Riverside JS Workshop
JavaScript Skills Ladder, Lesson 1.9

Introduction to JavaScript Objects

Lesson Objectives:

When you complete this course, you will be able to:

- *create your own objects.*
- *store objects within an array.*
- *use arrays as object property values.*
- *use the constructor function to create objects.*
- *change the value of a property in an object after by assigning it a new value.*
- *use objects to collect global variables.*

We've mentioned *objects* a few times in this course so far; for instance, you know that the browser represents the web page as a collection of objects in a tree structure that we refer to as the *Document Object Model* (or DOM). You also know that elements are objects. You might have even picked up that objects have *properties*. Now it's time to look more deeply at what an object actually is, and even create some of your own!

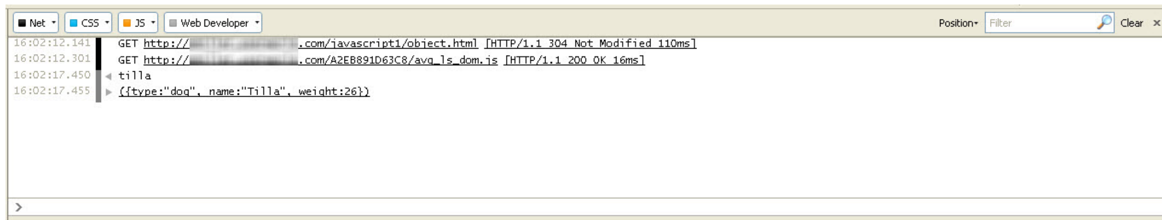
Objects and Collections of Properties

An object is a *collection of properties*. Let's take a look at an example.

CODE TO TYPE:

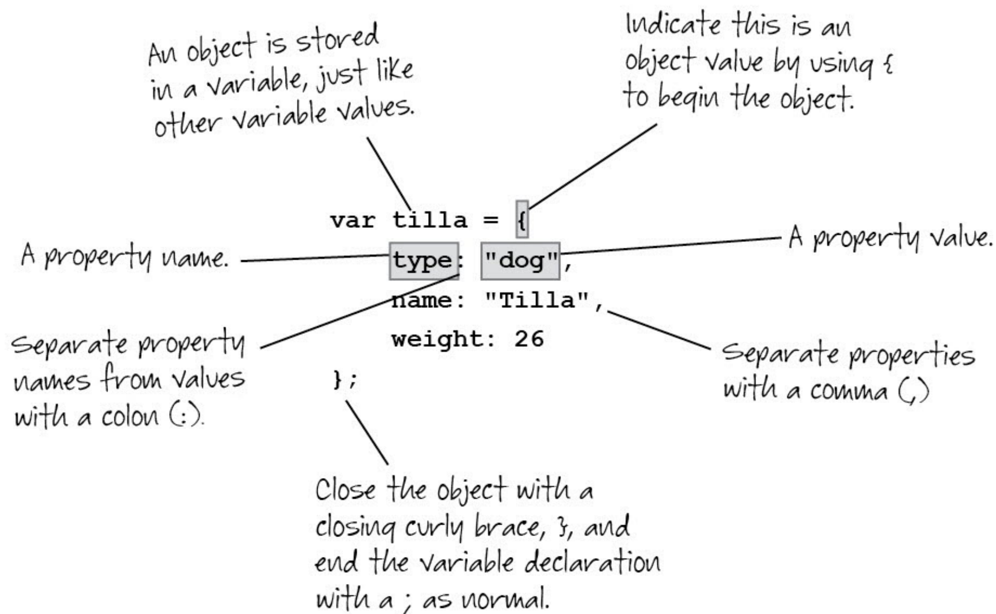
```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    var tilla = {
      type: "dog",
      name: "Tilla",
      weight: 26
    };
  </script>
</head>
<body>
</body>
</html>
```

Save it in your work folder as *object.html*, and open it in a browser. You won't see anything (because there's nothing in the page). Open the developer console. You should see something like this:



Let's take a closer look at what you just did. Notice that you declare an object just like you would a variable, by starting with the `var` keyword and giving your object a variable name, in this case, `tilla`. You initialize your object variable to an object value. You know it's an object because of the curly braces {}. Inside the braces are a collection of *properties*. You can think of each property as being similar to a variable declaration, except, of course, the syntax is different. The *property name* is on the left, then a colon, and then the *property value*. So in this case, you added the property name `type` to the object `tilla`, and this property has the string value "dog." Similarly, you added `name` and `weight` properties. Notice that the `type` and `name` properties have string values, while the `weight` property has an integer number value. Also notice that you separate each property name/value pair with a comma, except for the last one. Finally, notice that we end the object declaration and initialization with a semicolon, as usual when we're declaring a variable.

Here's a review :



Let's add another object:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    var tilla = {
      type: "dog",
      name: "Tilla",
      weight: 26
    };

    var pickles = {
      type: "cat",
      name: "Pickles",
      weight: 7
    };
</script>
</head>
<body>
</body>
</html>
```

This object is similar to `tilla`. It's got the `type`, `name`, and `weight` properties, except of course the values of the properties are different, and the variable name is `pickles` instead of `tilla`. Try viewing `pickles` in the console like we did earlier with `tilla`.

Accessing Object Properties

So, now you've got a couple of objects; what can you do with them?

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>

    window.onload = init;

    function init() {
      var tilla = {
        type: "dog",
```

```
        name: "Tilla",
        weight: 26
      };

      var pickles = {
        type: "cat",
        name: "Pickles",
        weight: 7
      };

      var div = document.getElementById("pets");
      div.innerHTML =
          "My " + tilla.type + " is named " + tilla.name +
          " and she weighs " + tilla.weight + " pounds.";
    }

  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it, and open it in a browser. You should now see your web page update with content that shows the text "My dog is named Tilla and she weighs 26 pounds." Notice that we added a `<div>` element to the body of the page, and we're updating that element with values from the tilla object.

To access a property in an object, you use *dot notation*. For instance, to access the type property of the tilla object, you write `tilla.type`. That is, you write the variable name of the object, and then a dot, and then the property name. The result of this expression is the value of the property, in this case "dog."

You access the `name` and `weight` properties in the same way, using `tilla.name` and `tilla.weight`, which have the values "Tilla" and 26, respectively. In this example, we used the object values to create a string and then updated the page by setting the content of the "pets" `<div>` to that string. Now, try writing a string using the `pickles` object instead.

Storing Objects in an Array

You can store objects in an array, just like you can other values:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
```

```
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var tilla = {
        type: "dog",
        name: "Tilla",
        weight: 26
      };

      var pickles = {
        type: "cat",
        name: "Pickles",
        weight: 7
      };

      var div = document.getElementById("pets");
      div.innerHTML =
          "My " + tilla.type + " is named " + tilla.name +
          " and she weighs " + tilla.weight + " pounds.";
      var pets = [tilla, pickles];
      for (var i = 0; i < pets.length; i++) {
        var pet = pets[i];
        if (pet.type == "dog") {
          div.innerHTML += pet.name + " says Woof! <br>";
        }
        else if (pet.type == "cat") {
          div.innerHTML += pet.name + " says Meow! <br>";
        }
      }
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

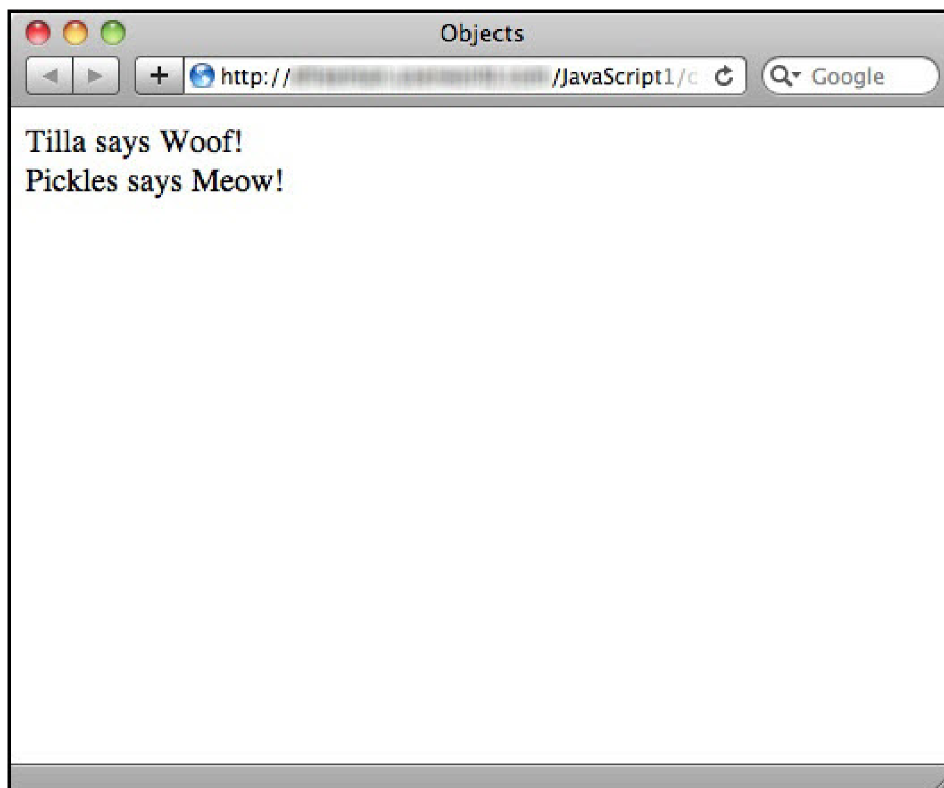
Save it, and open it in a browser. You should see a page that looks like this:

Tilla says Woof!
Pickles says Meow!

OBSERVE:

```
var div = document.getElementById("pets");

var pets = [tilla, pickles];
for (var i = 0; i < pets.length; i++) {
  var pet = pets[i];
  if (pet.type == "dog") {
    div.innerHTML += pet.name + " says Woof! <br>";
  }
  else if (pet.type == "cat") {
    div.innerHTML += pet.name + " says Meow! <br>";
  }
}
```

In this example, we create an array named `pets`, of length 2, containing the two objects. You can iterate over the array just like you normally would, getting each item from the array using its index. In this case, each time through the loop, the variable pet is set to the object at index `i` in the array. Once you have this variable that contains the object, you can access the object's properties and values, like you normally would; for instance, `pet.type` will be "dog" for the first object in the array and "cat" for the second object in the array.



Arrays as Object Property Values

You can also put arrays in objects! Arrays are just like other values that you can put in objects, so let's add a `likes` array to each of our pets. (Don't forget to add the comma after the `weight` property value, because that's no longer the last property in the object, and you'll get an error if you forget it.)

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var tilla = {
        type: "dog",
        name: "Tilla",
        weight: 26,
        likes: ["playing ball", "going for walks", "eating anything"]
      };

      var pickles = {
        type: "cat",
        name: "Pickles",
        weight: 7,
        likes: ["sleeping", "eating butter"]
      };

      var div = document.getElementById("pets");
      var pets = [tilla, pickles];
      for (var i = 0; i < pets.length; i++) {
        var pet = pets[i];
        if (pet.type == "dog") {
          div.innerHTML += pet.name + " says Woof! <br>";
        }
        else if (pet.type == "cat") {
          div.innerHTML += pet.name + " says Meow! <br>";
        }
      }

      div.innerHTML = tilla.name + " enjoys ";
      for (var i = 0; i < tilla.likes.length; i++) {
        div.innerHTML += tilla.likes[i];
        if (i < tilla.likes.length - 1) {
          div.innerHTML += " and ";
        }
      }
    }
  </script>
```

```
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it, and open it in a browser. You should see the page updated with the content, "Tilla enjoys playing ball and going for walks and eating anything". The code you added at the bottom of this program loops through all the items in the `tilla.likes` array. Just like you normally would with an array, you can use the `length` property to get the length of your array and use it in the for loop. Notice that we are accessing each item in the array with `tilla.likes[i]`, using the same syntax you normally would for an array, only using the dot notation to access the array. We also added a check to add an " and " to the string to separate each likes item for display. We check to see if we're not on the last item, and if we're not, we add the " and ".

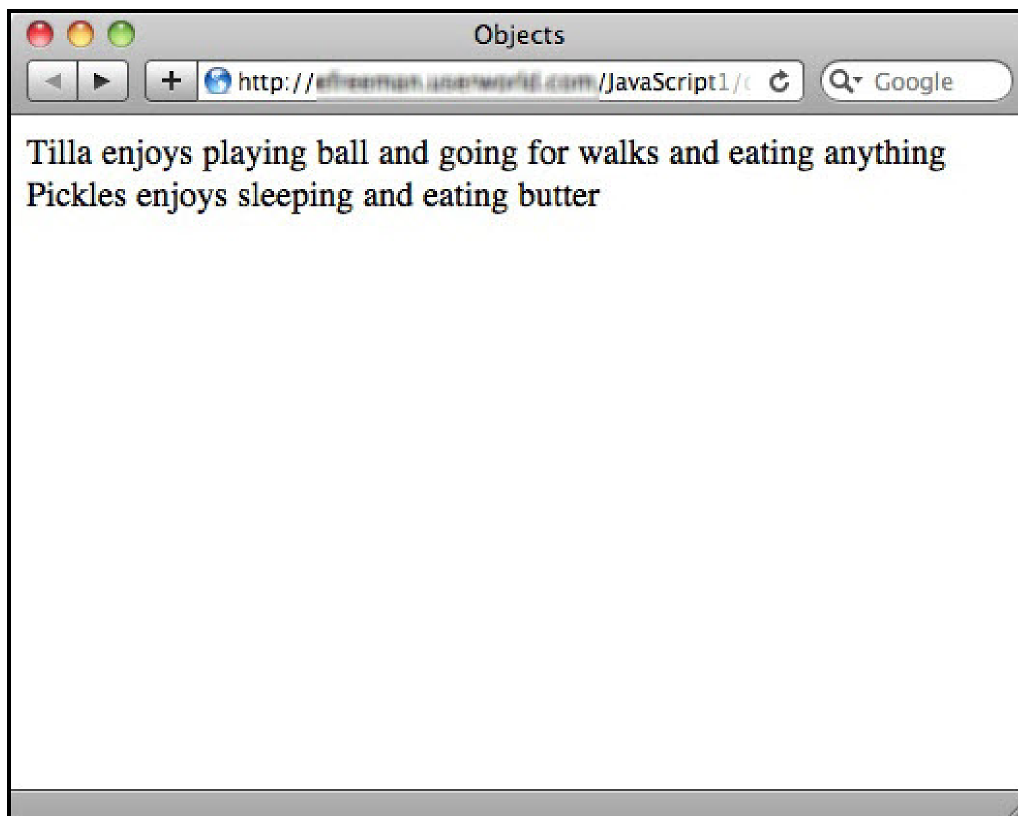
Go ahead and write the code to do the same thing for the `pickles` object and make sure you get the results you expect.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var tilla = {
        type: "dog",
        name: "Tilla",
        weight: 26,
        likes: ["playing ball", "going for walks", "eating anything"]
      };

      var pickles = {
        type: "cat",
        name: "Pickles",
        weight: 7,
        likes: ["sleeping", "eating butter"]
      };

      var div = document.getElementById("pets");

      div.innerHTML = tilla.name + " enjoys ";
      for (var i = 0; i < tilla.likes.length; i++) {
        div.innerHTML += tilla.likes[i];
        if (i < tilla.likes.length - 1) {
          div.innerHTML += " and ";
```

```
            }
        }
        div.innerHTML += "<br>" + pickles.name + " enjoys ";
        for (var i = 0; i < pickles.likes.length; i++) {
            div.innerHTML += pickles.likes[i];
            if (i < pickles.likes.length - 1) {
                div.innerHTML += " and ";
            }
        }
    }
  </script>
</head>
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it, and open it in a browser. Do you see Pickle's likes in your web page? You should see this:



How are Objects and Arrays Similar and Different?

You might be thinking that objects have some similarities with arrays, and you'd be right. They both "collect" things together. However, as you can see, objects collect properties, which are

pairs of names and values, while arrays collect just values. And you access an object property using the property name, while you access array items using an index.

Object Constructors

Now that you know how to create objects, what do you think you'd do if we asked you to create 100 different pets? Or even just 10? You might think that it would get awfully tedious to keep creating the same kind of object over and over and over and over... but don't worry, we won't ask you to create any more pet objects like this, because there is a better way: you can use an *object constructor* instead.

You probably noticed that `tilla` and `pickles` have a lot in common, right? They both have a type, a name, a weight, and an array of likes. The *values* of these properties are different, but they are the same properties.

So, let's create a special kind of function, called a *constructor*, that we can use to *construct objects*. For this example, go ahead and create a new file (saving your previous work separately).

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Object Constructors </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

   function Pet(type, name, weight, likes) {
      this.type = type;
      this.name = name;
      this.weight = weight;
      this.likes = likes;
    }

    function init() {
      var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing
pickles"]);
      var willie = new Pet("dog", "Willie", 45, ["slobbering", "panting",
"eating"]);

      var div = document.getElementById("pets");
      div.innerHTML = annie.name + " is a " + annie.type + " and " +
willie.name +
                      " is a " + willie.type;
    }

  </script>
</head>
```

```
<body>
  <div id="pets">
  </div>
</body>
</html>
```

Save it in your work folder as *constructor.html*, and open it in a browser. The "pets" `<div>` is updated with the content "Annie is a cat and Willie is a dog."

OBSERVE:

```
  <script>
    window.onload = init;

   function Pet(type, name, weight, likes) {
      this.type = type;
      this.name = name;
      this.weight = weight;
      this.likes = likes;
    }

    function init() {
      var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing
pickles"]);
      var willie = new Pet("dog", "Willie", 45, ["slobbering", "panting",
"eating"]);

      var div = document.getElementById("pets");
      div.innerHTML = annie.name + " is a " + annie.type + " and " +
willie.name +
                      " is a " + willie.type;
    }

  </script>
```

Now, instead of creating pets by writing out the object each time (also known as an *object literal*), we call a function Pet. Notice that we used an uppercase "P" to start the name of the function; this is a convention JavaScript programmers use to distinguish *constructor functions* from regular functions.

The Pet constructor function creates a pet object with each of the properties of a pet: type, name, weight, and likes. So to create a pet object, we need to call the Pet constructor function and pass in all the property values we want for that pet. But notice something else: when we called the Pet constructor, we called it in a special way, using the new keyword.

This says "Create a new pet object with these property values." The Pet constructor function uses the values to create a customized Pet object. So, in our example, we use Pet to create two pet objects, annie and willie, each with different property values.

What's the deal with this?

In the Pet constructor, we use this to mean "this object," and to distinguish properties we're setting from regular variables. If we didn't use this, we wouldn't be setting the properties, we'd just be setting variables and those would then not be part of the object created by the constructor (and so, not accessible using the dot notation).

So, when we write `this.type = type`, it's analogous to the `type: "dog"` line in the original code where we wrote out each object literally, only of course when we write `this.type = type` we're setting the value of the *property* type to the value of the *parameter* type.

Are you wondering why the parameter names are the same as the object's property names? You might think that is confusing. They don't actually need to be the same. That's another convention that helps programmers make sure they're passing in all the right values for setting the properties.

Changing Object Properties

You can change the value of a property in an object after that object's created, simply by assigning it a new value. For instance, suppose you want to change Annie's name to her full name, "Annie Boots" later in your program:

DEVELOPER CONSOLE SESSION:

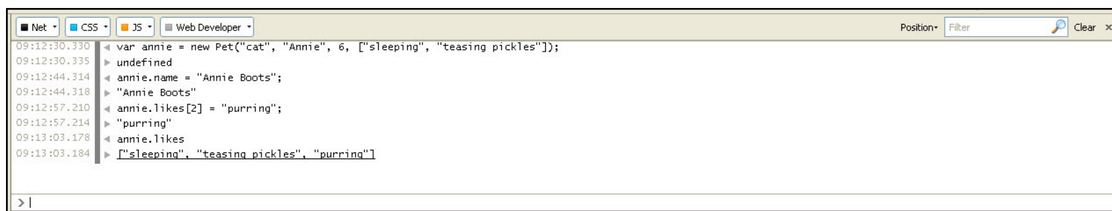
```
var annie = new Pet("cat", "Annie", 6, ["sleeping", "teasing pickles"]);
annie.name = "Annie Boots";
```

You can even change the values in the `likes` array property:

DEVELOPER CONSOLE SESSION:

```
annie.likes[2] = "purring";
```

Now, if you use console.log to display the value of `annie.likes`, you'll see:



Built-In Objects

Now that you know about objects, things like *document* and *window* and element objects and `new Array()` are going to make a whole lot more sense, right?

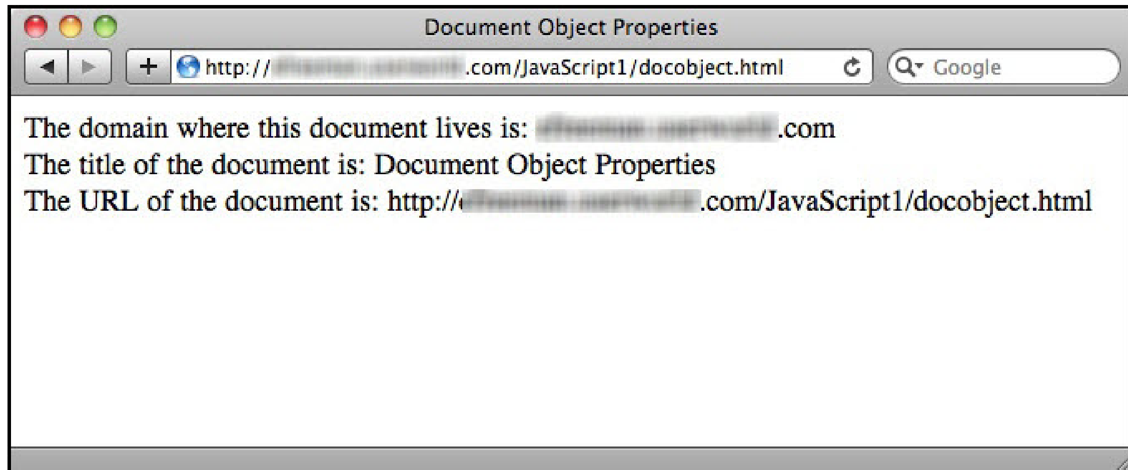
Let's take a look at some of the properties of the *document object*. What do you think the following program will do?

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Document Object Properties </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;
    function init() {
      var div = document.getElementById("result");
      div.innerHTML =
          "The domain where this document lives is: " + document.domain +
"<br>" +
          "The title of the document is: " + document.title + "<br>" +
          "The URL of the document is: " + document.URL;
    }
  </script>
</head>
<body>
  <div id="result">
  </div>
</body>
</html>
```

This program uses the *document object*, which is a built-in object in JavaScript (that is, you don't have to create it—it's just there). We're accessing three properties of this object, `document.domain`, `document.title`, and `document.URL`.

Save it in your work folder as *documentobject.html*, and open it in a browser. Did you get the results you expected? Here's what we get (your property values might be slightly different):



Element Objects

You've been using element objects already, in quite a few examples. Any time you write something like this, you're using an element object:

OBSERVE:

```
var div = document.getElementById("result");
```

In this case, the variable `div` contains an element object that is returned by calling `document.getElementById("result")`. The element object you get back is the object corresponding to the `<div>` element with the id "result." (Look in the HTML for the example above and you'll see the `<div>`). Element objects also have properties, and in fact you've been using one of them already:

OBSERVE:

```
div.innerHTML = "The domain where ...";
```

`innerHTML` is a property that all element objects have. We'll be exploring element objects in more depth later in the course when you learn how to create elements and add them to the DOM!

Using Objects to Collect Global Variables

Before we end this lesson, let's do one more thing that we mentioned in the previous lesson. Look at how you might use an object to reduce the number of global variables in your programs. It's pretty easy, actually. Let's say you have three global variables:

```
var random = 0;
var prevValue = 0;
var currentValue = 0;
```

You can't make any of them local variables, because you need them all in multiple functions, and you can't easily pass the variables between functions as arguments. You can reduce the number of global variables from three to one, simply by collecting these variables inside an object, like this.

OBSERVE:

```
var global = {
  random: 0,
  prevValue: 0,
  currentValue: 0
}
```

Then, of course, when you need to use one of these global variables, you just write `global.random`, or `global.prevValue`, or `global.currentValue`. One way to make sure your global variables never clash with any other JavaScript you might be including (such as libraries, or other code in separate files) is to name your global object something you know will be unique. For instance, you could use your initials and name it `JTS_global` if your name is "Joe Thomas Smith".

In the next lesson, we'll explore how to add behavior to objects. For now, practice creating a few objects of your own.

More about the material in this lesson

The Riverside JS Workshop would like to acknowledge the generosity of O'Reilly Media, Inc. for making this material available to us through the Creative Commons License. We would also like to acknowledge the great work of Elisabeth Robson who authored this content. She is one of our favorites. Elisabeth has written a number of *Head First* programming books for web developers. We recommend them highly. You can browse her work here.

Copyright © 1998-2015 O'Reilly Media, Inc.
This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

The original source document has been altered. It has been edited to accommodate this format and enhance readability.