "hi", or false. With reference variables, the value of the variable is a *reference*: a value that represents a way to get to a specific object.



We don't know (or care) how the JavaScript interpreter represents object references.

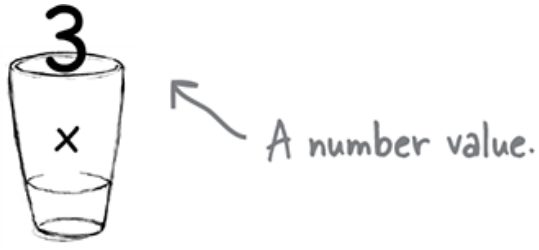
We just know we can access an object and its properties using dot notation.

Initializing a primitive variable

When you declare and initialize a primitive, you give it a value, and that value goes right in the cup, like this:

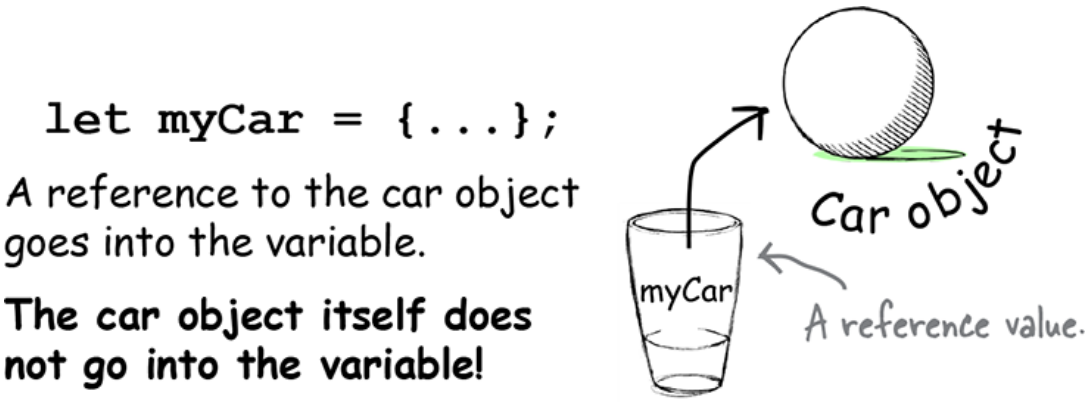
```
let x = 3;
```

The variable holds the number three.



Initializing an object (reference) variable

When you declare and initialize an object, you make the object using object notation, but that object won't fit in the cup. So what goes in the cup is a *reference* to the object:



Doing even more with objects

Let's say you're looking for a good car for your stay in Webville. Your criteria? How about:

- □ Built in 1960 or before
- □ 10,000 miles or less

You also want to put your new coding skills to work (and make your life easier), so you want to write a function that will "prequalify" cars for you—that is, if the car meets your criteria, then the function returns true; otherwise, the car isn't worth your time and the function returns false.

More specifically, you're going to write a *function* that *takes a car object as an argument* and puts that car through the test, returning a boolean value. Your function is going to work for *any car object*.

Let's give it a shot:

Here's the function.

You're going to pass it a car object.

```
function prequal(car) {
    if (car.mileage > 10000) {
        return false;
    } else if (car.year > 1960) {
        return false;
    }
    return true;
}
```

Just use dot notation on the car parameter to access the mileage and year properties.

Test each property value against the prequalification criteria.

If either of the disqualification tests succeeds, we return false. Otherwise, we return true, meaning we've successfully prequalified!

Now let's give this function a try. First, you need a car object. How about this one:

```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};
```

Doing some prequalification

We've done enough talking about objects. Let's actually create one and put it through its paces using the `prequal` function. Grab your favorite basic HTML page ("prequal.html") and throw in the code below. Then load the page and see if the taxi qualifies:

```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};

function prequal(car) {
    if (car.mileage > 10000) {
        return false;
    } else if (car.year > 1960) {
        return false;
    }
    return true;
}

let worthALook = prequal(taxi);

if (worthALook) {
    console.log("You gotta check out this " + taxi.make + " " + taxi.model);
} else {
    console.log("You should really pass on the " + taxi.make + " " + taxi.model);
}
```

Does the taxi cut it?

Here's what we got. Let's quickly trace through the code on the next page to see how the taxi got rejected...

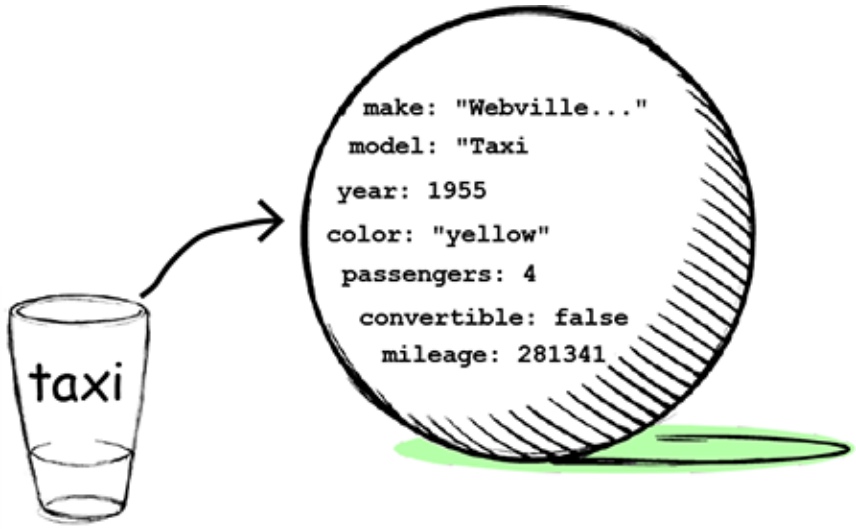
JavaScript console

```
You should really pass on the Webville Motors Taxi
```

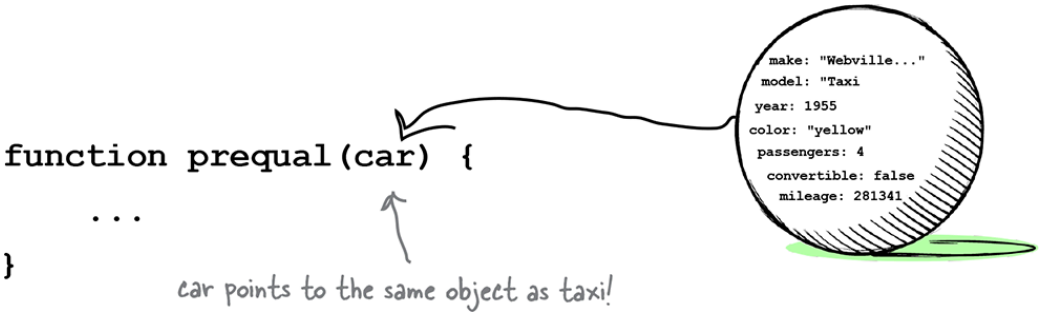
Stepping through prequalification

1. First, we create the taxi object and assign it to the variable `taxi`. Of course, the `taxi` variable holds a reference to the taxi object, not the object itself.

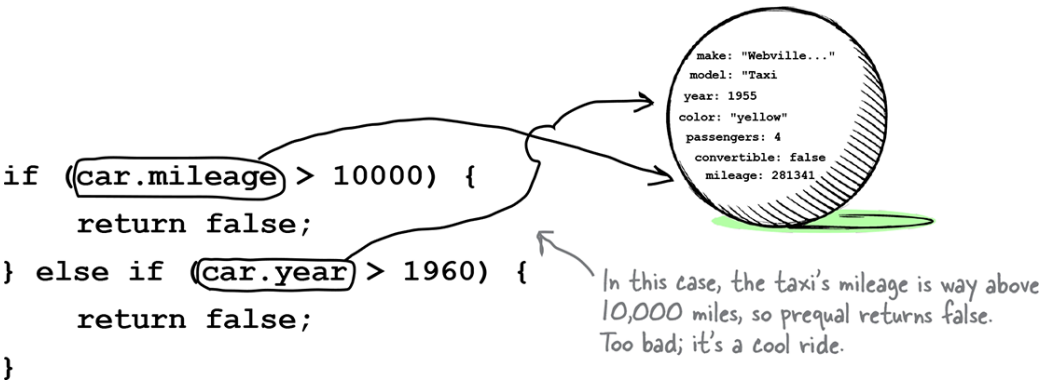
```
let taxi = { ... };
```



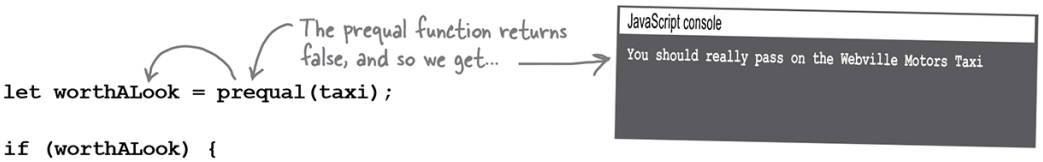
2. Next, we call `prequal`, passing it the value in the argument `taxi`, which is assigned to the parameter `car`.



3 We then perform the tests in the body of the function, using the taxi object in the `car` parameter.

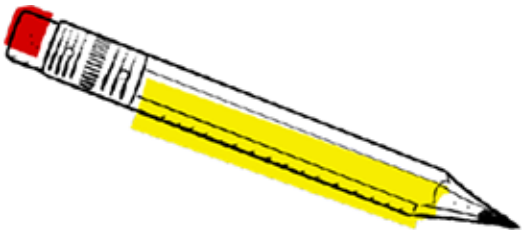


4 Unfortunately, the taxi has a lot of miles, so the first test of car.mileage > 10000 is true. The function returns false, so `worthALook` is set to false. We then get "You should really pass on the Webville Motors Taxi" displayed in the console.




```
if (worthALook) {
    console.log("You gotta check out this " + taxi.make + " " + taxi.model);
} else {
    console.log("You should really pass on the " + taxi.make + " " + taxi.model);
}
```

SHARPEN YOUR PENCIL



Your turn. Here are three more car objects; what is the result of passing each car to the prequal function? Work out the answers by hand, and then write the code to check your answers.

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

prequal(cadi);
```

_____

Write the value of prequal here.

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};

prequal(fiat);
```

_____

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

prequal(chevy);
```

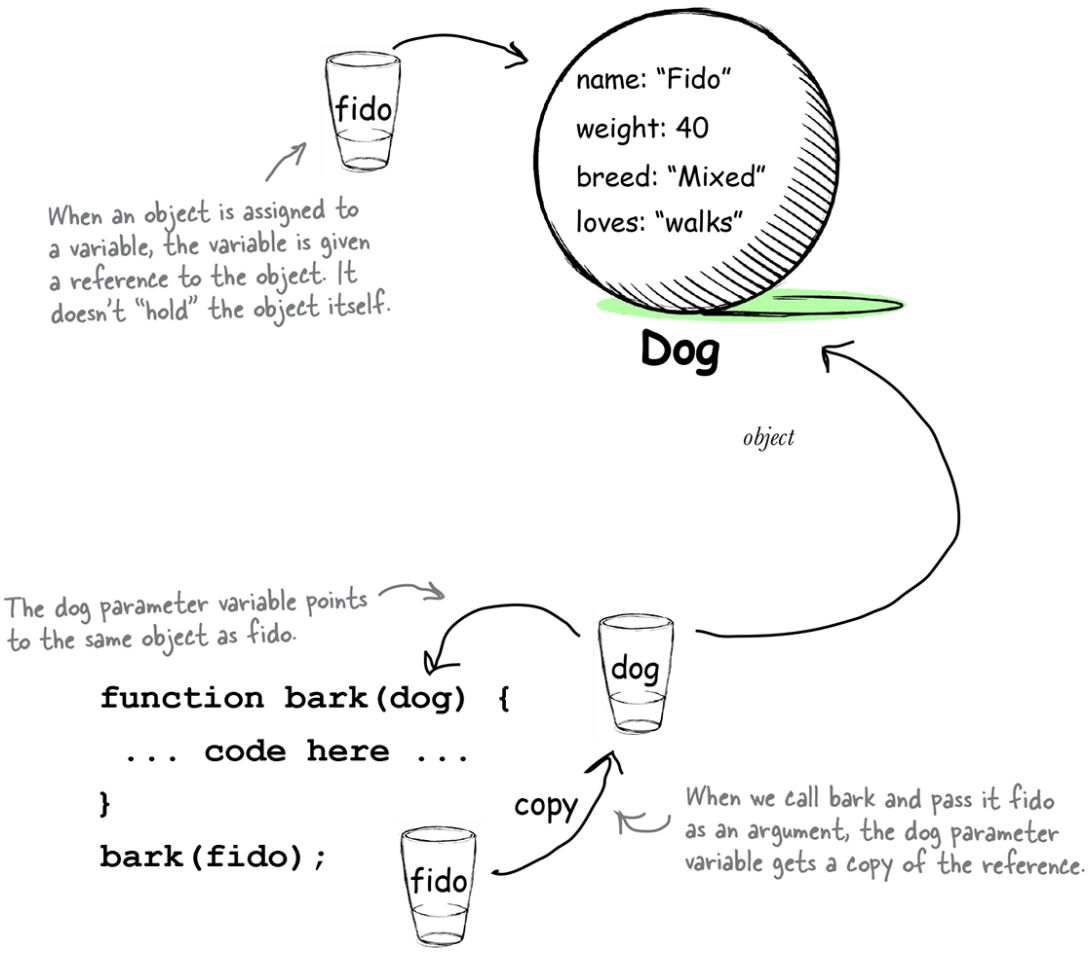
_____

→ **Solution in "Sharpen your pencil Solution"**

Let's talk a little more about passing objects to functions

We've already talked a bit about how arguments are passed to functions —arguments are *passed by value*, which means *pass-by-copy*. So if we pass an integer, the corresponding function parameter gets a copy of the value of that integer for its use in the function. The same rules hold true for objects; however, we should look a little more closely at what pass-by-value means for objects to understand what happens when you pass an object to a function.

You already know that when an object is assigned to a variable, that variable holds a *reference* to the object, not the object itself. Again, think of a reference as a pointer to the object:



So, when you call a function and pass it an object, you're passing the *object reference*, not the object itself. Using our pass-by-value semantics, a

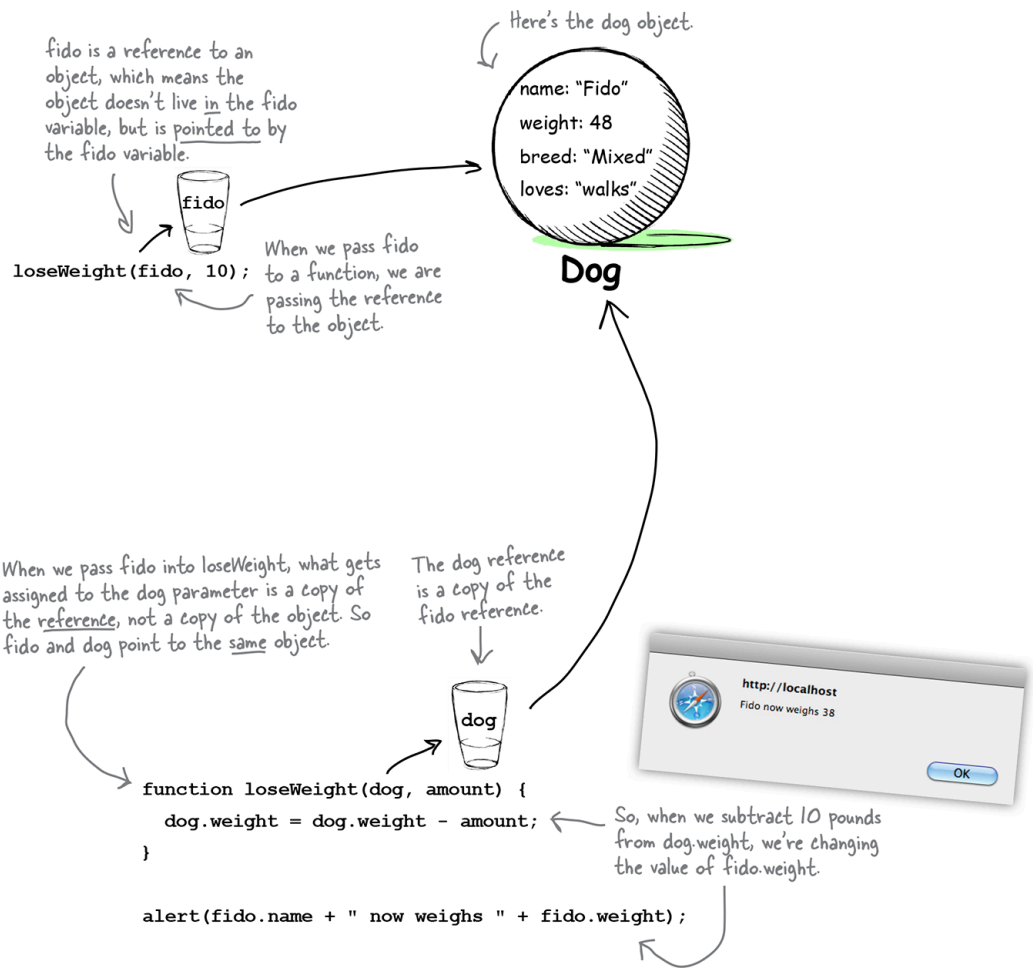
copy of the reference is passed into the parameter, and that reference remains a pointer to the original object.

So what does this all mean? Well, one of the biggest ramifications is that if you change a property of the object in a function, you're changing the property in the *original* object. So, any changes you make to the object inside a function will still be there when the function completes. Let's step through an example...

Putting Fido on a diet

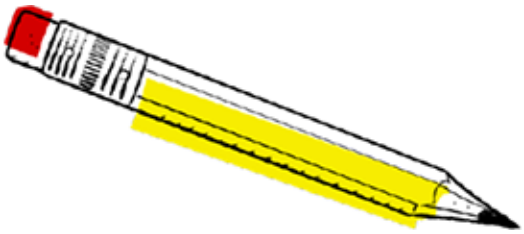
Let's say we are testing a new method of weight loss for dogs, which we want to neatly implement in a function named `loseWeight`. All you need to do is pass `loseWeight` your dog object and an amount to lose, and like magic, the dog's weight will be reduced. Here's how it works:

① First, check out the dog object, fido, which we are going to pass to the loseWeight function:



② The dog parameter of the loseWeight function gets a copy of the reference to fido. So, any changes to the properties of the parameter variable affect the object that was passed in.

SHARPEN YOUR PENCIL



You've been given a super-secret file and two functions that allow access to get and set the contents of the file, but only if you have the right password. The first function, getSecret, returns the contents of the file if the password is correct and logs each attempt to access the file. The second function, setSecret, updates the contents of the file and resets the access tracking to 0. It's your job to fill in the blanks below to complete the JavaScript and test your functions.

```
function getSecret(file, secretPassword) {
    _______.opened = _______.opened + 1;
    if (secretPassword == _______.password) {
        return ______.contents;
    }
    else {
        return "Invalid password! No secret for you.";
    }
}
function setSecret(file, secretPassword, secret) {
    if (secretPassword == _______.password) {
        ______.opened = 0;
        ______.contents = secret;
    }
}

let superSecretFile = {
    level: "classified",
    opened: 0,
    password: 2,
    contents: "Dr. Evel's next meeting is in Detroit."
};
let secret = getSecret(_________________, _____);
console.log(secret);

setSecret(_________________, _____, "Dr. Evel's next meeting is in Philadelphia."
```

```
secret = getSecret(_____, _____);
console.log(secret);
```

→ **Solution in "Sharpen your pencil Solution"**

The Auto-O-Matic

TheAuto-O-Matic is a variation of the Phrase-O-Matic from Chapter 4, this time designed to sell cars.

```
<!doctype html>
<html lang="en">
<head>
  <title>Auto-O-Matic</title>
  <meta charset="utf-8">
  <script>
    function makeCar() {
        let makes = ["Chevy", "GM", "Fiat", "Webville Motors", "Tucker"];
        let models = ["Cadillac", "500", "Bel-Air", "Taxi", "Torpedo"];
        let years = [1955, 1957, 1948, 1954, 1961];
        let colors = ["red", "blue", "tan", "yellow", "white"];
        let convertible = [true, false];

        let rand1 = Math.floor(Math.random() * makes.length);
        let rand2 = Math.floor(Math.random() * models.length);
        let rand3 = Math.floor(Math.random() * years.length);
        let rand4 = Math.floor(Math.random() * colors.length);
        let rand5 = Math.floor(Math.random() * 5) + 1;
        let rand6 = Math.floor(Math.random() * convertible.length);

        let car = {
            make: makes[rand1],
            model: models[rand2],
            year: years[rand3],
            color: colors[rand4],
            passengers: rand5,
            convertible: convertible[rand6],
            mileage: 0
        };
        return car;
    }

    function displayCar(car) {
        console.log("Your new car is a " + car.year + " " + car.make + " " + car.model);
    }

    let carToSell = makeCar();
    displayCar(carToSell);

  </script>
</head>
<body></body>
</html>
```

The Auto-O-Matic is similar to the Phrase-O-Matic from Chapter 4, except that the words are car properties, and we're generating a new car object instead of a marketing phrase!

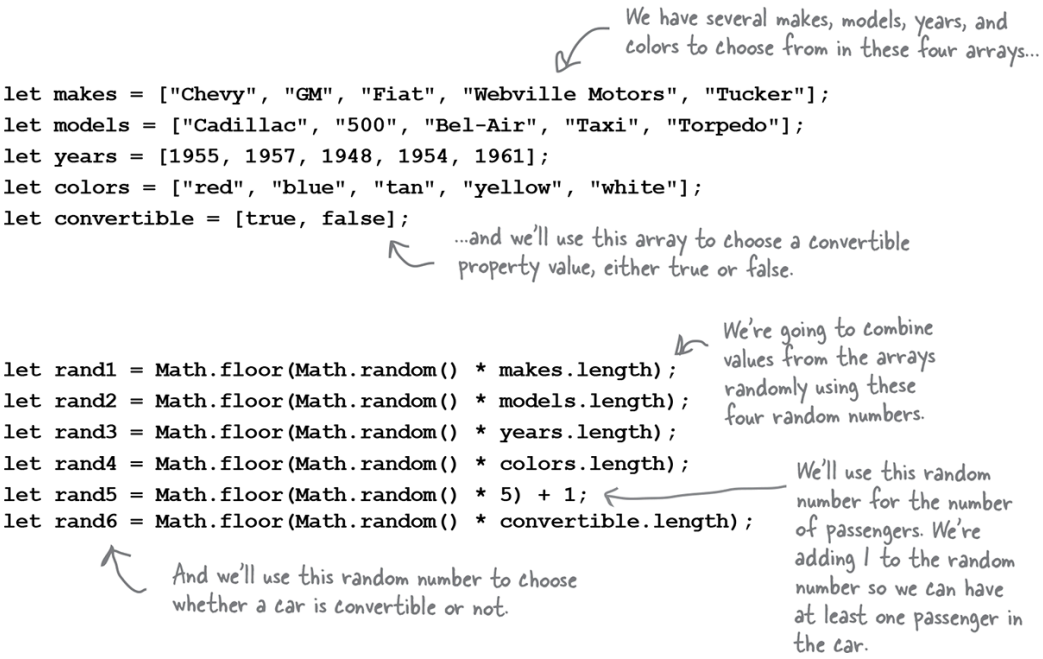
Check out what it does and how it works.

The Auto-O-Matic

Brought to you by the same guy who brought you the Phrase-O-Matic, the Auto-O-Matic creates knock-off cars all day long. That is, instead of generating marketing messages, this code generates makes, models, years, and

all the other properties of a car object. *It's your very own car factory in code*. Let's take a closer look at how it works:

1. First, we have a `makeCar` function that we can call whenever we want to make a new car. We've got four arrays with the makes, models, years, and colors of cars, and an array with true and false options for whether a car is a convertible. We generate five random numbers so we can pick a make, a model, a year, a color, and whether the car is a convertible randomly from these five arrays. And we generate one more random number we're using for the number of passengers:



2. Instead of creating a string by mixing and matching the various car properties, like we did with the Phrase-O-Matic, this time we're creating a new object, `car`. This car has all the properties you'd expect. We pick values for the `make`, `model`, `year`, and `color` properties from the arrays using the random numbers we created in step 1, and also add the `passengers`, `convertible`, and `mileage` properties.

```
let car = {
    make: makes[rand1],
    model: models[rand2],
    year: years[rand3],
    color: colors[rand4],
    passengers: rand5,
    convertible: convertible[rand6],
    mileage: 0
};
```

We're creating a new car object, with property values made from the values in the arrays.

We're also setting the number of passengers to the random number we created and setting the convertible property to true or false using the convertible array.

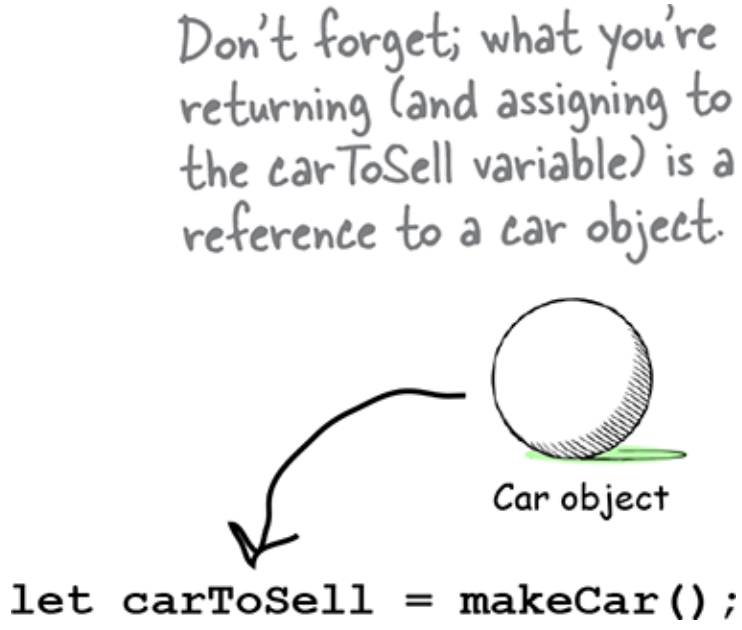
Finally, we're just setting the mileage property to 0 (it is a new car, after all).

3 The last statement in `makeCar` returns the new `car` object:

```
return car;
```

Returning an object from a function is just like returning any other value. Let's now look at the code that calls `makeCar`:

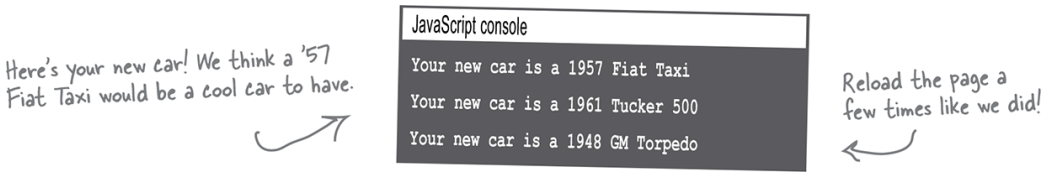
```
function displayCar(car) {
  console.log("Your new car is a " + car.year + " " +
              car.make + " " + car.model);
}
let carToSell = makeCar();
displayCar(carToSell);
```



First, we call the `makeCar` function and assign the value it returns to `carToSell`. We then pass the `car` object returned from `makeCar` to

the function `displayCar`, which simply displays a few of its properties in the console.

④ Go ahead and load up the Auto-O-Matic in your browser ("autoomatic.html") and give it a whirl. You'll find no shortage of new cars to generate, and remember there's a sucker born every minute.



Oh, behave! Or, how to add behavior to your objects

You didn't think objects were just for storing numbers and strings, did you? Objects are *active*. Objects can *do things*. Dogs don't just sit there...they bark, run, and play catch, and a dog object should too! Likewise, we drive cars, park them, put them in reverse, and make them brake. Given everything you've learned in this chapter, you're all set to add behavior to your objects. Here's how we do that:

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    drive: function() {
       alert("Zoom zoom!");
    }
};
```

You can add a function directly to an object like this.

All you do is assign a function definition to a property. Yup, properties can be functions too!

Notice we don't supply a name in the function definition; we just use the function keyword followed by the body. The name of the function is the name of the property.

And a bit of nomenclature: we typically refer to functions inside an object as methods. That is a common object-oriented term for a function in an object.

To call the `drive` function—excuse us—to call the `drive` *method*, you use dot notation again, this time with the object name `fiat` and the property name `drive`, only you follow the property name with parentheses (just like you would when you call any other function).

Improving the drive method

Let's make the `fiat` a little more car-like in behavior. Most cars can't be driven until the engine is started, right? How about we model that behavior? We'll need the following:

- □ **A boolean property to hold the state of the car (the engine is either on or off)**
- □ **A couple of methods to start and stop the car**
- □ **A conditional check in the drive method to make sure the car is started before we drive it**

We'll begin by adding a boolean `started` property along with methods to `start` and `stop` the car; then we'll update the `drive` method to use the `started` property:

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    started: false,

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

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

    drive: function() {
        if (started) {
            alert("Zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Here's the property to hold the current state of the engine (true if it is started and false if it is off).

Here's a method to start the car. All it does (for now) is set the started property to true.

And here's a method to stop the car. All it does is set the started property to false.

And here's where the interesting behavior happens: when we try to drive the car, if it is started we get a "Zoom zoom!" and if not, we get a warning that we should start the car first.

Good catch. You're right; to start the car we could have replaced the code:

```
fiat.start();
```

with:

```
fiat.started = true;
```

That would have saved us from writing a method to start the car.

So why did we create and call the `start` method instead of just changing the `started` property directly? Using a method to change a property is another example of encapsulation, whereby we can often improve the maintainability and extensibility of code by letting an object worry about how it gets things done. It's better to have a `start` method that knows how to start the car than for you to have to know "to start the car we need to take the `started` property and set it to true."

Now, you may still be saying, "What's the big deal? Why not just set the property to true to start the car?!" Consider a more complex `start` method that checks the seatbelts, ensures there is enough fuel, checks the battery, checks the engine temperature, and so on, all before setting `started` to true. You certainly don't want to think about all that every time you start the car. You just want a handy method to call that gets the job done. By putting all those details into a method, we've created a simple way for you to get an object to do some work while letting the object worry about how it gets that work done.

Take the fiat for a test drive

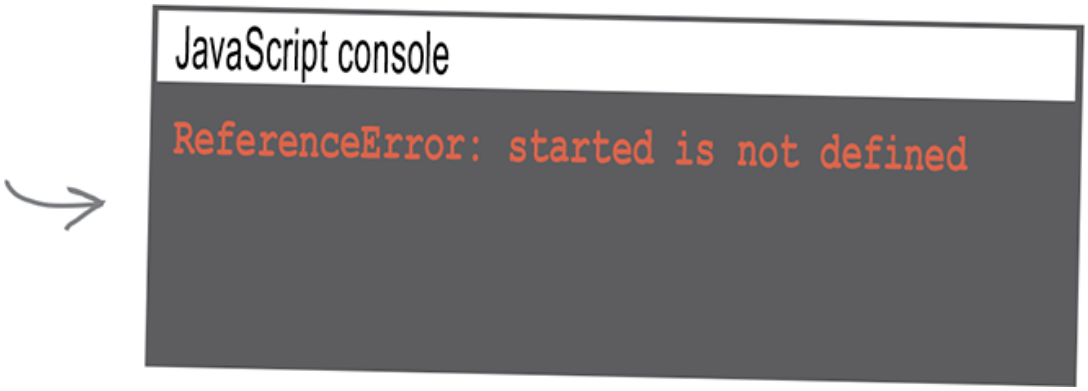
Let's take our new and improved `fiat` object for a test drive. Let's give it a good testing—we'll try to drive it before it's started, and then start, drive, and stop it. To do that, make sure you have the code for the `fiat` object typed into a simple HTML page ("carWithDrive.html"), including

the new methods `start`, `stop`, and `drive`, and then add this code below the object:

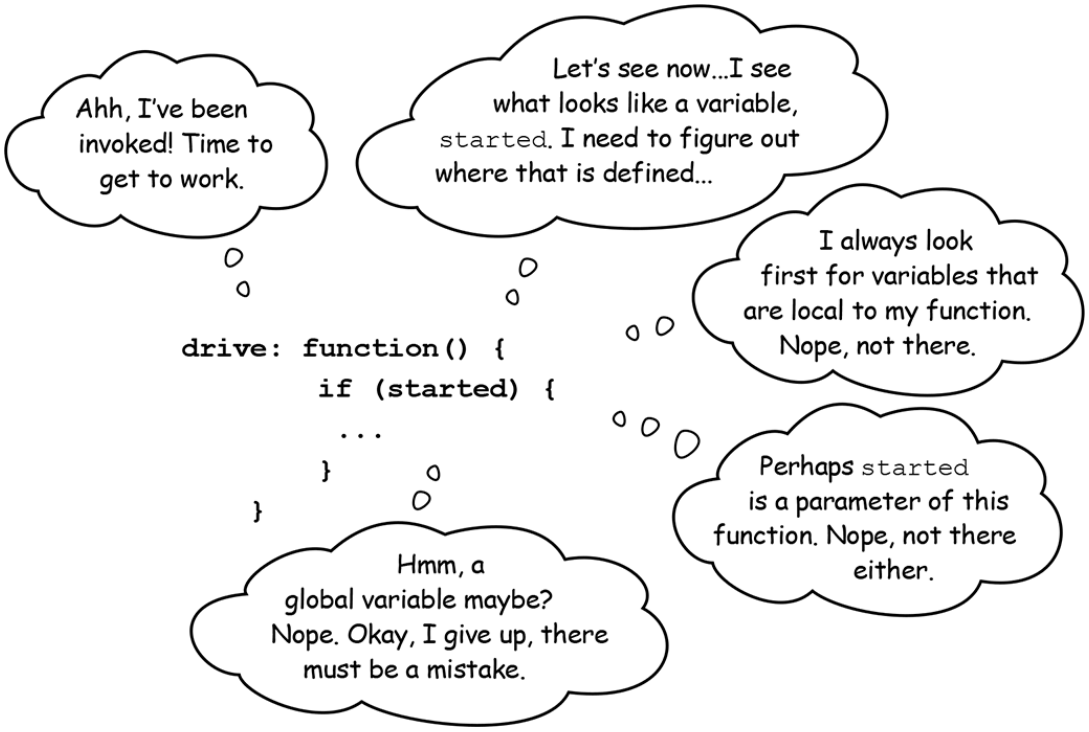
```
fiat.drive();  <- First, we'll try to drive the car, which should
fiat.start();  <- give us a message to start the car. Then we'll
fiat.drive();  <- start it for real, and we'll drive it. Finally,
fiat.stop();   <- when we're done, we'll stop the car.
```

Uh-oh, not so fast...

If you can't drive your Fiat, you're not alone. In fact, find your way to your JavaScript console and you're likely to see an error message similar to the one we got saying that `started` is not defined.



So, what's going on? Let's listen in on the `drive` method and see what's happening as we try to drive the car with `fiat.drive()`:

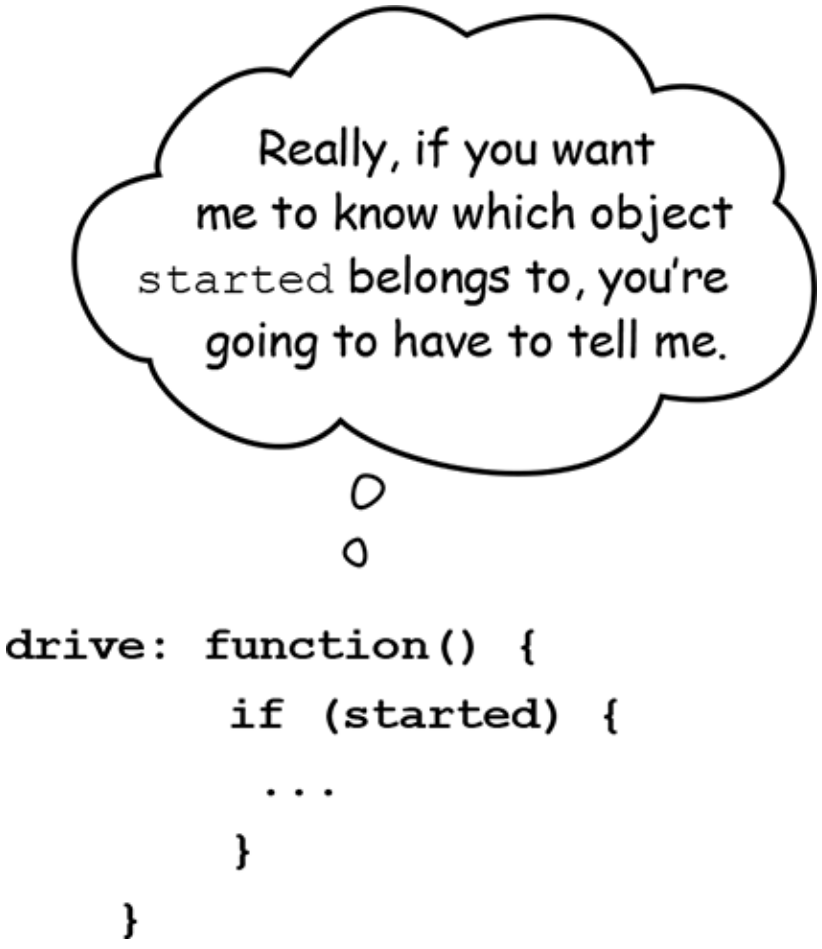


Why doesn't the drive method know about the started property?

Here's the conundrum: we've got references to the property `started` in the `fiat` object's methods, and normally when we're trying to resolve a variable in a function, that variable turns out to be a local variable, a parameter of the function, or a global variable. But in the `drive` method, `started` is none of those things; instead, it's a *property* of the `fiat` object.

Shouldn't this code just work, though? In other words, we wrote `started` in the `fiat` object; shouldn't JavaScript be smart enough to figure out we mean the `started` property?

Nope. As you can see, it isn't. How can that be?



Okay, here's the deal: what looks like a variable in the method is really a property of the object, but we aren't telling JavaScript which object. You might say to yourself, "Well, obviously we mean THIS object, this one right here! How could there be any confusion about that?" And, yes, we want the property of this very object. In fact, there's a keyword in JavaScript named `this`, and that is exactly how you tell JavaScript you mean *this object we're in*.

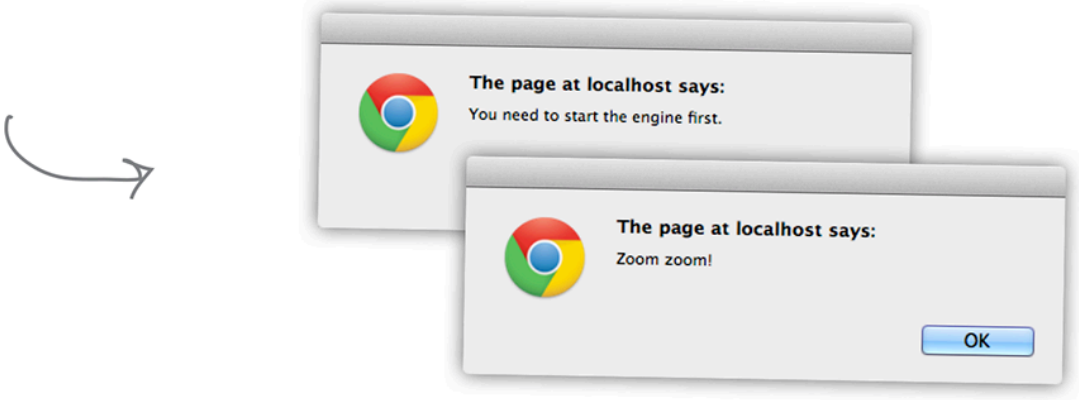
So, let's add the `this` keyword and get this code working:

```
let fiat = {
    make: "Fiat",
    // other properties are here, we're just saving space
    started: false,

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

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

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

Use this along with dot notation before each occurrence of the started property to tell the JavaScript interpreter you mean the property of THIS very object, rather than having JavaScript think you're referring to a variable.

A test drive with "this"

Go ahead and update your code, and take it for a spin! Here's what we got:



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.

```
let song = {
    name: "Walk This Way",
    artist: "Run-D.M.C.",
    minutes: 4,
    seconds: 3,
    genre: "80s",
    playing: false,

    play: function() {
        if (!playing) {
            this = true;
            console.log("Playing "
                + name + " by " + artist);
        }
    },

    pause: function() {
```

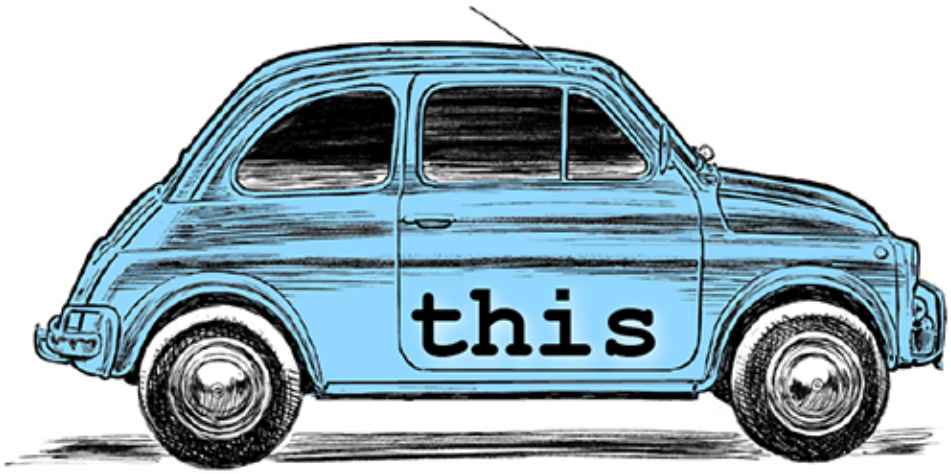
```
            if (playing) {
                this.playing = false;
            }
        }
    };

    this.play();
    this.pause();
```

NOTE

Go ahead and mark up the code right here...

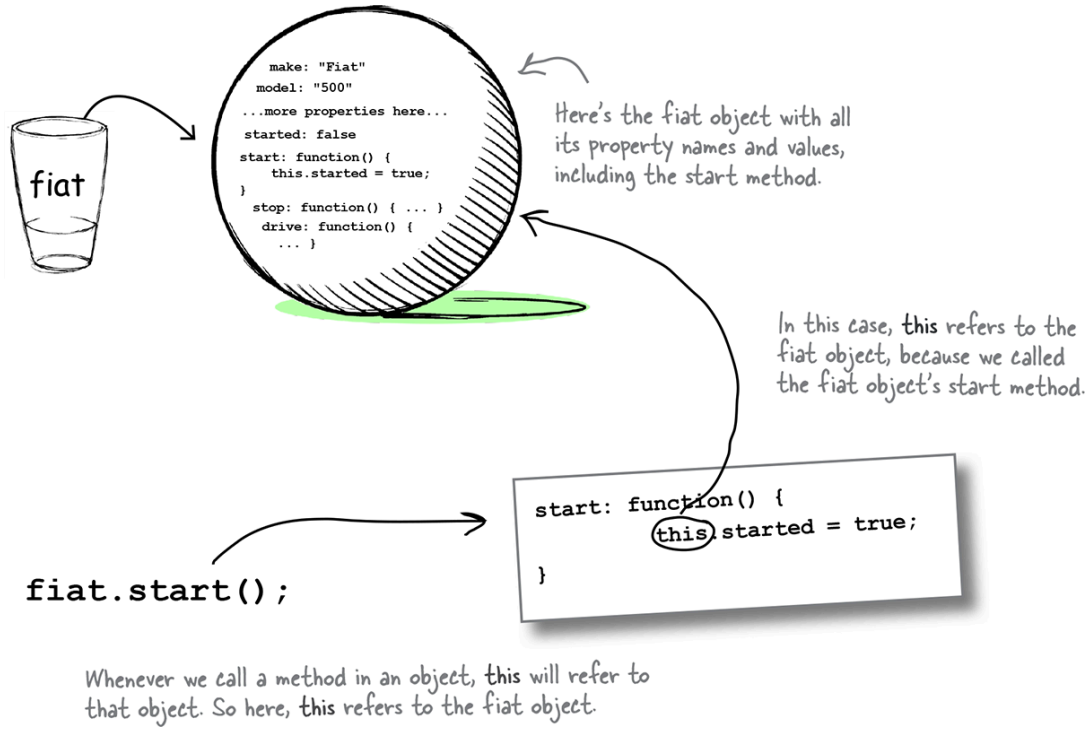
→ **Solution in [“BE the Browser Solution”]**

How “this” works



You can think of `this` like a variable that is assigned to the object whose method was just called. In other words, if you call the `fiat` object’s `start` method with `fiat.start()` and use `this` in the body of the `start` method, then `this` will refer to the `fiat` object. Let’s look more closely at what happens when we call the `start` method of the `fiat` object.

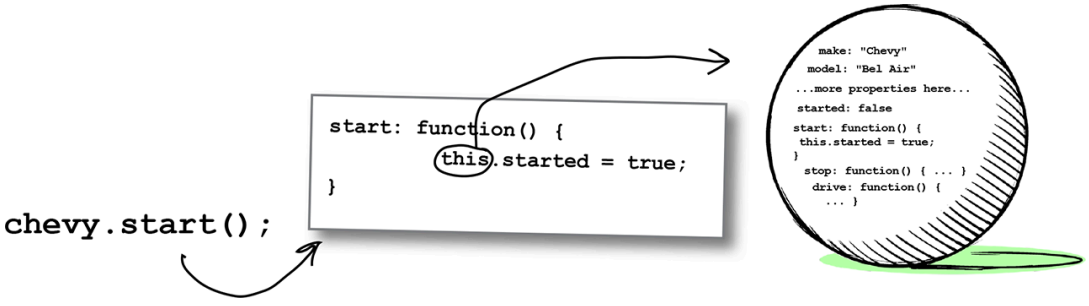
First, we have an object representing the Fiat car, which is assigned to the `fiat` variable:



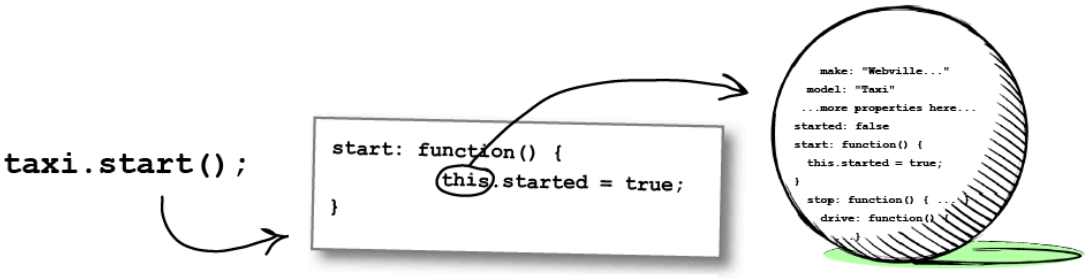
Then, when we call the `start` method, JavaScript takes care of assigning `this` to the `fiat` object.

The real key to understanding `this` is that whenever a method is called, in the body of that method you can count on `this` to be assigned to the *object whose method was called*. Just to drive the point home, let's try it on a few other objects...

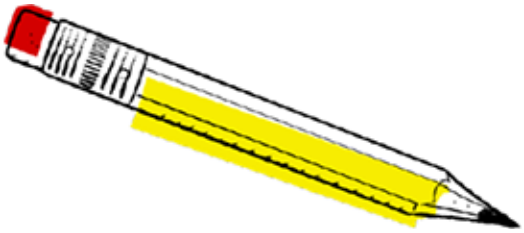
If you call the `chevy` object's `start` method, then `this` will refer to the `chevy` object in the body of the method.



And in the `start` method of the `taxi` object, `this` refers to the `taxi`.



SHARPEN YOUR PENCIL



Use your new **this** skills to help us finish this code. Check your answer at the end of the chapter.

```
let eightBall = { index: 0,
                  advice: ["yes", "no", "maybe", "not a chance"],
                  shake: function() {
                             this.index = _________.index + 1;
                             if (_________.index  >= _________.advice.length) {
                                _________.index = 0;
                             }
                  },
                  look: function() {
                             return _________.advice[_________.index];
                  }
};
eightBall.shake();
console.log(eightBall.look());
```

Repeat this sequence several times to test your code.

JavaScript console

```
no
maybe
not a chance
```

→ **Solution in [“Sharpen your pencil Solution”]**

Q: What's the difference between a method and a function?

A: A method is just a function that's been assigned to a property name in an object. You call functions using the function name, while you call methods using the object dot notation and the name of the property. You can also use the keyword **this** in a method to refer to the object whose method was called.

Q: I noticed that when using the "function" keyword within an object we don't give the function an explicit name. What happened to the function name?

A: Right. To call methods, we use the property name in the object rather than explicitly naming the function and using that name. For now, just take this as the convention we use, but later in the book we'll dive into the topic of anonymous functions (which is what we call functions that don't explicitly have names).

Q: Can methods have local variables, like functions can?

A: Yes. A method is a function. We just call it a method because it lives inside an object. So, a method can do anything a function can do precisely because a method is a function.

Q: So, you can return values from methods too?

A: Yes. What we said in the last answer!

Q: What about passing arguments to methods? Can we do that too?

A: Err, maybe you didn't read the answer two questions back? Yes!

Q: Can I add a method to an object after it's created, like I can with a property?

A: Yes. Think of a method as a function assigned to a property, so you can add a new one at any time:

```
// add a turbo method
car.engageTurbo =
       function() { ... };
```

Q: If I add a method like engageTurbo above, will the "this" keyword still work?

A: Yes. Remember, **this** is assigned the object whose method is called *at the time it is called.*

Q: When is the value of "this" set to the object? When we define the object, or when we call the method?

A: The value of **this** is set to the object when you call the method. So, when you call fiat.start(), **this** is set to fiat, and when you call chevy.start(), **this** is set to chevy. It *looks* like **this** is set when you define the object, because in fiat.start, **this** is always set to fiat, and in chevy.start, **this** is always set to chevy. But as you'll see later, there is a good reason the value of **this** is set when you call the method and not when you define the object. This is an important point we'll be coming back to a few different times.

BRAIN POWER

If you copy the start, stop, and drive methods into the chevy and cadi objects we created earlier, what do you have to change to make the methods work correctly?

NOTE

Answer: Nothing! this refers to “this object,” the one whose method we’re calling.

EXERCISE

It's time to get the whole fleet up and running. Add the drive method to each car object. When you've done that, add the code to start, drive, and stop each of them. Check your answers at the end of the chapter.

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

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

```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341
};
```

Don't forget to add a comma after mileage when you add the new properties!

Add the started property and the methods to each car. Then use the code below to give them a test drive.

```
started: false,

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

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

drive: function() {
   if (this.started) {
     alert(this.make + " " +
           this.model + " goes zoom zoom!");
   } else {
     alert("You need to start the engine first.");
   }
}
```

We improved the drive method just a bit, so make sure you get this new code.

Throw this code after the car object definitions to give them all a test drive.

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

Solution in "Exercise Solution"

Ah, good eye.

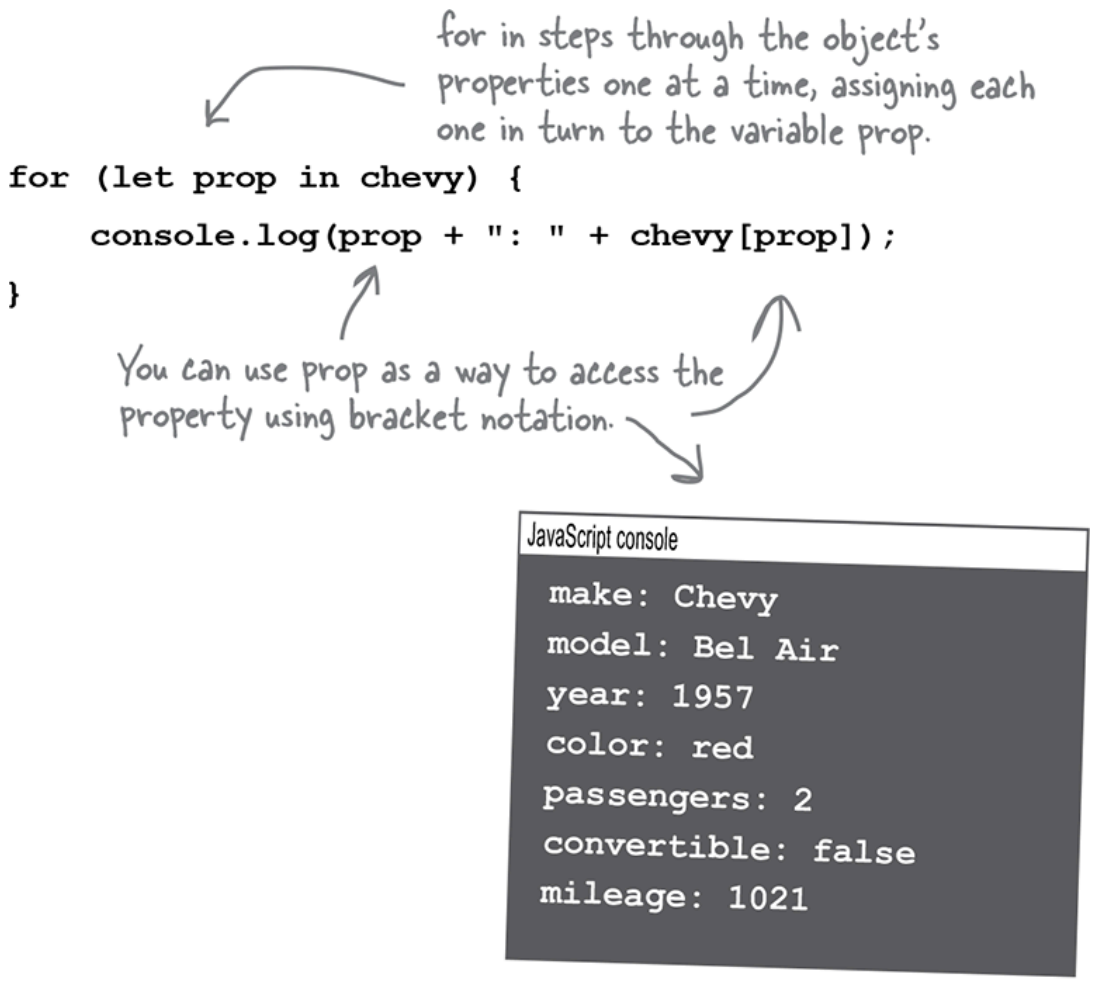
Yes, when we copy `start`, `stop`, and `drive` into each car object we're definitely duplicating code. Unlike the other properties, which have values that depend on which car object they're in, the methods are the same for all of the objects.

Now, if you're saying "Great, we're reusing code!"...not so fast. Sure, we're reusing it, but we're doing that by copying it, not just once, but many times! What happens now if you want `drive` to work differently? Then you've got to redo the code in every single car object. Not good. Not only is that a waste, it can be error-prone.

But you're identifying a problem even larger than simple copying and pasting; we're assuming that just because we put the same properties in all our objects, that makes them all car objects. What if you accidentally leave out the `mileage` property from one of the objects—is it still a car?

These are all real problems with our code so far, and we're going to tackle all these questions in an upcoming chapter on advanced objects, where we'll talk about some techniques for properly reusing the code in your objects.

One thing you can do is iterate through an object's properties. To do that, you can use a form of iteration we haven't seen yet called `for in`. The `for in` iterator steps through every property of an object in an arbitrary order. Here's how you could display all the properties of the `chevy` object:



This brings up another topic: there's another way to access properties. Did you catch the alternative syntax we just used to access the properties of the `chevy` object? As it turns out, you've got two options when accessing a property of an object. You already know dot notation:

```
chevy.color
```

We just use the object name followed by a dot and a property name.

But there's another way, **bracket notation**, which looks like this:

```
chevy["color"]
```

Here we use the object name followed by brackets that enclose a property name in quotes.

Looks a bit like how we access array items.

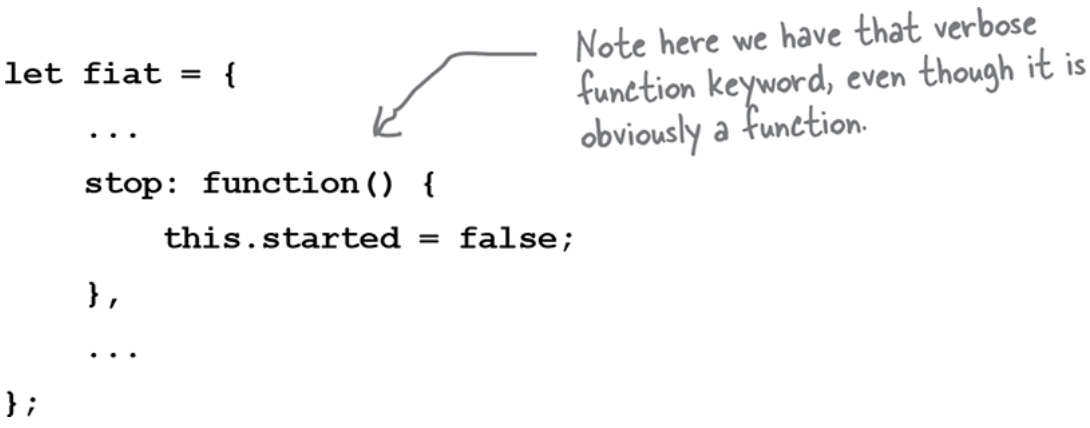
The thing to know about these two forms is that they are equivalent and do the same thing. The only difference to be aware of is that bracket notation sometimes allows a little more flexibility, because you can make the property name an expression, like this:

```
chevy["co" + "lor"]
```

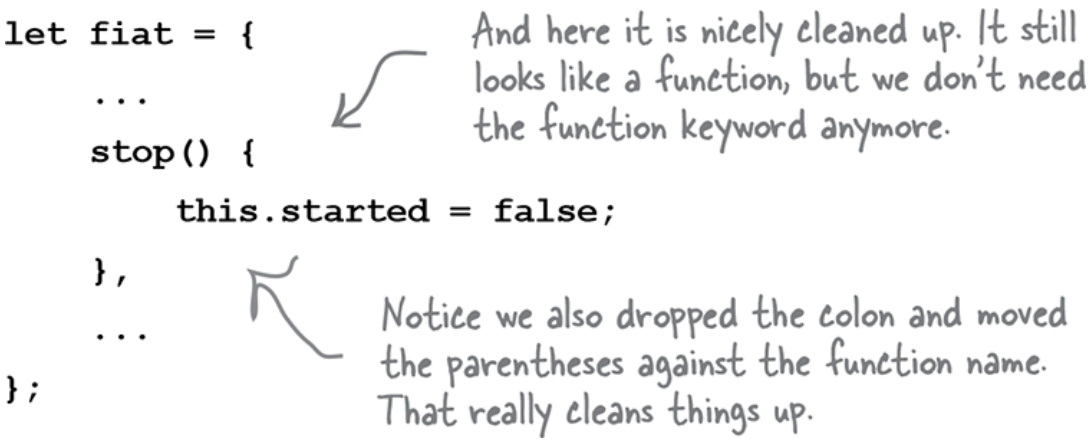
As long as the expression evaluates to a property name represented by a string, you can put any expression you want inside the brackets.

Method shorthand

While we're on the topic of syntax...there's a shorthand for declaring methods that we haven't told you about yet. You can drop the keyword `function`, and instead of writing your method declarations like this:



you can write them like this:



Now, you'll still call the method in the same way:

```
fiat.stop();
```

and this new syntax doesn't affect the method's semantics in any way.

EXERCISE

Go ahead and update your code for the cadi, chevy, and taxi objects to use the method shorthand. Make sure your code still runs, and check your solution at the end of the chapter before you go on. We'll use the shorthand for methods for the rest of the chapter.

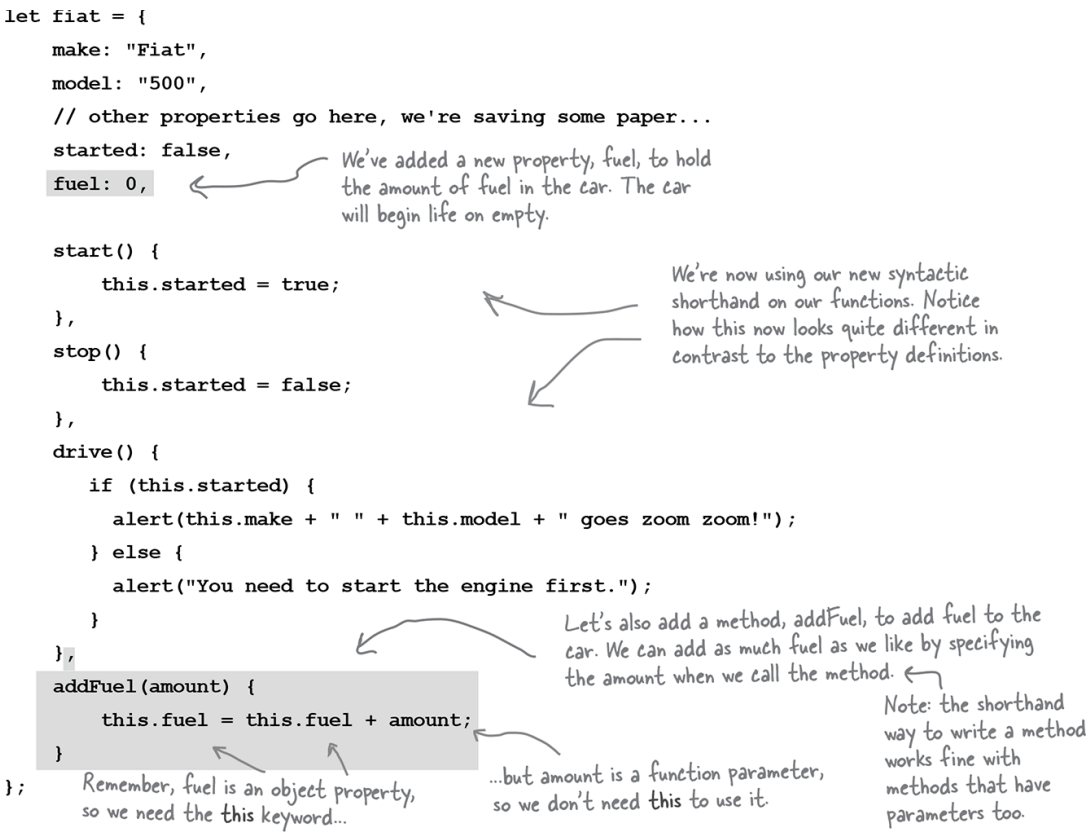
→ **Solution in "Exercise Solution"**

How behavior affects state

Adding some gas-o-line

Objects have *state* and *behavior*. An object's properties allow us to keep state about the object—like its fuel level, its current temperature, or, say, the current song that is playing on the radio. An object's methods allow it to have behavior—like starting a car, turning up the heat, or fast-forwarding the playback of a song. Have you also noticed these two *interact*? Like, we can't start a car if it doesn't have fuel, and the amount of fuel should get reduced as we drive the car. Kinda like real life, right?

Let's play with this concept a little more by giving our cars some fuel, and then we can start to add interesting behavior. To add fuel, we'll add a new property, `fuel`, and a new method, `addFuel`. The `addFuel` method will have a parameter, `amount`, which we'll use to increase the amount of fuel in the `fuel` property. Let's add these properties to the `fiat` object:



Now let's affect the behavior with the state

Now that we have fuel, we can start to implement some interesting behaviors. For instance, if there's no fuel, we shouldn't be able to drive the car! So, let's start by tweaking the `drive` method a bit to check the fuel level to make sure we've got some, and then we'll subtract one from `fuel` each time the car is driven. Here's the code to do that:

```
let fiat = {
    // other properties and methods here...
    drive() {
        if (this.started) {
            if (this.fuel > 0) {
                alert(this.make + " " +
                    this.model + " goes zoom zoom!");
               this.fuel = this.fuel - 1;
            } else {
                alert("Uh-oh, out of fuel.");
                this.stop();
            }
        } else {
            alert("You need to start the engine first.");
        }
    },
    addFuel(amount) {
        this.fuel = this.fuel + amount;
    }
};
```

Now we can check to make sure there's fuel before we drive the car. And, if we can drive the car, we should reduce the amount of fuel left each time we drive.

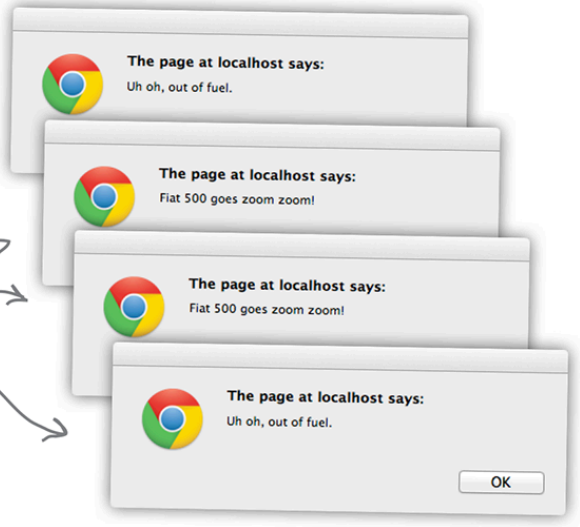
If there's no fuel left, we display a message and stop the engine. To drive the car again, you'll have to add fuel and restart the car.

Gas up for a test drive

Go ahead and update your code, and take it for a spin! Here's what we got with the following test code:

```
fiat.start();
fiat.drive();
fiat.addFuel(2);
fiat.start();
fiat.drive();
fiat.drive();
fiat.drive();
fiat.stop();
```

First, we tried to drive the car with no fuel; then we added some fuel and drove it until we ran out of fuel again! Try adding your own test code and make sure it works like you think it should.



EXERCISE

We still have some more work to do to fully integrate the fuel property into the car. For instance, should you be able to start the car if there's no fuel? Check out the start method:

```
start() {
    this.started = true;
}
```

It certainly looks like we can.

Help us integrate the fuel property into this code by checking the fuel level before the car is started. If there's no fuel and the start method is called, let the driver know with a handy alert like `"The car is on empty, fill up before starting!"` Rewrite the start method below, and then add it to your code and test it. Check your answer at the end of the chapter before you go on.

NOTE

Your code here.

→ **Solution in "Exercise Solution"**

Take a look at all the fiat code. Are there other places you could use the fuel property to alter the car's behavior (or create behavior to modify the fuel property)? Jot down your ideas below.

Congrats on your first objects!


I really am begining to think in terms of objects now.

You've made it through the first objects chapter, and you're ready to move forward. Remember how you began with JavaScript? You were thinking of the world in terms of low-level numbers and strings and statements and conditionals and for loops and so on. Look how far you've come. You're starting to think at a higher level, and in terms of objects and methods. Just look at this code:

```
fiat.addFuel(2);
fiat.start();
fiat.drive();
fiat.stop();
```

It's so much easier to understand what's going on in this code, because it describes the world as a set of objects with state and behavior.

And this is just the beginning. You can take it so much further, and we will. Now that you know about objects, we're going to keep developing your skills to write truly object-oriented code using even more features of JavaScript and quite a few best practices (which become oh-so-important with objects).

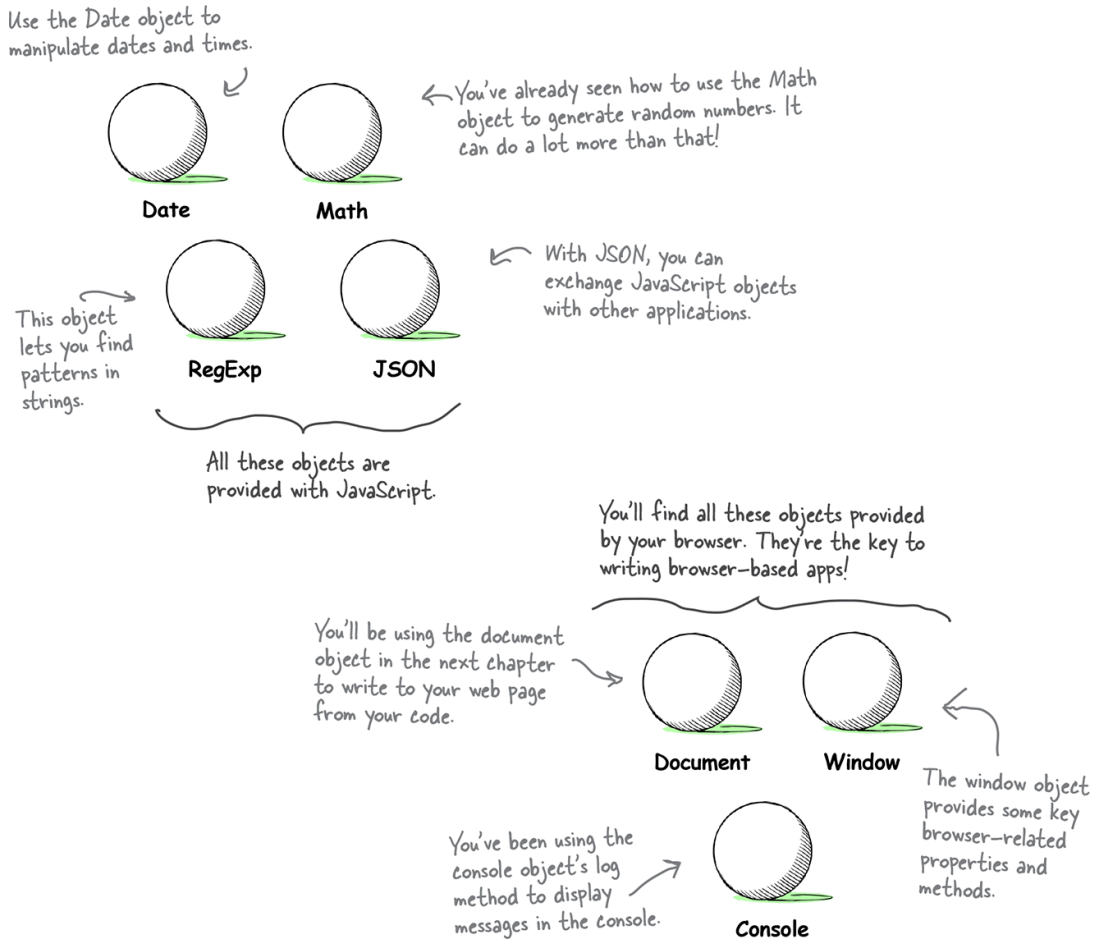
But there's one more thing you should know, before you leave this chapter...

Guess what? There are objects all around you!

(And they'll make your life easier)

Now that you know a bit about objects, a whole new world is going to open up for you because JavaScript provides you with lots of objects (for doing math computations, manipulating strings, and creating dates and times, to name a few) that you can use in your own code. JavaScript also provides some really key objects that you need to write code for the browser (and we're going to take a look at one of those objects in the next

chapter). For now, take a second to get acquainted with a few more of these objects; we'll touch on these throughout the rest of the book:



This week's interview:

In Object's own words...

Head First: Welcome Object, it's been a fascinating chapter. It's a real head-spinner thinking about code as objects.

Object: Oh, well...we've only just begun.

Head First: How so?

Object: An object is a set of properties, right? Some of those properties are used to keep the state of the object, and some are actually functions—or rather, methods—that give an object behavior.

Head First: I'm with you so far. I hadn't actually thought about the methods being properties too, but I guess they are just another name and value, if you can call a function a value?

Object: Oh, you can! Believe me, you can. In fact, that's a huge insight, whether you realize it or not. Hold on to that thought; I'm guessing there's a lot in store for you on that topic.

Head First: But you were saying...

Object: So, you've looked at these objects with their properties and you've created lots of them, like a bunch of different types of cars.

Head First: Right...

Object: But it was very *ad hoc*. The real power comes when you can create a template of sorts, something that can basically stamp out uniform objects for you.

Head First: Oh, you mean objects that all have the same type?

Object: Sort of...as you'll see, the concept of type is an interesting one in JavaScript. But you're on the right track. You'll see that you have real power when you can start to write code that deals with objects of the same kind. Like, you could write code that deals with vehicles, and you wouldn't have to care if they were bicycles, cars, or buses. That's power.

Head First: It certainly sounds interesting. What else do we need to know to do that?

Object: Well, you have to understand objects a little better, and you need a way to create objects of the same kind.

Head First: We just did that, didn't we? All those cars?

Object: They're sort of the same kind by convention, because you happened to write code that creates cars that look alike. In other words, they have the same properties and methods.

Head First: Right, and in fact we talked a little about how we are replicating code across all those objects, which is not necessarily a good thing in terms of maintaining that code.

Object: The next step is to learn how to create objects that really are all guaranteed to be the same, and that make use of the same code—code that's all in one place. That's getting into how to design object-oriented code. And you're pretty much ready for that now that you know the basics.

Head First: I'm sure our readers are happy to hear that!

Object: But there are a few more things about objects to be aware of.

Head First: Oh?

Object: There are many objects already out there in the wild that you can use in your code.

Head First: Oh? I hadn't noticed, where?

Object: How about `console.log`. What do you think `console` is?

Head First: Based on this discussion, I'm guessing it's an object?

Object: BINGO. And `log`?

Head First: A property...err, a method?

Object: BINGO again. And what about `alert`?

Head First: I haven't a clue.

Object: It has to do with an object, but we'll save that for a bit later.

Head First: Well, you've certainly given us a lot to think about, Object, and I'm hoping you'll join us again.

Object: I'm sure we can make that work.

Head First: Great! Until next time then.

Crack the Code Challenge

In his quest for world domination, Dr. Evel has accidentally exposed an internal web page with the current passcode to his operation. With the passcode, we can finally get the upper hand. Of course, as soon as Dr. Evel discovered the page was live on the internet, he quickly took it down. Luckily, our agents made a record of the page. The only problem is, our agents don't know HTML or JavaScript. Can you help figure out the access code using the code below? Keep in mind, if you are wrong, it could be quite costly to Queen and Country.

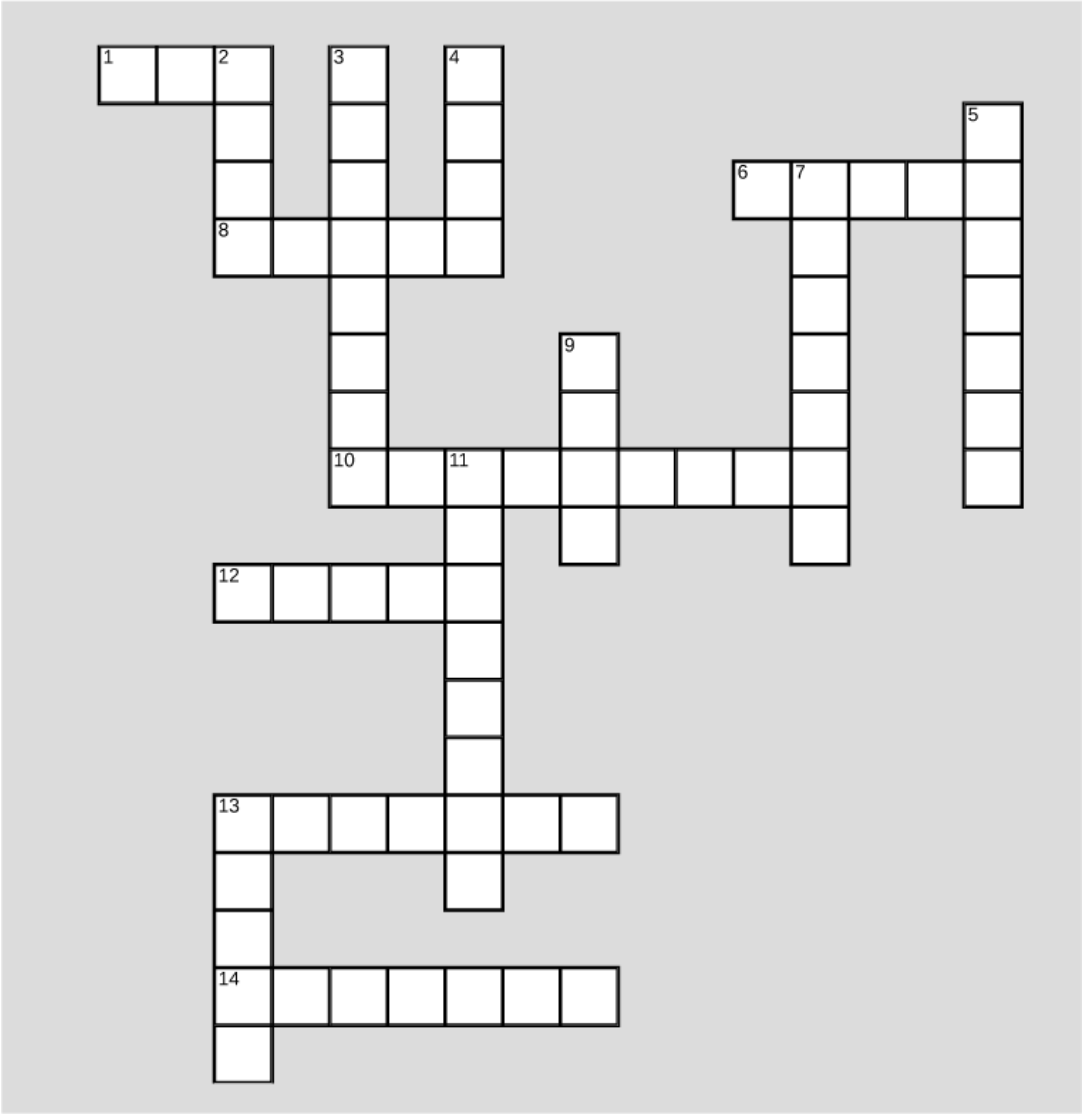
If you skipped the last page, go back and do the challenge. It is vitally important to Chapter 6!

BULLET POINTS

- An object is a collection of **properties**.
- To access a property, use **dot notation**: the name of the variable containing the object, then a period, then the name of the property.
- You can also access a property using **bracket notation**: the name of the variable containing the object, then the name of the property as a string in brackets.
- You can add new properties to an object at any time, by assigning a value to a new property name.
- You can also delete properties from objects, using the **delete** operator.
- Unlike variables that contain primitive values, like strings, numbers, and booleans, a variable can't actually contain an object. Instead, it contains a **reference** to an object. We say that objects are "reference variables."
- When you pass an object to a function, the function gets a copy of the reference to the object, not a copy of the object itself. So, if you change the value of one of the object's properties, it changes the value in the original object.
- Object properties can contain functions. When a function is in an object, we call it a **method**.
- You call a method by using **dot notation**: the object name, a period, and the property name of the method, followed by parentheses.
- A method is just like a function except that it is in an object.
- You can pass arguments to methods, just like you can to regular functions.
- When you call an object's method, the keyword **this** in the method refers to the object whose method you are calling.
- To access an object's properties in an object's method, use dot notation or bracket notation, with **this** in place of the object's name.
- In object-oriented programming, we think in terms of objects rather than procedures.
- An object has both **state** (properties) and **behavior** (methods). State can affect behavior, and behavior can affect state.
- Objects **encapsulate**, or hide, the complexity of the state and behavior of that object.
- A well-designed object has methods that abstract the details of how to get work done with the object, so you don't have to worry about it.

- Along with the objects you create, JavaScript has many built-in objects that you can use. We'll be using many of these built-in objects throughout the rest of the book.

How about a crossword object? It's got lots of clue properties that will help objects stick in your brain.



ACROSS

1. To access the properties of an object we use _____ notation.

6. Object references are passed by _____ to functions, just like primitive variables.

8. car and dog objects can have both _____ and behavior.

10. When you assign an object to a variable, the variable contains a _____ to the object.

12. The name and value of a property of an object are separated by a _____.

13. The method log is a property of the _____ object.

14. _____ can have local variables and parameters, just like regular functions can.

DOWN

2. The Fiat wouldn't start because we weren't using _____ to access the started property.

3. An object gets _____ with its methods.

4. We used a _____ property to represent the make of a car object.

5. this is a _____, not a regular variable.

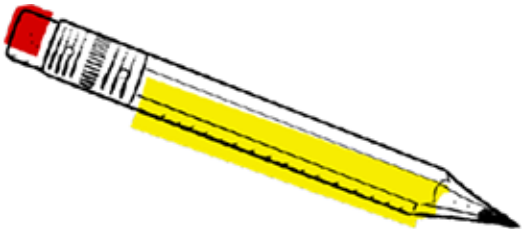
7. The _____ method affects the state of the car object, by adding to the amount of fuel in the car.

9. We usually use one _____ for property names.

11. Shorthand notation for methods eliminates the colon and the _____ keyword.

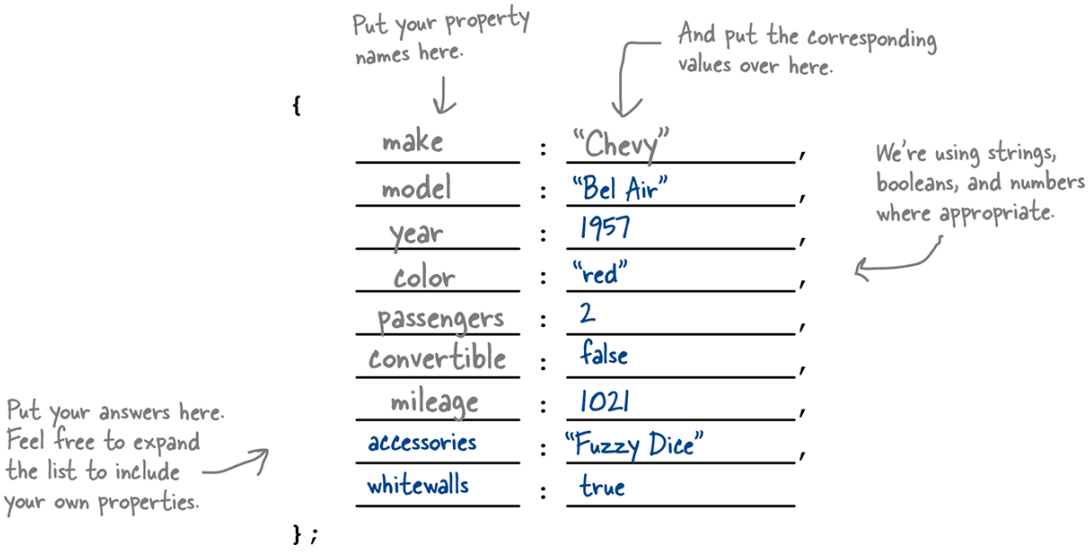
13. Don't forget to use a _____ after each property value except the last one.

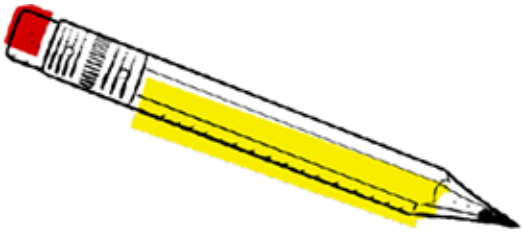
Solution in "JavaScript cross Solution"



From [“Sharpen your pencil”]

We’ve started making a table of property names and values for a car. Can you help complete it? Here’s our solution.





From "Sharpen your pencil"

Use your new **this** skills to help us finish this code. Here's our solution.

```
let eightBall = { index: 0,
                  advice: ["yes", "no", "maybe", "not a chance"],
                  shake: function() {
                       this.index = this.index + 1;
                  if (this.index  >= this.advice.length) {
                     this.index = 0;
                  }
            },
            look: function() {
                       return this.advice[this.index];
            }

};
eightBall.shake();
console.log(eightBall.look());
```

Repeat this sequence several times to test your code.

JavaScript console

```
no
maybe
not a chance
```

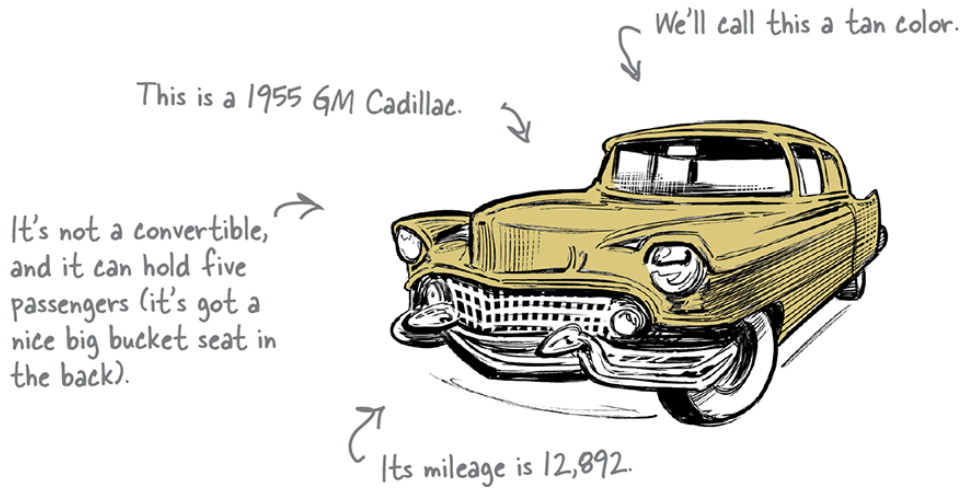
EXERCISE SOLUTION

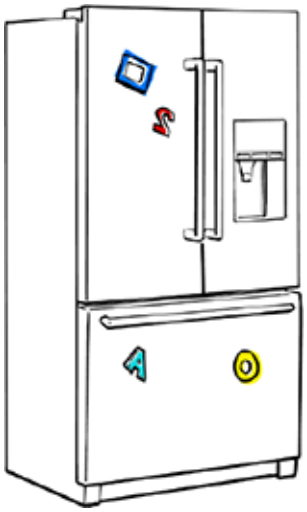
From “Exercise”

You don’t have to be stuck with just one object. The real power of objects (as you’ll see soon enough) is having lots of objects and writing code that can operate on whatever object you give it. Try your hand at creating another object from scratch...another car object. Go ahead and work out the code for your second object. Here’s our solution.

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

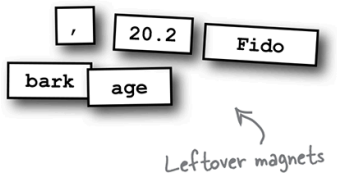
Here are the properties for the Cadillac.

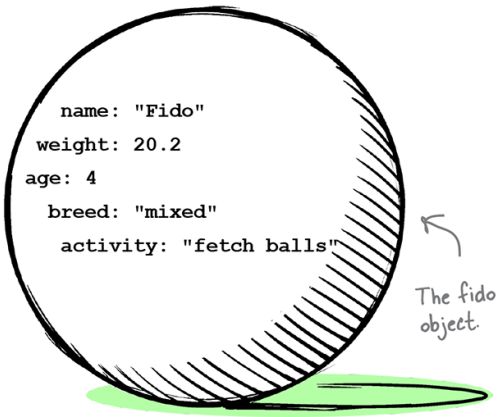




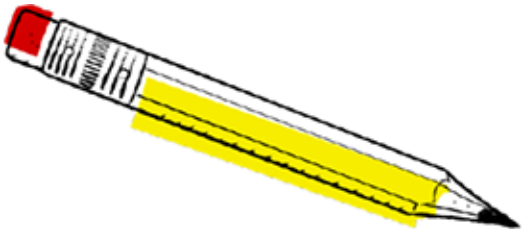
From "Object Magnets"

Practice your object creating and dot notation skills by completing the code below with the fridge magnets. Be careful, some extra magnets got mixed in! Here's our solution.






```
let fido = {
    name: "Fido",
    weight: 20.2,
    age: 4,
    breed: "mixed",
    activity: "fetch balls"
};
let bark;
if (fido.weight > 20) {
   bark = "WOOF WOOF";
} else {
   bark = "woof woof";
}
let speak = fido.name + " says " + bark + " when he wants to " + fido.activity;
console.log(speak);
```



From [“Sharpen your pencil”]

Your turn. Here are three more car objects; what is the result of passing each car to the prequal function? Work out the answers by hand, and then write the code to check your answers. Here’s our solution.

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

prequal(cadi);
```

false

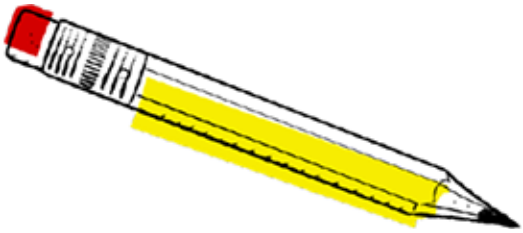
Write the value of prequal here.

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000
};

prequal(fiat);
```

false

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

prequal(chevy);
```

true



From "Sharpen your pencil"

You've been given a super-secret file and two functions that allow access to get and set the contents of the file, but only if you have the right password. The first function, getSecret, returns the contents of the file if the password is correct and logs each attempt to access the file. The second function, setSecret, updates the contents of the file and resets the access tracking back to 0. It's your job to fill in the blanks below to complete the JavaScript, and test your functions. Here's our solution.

```
function getSecret(file, secretPassword) {
    file.opened = file.opened + 1;
    if (secretPassword == file.password) {
        return file.contents;
    }
    else {
        return "Invalid password! No secret for you.";
    }
}
function setSecret(file, secretPassword, secret) {
    if (secretPassword == file.password) {
        file.opened = 0;
        file.contents = secret;
    }
}

let superSecretFile = {
    level: "classified",
    opened: 0,
    password: 2,
    contents: "Dr. Evel's next meeting is in Detroit."
};
let secret = getSecret(superSecretFile, 2);
console.log(secret);

setSecret(superSecretFile, 2, "Dr. Evel's next meeting is in Philadelphia.");
secret = getSecret(superSecretFile, 2);
console.log(secret);
```

The superSecretFile object is passed into the getSecret function and gets the parameter name file. So, we need to make sure we use the object name, file, and dot notation to access the object's properties, like opened and password.

We can pass the supserSecretFile object to the getSecret and setSecret functions.

From **“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. Here’s our solution.

```
let song = {
    name: "Walk This Way",
    artist: "Run-D.M.C.",
    minutes: 4,
    seconds: 3,
    genre: "80s",
    playing: false,
    play: function() {
        if (!this.playing) {
            this.playing = true;
            console.log("Playing "
                + this.name + " by " + this.artist);
        }
    },

    pause: function() {
        if (this.playing) {
            this.playing = false;
        }
    }
};

this song.play();
this song.pause();
```

We were missing a this here.

And missing the playing property name here.

We need to use this to access both these properties, too.

Again here, we need this to access the playing property.

We don't use this outside of a method; we call an object using the object's variable name.

EXERCISE SOLUTION

From "Exercise"

It's time to get the whole fleet up and running. Add the drive method to each car object. When you've done that, add the code to start, drive, and stop each of them. Here's our solution.

```
let cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892,
    started: false,
    start: function() {
        this.started = true;
    },
    stop: function() {
        this.started = false;
    },
    drive: function() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

```
let 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) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

```
let 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() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Make sure you add a comma after any new properties you add.

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

chevy.start();
chevy.drive();
chevy.stop();

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

We copied and pasted the code into each object, so every car has the same properties and methods.

Now we can start, drive, and stop each of the cars, using the same method names.

From "Exercise"

Go ahead and update your code for the cadi, chevy, and taxi objects to use the method shorthand. Here's our solution.

```
let cadi = {
    make: "GM",
    model: "Cadillac",
    year: 1955,
    color: "tan",
    passengers: 5,
    convertible: false,
    mileage: 12892,
    started: false,
    start() {
        this.started = true;
    },
    stop() {
        this.started = false;
    },
    drive() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

```
let chevy = {
    make: "Chevy",
    model: "Bel Air",
    year: 1957,
    color: "red",
    passengers: 2,
    convertible: false,
    mileage: 1021,
    started: false,
    start() {
        this.started = true;
    },
    stop() {
        this.started = false;
    },
    drive() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

```
let taxi = {
    make: "Webville Motors",
    model: "Taxi",
    year: 1955,
    color: "yellow",
    passengers: 4,
    convertible: false,
    mileage: 281341,
    started: false,
    start() {
        this.started = true;
    },
    stop() {
        this.started = false;
    },
    drive() {
        if (this.started) {
            alert(this.make + " " +
                  this.model + " goes zoom zoom!");
        } else {
            alert("You need to start the engine first.");
        }
    }
};
```

Make sure you add a comma after any new properties you add.

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

chevy.start();
chevy.drive();
chevy.stop();

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

We copied and pasted the code into each object, so every car has the same properties and methods.

Now we can start, drive, and stop each of the cars, using the same method names.

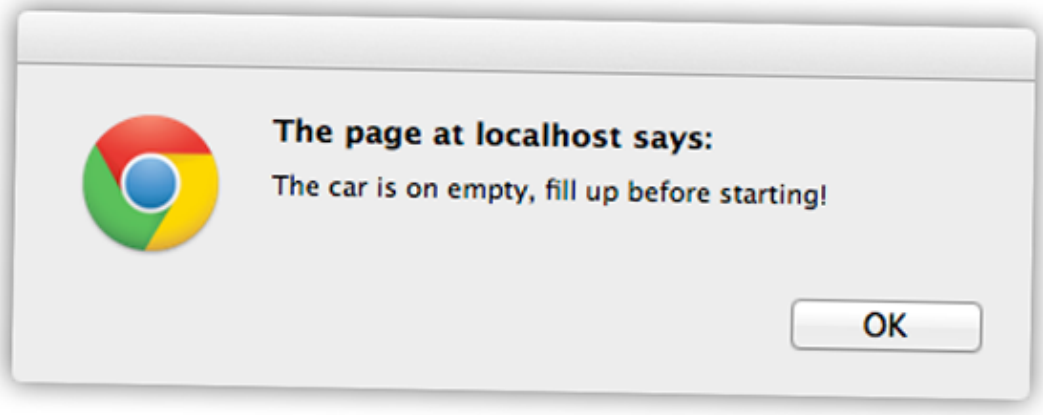
EXERCISE SOLUTION

From "Exercise"

We still have some more work to do to fully integrate the fuel property into the car. For instance, should you really be able to start the car if there's no fuel? Help us integrate the fuel property into this code by checking the fuel level before the car is started. If there's no fuel and the start method is called, let the driver know with a handy alert like `"The car is on empty, fill up before starting!"` Rewrite the start method below, and then add it to your code and test it. Here's our solution.

```
let fiat = {
    make: "Fiat",
    model: "500",
    year: 1957,
    color: "Medium Blue",
    passengers: 2,
    convertible: false,
    mileage: 88000,
    fuel: 0,
    started: false,

    start() {
        if (this.fuel == 0) {
            alert("The car is on empty, fill up before starting!");
        } else {
            this.started = true;
        }
    },

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

```
    drive() {
        if (this.started) {
            if (this.fuel > 0) {
                alert(this.make + " " +
                    this.model + " goes zoom zoom!");
                this.fuel = this.fuel - 1;
            } else {
                alert("Uh-oh, out of fuel.");
                this.stop();
            }
        } else {
            alert("You need to start the engine first.");
        }
    },
    addFuel(amount) {
        this.fuel = this.fuel + amount;
    }
};
```



From [“JavaScript cross”]

How about a crossword object? It’s got lots of clue properties that will help objects stick in your brain. Here’s our solution.