Truthy, Falsey, and Equality

Lesson Objectives

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

- *test for null and undefined.*
- *test values for truthiness.*
- *compare values using the strict equality operator.*
- *examine and compare the property names and the property values of objects.*
- *design appropriate conditional tests when using equality operators and typecasting.*
- *explore equality of objects.*

You might think that testing for the equality of two values is simple and straightforward. Well actually, sometimes it is, and sometimes it's not. In JavaScript, when we compare two values, we get a result: true or false. However, determining the result of a comparison is not always as straightforward as you might think, because along with true and false, we also have *truthy* and *falsey* values. In addition, testing equality for primitives is different from testing equality for objects. In this lesson, you'll learn what's really happening behind the scenes when you compare values.

For the examples in this lesson, you're welcome to create an HTML file with a `<script>` or use the console. Either is fine. We'll show examples of both.

Truthy, Falsey, and Equality

In JavaScript, we compare values all the time. For instance, you might do this:

OBSERVE:

```
var weather = "sunny";
if (weather == "sunny") {
    console.log("It's sunny today!");
} else {
    console.log("It must be rainy.");
}
```

We compare a string value with another string value using a *conditional expression*, `weather == "sunny"`, to see if they are equal, and the result of that conditional expression is either *true* or *false* (in this case, it's true). In an *if* statement, we use conditional expressions to determine whether to execute a block of code. If the expression in the parentheses results in

true, the first block of code is executed. If it's not, the *else* block is executed (if there is one—if there isn't, execution just continues with the next line of code after the if statement). We can do the same thing with values that are directly true or false like this:

OBSERVE:

```
var isItSunny = true;
if (isItSunny) {
    console.log("It's sunny today!");
} else {
    console.log("It must be rainy.");
}
```

Notice that we don't have to compare `isItSunny` to `true`, because we know that `isItSunny` *is* a boolean value. This means we can shorten the expression to `isItSunny`.

In many cases, we're working with expressions that are true or false, but sometimes we work with values that are *truthy* or *falsey*. What does this mean? It means that some values aren't directly true or false, but are interpreted by JavaScript to *mean* true or false in certain situations, like conditional expressions. Here's an example (before you try these in the console yourself, see if you can guess what you'll see as the result of each statement):

OBSERVE:

```
if (1 == 1) { console.log("1 really does equal 1"); }
if (1) { console.log("1 is true"); }
```

Go ahead and try these statements in the console (remember that you might have to load an HTML page to access the console):

INTERACTIVE SESSION:

```
> if (1 == 1) { console.log("1 really does equal 1"); }
1 really does equal 1
< undefined
> if (1) { console.log("1 is true"); }
1 is true
< undefined
```

The first statement is straightforward. We compare the value `1` with the value `1`, so of course we expect them to be equal, and expect to see the console log message, `1 really does equal 1`.

So what about the second statement? There, we test the value `1` to see if it's true or false, but `1` isn't either true or false, it's 1, right? Yet the result of this statement is that we *do* see the

console log message, `1 is true,` which means that JavaScript must think that 1 is true. Hmmm. That's perplexing. What about this next example; what do you think you'll get?

OBSERVE:

```
if (0) { console.log("0 is true!"); }
```

Try it. This time we *don't* see the console log message, which means JavaScript must think that `0` is false:

INTERACTIVE SESSION:

```
> if (0) { console.log("0 is true!"); }
undefined
```

Try one more experiment:

INTERACTIVE SESSION:

```
> if (-5) { console.log("-5 is true!"); }
-5 is true!
< undefined
```

That's interesting. JavaScript thinks that -5 is true!

So it turns out that numbers other than `0` are *truthy*, and `0` is *falsey*. We use those terms to indicate that even though `-5` isn't true, it results in true in a conditional expression. The same is true with `0`. Even though `0` isn't false, it results in false in a conditional expression.

Experiment a bit on your own. For instance, is `NaN` truthy or falsey? What about `Infinity`?

Values That are Truthy or Falsey

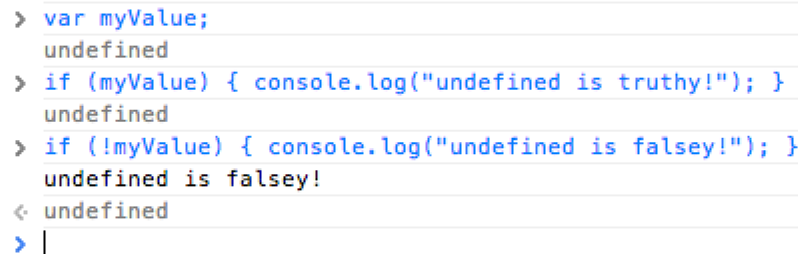
We need to find out which other values are truthy and falsey in JavaScript. Let's do some testing in the console to see how JavaScript treats values in truthy and falsey situations. We'll begin with `undefined`. Before you look at the example below, do you think `undefined` is truthy or falsey?

INTERACTIVE SESSION:

```
> var myValue;
undefined
> if (myValue) { console.log("undefined is truthy!"); }
undefined
```

```
> if (!myValue) { console.log("undefined is falsey!"); }
undefined is falsey!
< undefined
```

Remember that `!` means *NOT*, so if `myValue` is false, then `!myValue` is true. So, *undefined* is falsey, because in the second expression, `myValue` resolves to `false`, and then we say *NOT false* with `!myValue`, which results in `true`, so we execute the if statement block to display the message `"undefined is falsey!"`. Is that what you expected, that is, that *undefined* is falsey? Here's how this session looks in the Chrome console:



What about *null*? Can you guess?

INTERACTIVE SESSION:

```
> myValue = null;
null
> if (myValue) { console.log("null is truthy!"); }
undefined
> if (!myValue) { console.log("null is falsey!"); }
null is falsey!
< undefined
```

So, *null* is also falsey. It kind of makes sense that if *undefined* is falsey, then *null* would also be falsey, right? Okay, we've looked at numbers, undefined and null. What about strings?

INTERACTIVE SESSION:

```
> var myString = "a string";
undefined
> if (myString) { console.log("myString is truthy!"); }
myString is truthy!
< undefined
> if (!myString) { console.log("myString is falsey!"); }
undefined
```

In this case, we see the string `"myString is true"` which means that `myString` is truthy. How about if we set `myString` to the empty string, `""`. Now do you think `myString` will be truthy or falsey?

INTERACTIVE SESSION:

```
> myString = "";
""
> if (myString) { console.log("myString is truthy!"); }
undefined
> if (!myString) { console.log("myString is falsey!"); }
myString is falsey!
< undefined
```

So a string with characters is truthy, but an empty string is falsey. Experiment a bit. What if `myString` is a string with one space in it, like this: " "?

Shortcuts using truthy and falsey results

Knowing that *undefined*, *null*, and `""` are all falsey values, can you think of a good way to shorten the code below?

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Truthy, Falsey, Equality </title>
  <meta charset="utf-8">
  <script>
        var myString = prompt("Enter a string");;
        if (myString == null || myString == undefined ||
myString == "") {
            console.log("Please enter a non-empty string!");
        } else {
            console.log("Thanks for entering the string '" +
myString + "'");
        }
  </script>
</head>
<body>
</body>
</html>
```

Save this file your work folder as *stringTest.html*, and load it into a browser. Open the console (you might need to reload the page to see the output in the console). Enter a string and note

the message you see. Try entering different values at the prompt, like null (just click *OK*), `""`, and `"test"`, for instance. Now that you know about truthy and falsey values, you can shorten this code:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Truthy, Falsey, Equality </title>
  <meta charset="utf-8">
  <script>
        var myString = prompt("Enter a string");;
        if (myString == null || myString == undefined ||
myString == "" !myString) {
            console.log("Please enter a non-empty string!");
        } else {
            console.log("Thanks for entering the string '" +
myString + "'");
        }
  </script>
</head>
<body>
</body>
</html>
```

Save the changes to you file, and reload it in the browser. You get the same behavior as before, but with a much shorter conditional expression. Again, try entering a few different values to make sure this works as you expect. Try replacing the prompt for `myString` to undefined, null, the empty string, or something else.

In cases where you need to test to make sure that a variable has a truthy value, but you don't care what that value is exactly, you can use this type of shortcut to save yourself some typing. Be careful though. In cases where you need to test for a specific value, you also need to understand *implied typecasting*. We'll talk about next.

Implied Typecasting

Does 88 equal "88?" That's an interesting question. Let's see:

INTERACTIVE SESSION:

```
> var myNum = 88;
undefined
> var myString = "88";
```

```
undefined
> if (myNum == myString) {
      console.log("My number is equal to my string!");
  }
My number is equal to my string!
< undefined
```

JavaScript converts the string `"88"` into a number *before* doing the comparison between `myNum` and `myString`, so the comparison actually happens between 88 and 88. Of course those values are equal, so the result is true, and we see the console log message, `"My number is equal to my string!"`

This process of converting a string to number before doing a comparison or another operation is called *implied typecasting* (also known as *type coercion* or *type conversion*). JavaScript does implied typecasting as needed. For instance, whenever you do something like this:

INTERACTIVE SESSION:

```
> var age = 29;
undefined
> var output = "My age is " + age;
undefined
> output
"My age is 29"
```

Here you are using JavaScript's ability to convert the variable `age` from a number to a string automatically, so it can be concatenated with the string `"My age is 29"`.

When converting strings and numbers, be careful because the result may not always be what you expect. You know that sometimes JavaScript converts a string to a number, like when we compare `88` and `"88"`, and you know that sometimes JavaScript converts a number to a string, like when you want to concatenate a number to a string. So what do you think the result will be when we try to add a string that *contains* a number to a number?

INTERACTIVE SESSION:

```
> var x = 4;
undefined
> x = x + "4";
"44"
> x
"44"
```

You might've expected to get 8. Let's try another experiment:

INTERACTIVE SESSION:

```
> var myString = "test";
undefined
> if (myString) {
      console.log("'test' is truthy");
  } else {
      console.log("'test' is falsey");
  }
'test' is truthy
< undefined
> if (myString == true) {
      console.log("'test' is true");
  } else {
      console.log("'test' is false");
  }
'test' is false
< undefined
```

Notice that in the first if statement, `if (myString) ...`, we rely on the truthy-ness or falsey-ness of `myString` to result in true or false to determine the flow of execution. However, in the second *if* statement, `if (myString == true) ...`, we compare the value of `myString` with the boolean `true`, explicitly. JavaScript doesn't do implied typecasting and conversion of `myString` to true or false here.

As you can probably tell, it's a bit tricky to know for sure in every case exactly how JavaScript will (or won't) typecast and convert a value for comparison. One way that you can be more confident that you'll get the result you expect is to use the *strict equality operator,* `===`, in place of the *equality operator*, `==`.

For more information about how JavaScript performs type conversions, see the ECMAScript specification.

Testing Equality

JavaScript has two operators for testing equality, `==` and `===`. You've probably been using `==` in your JavaScript programming, but consider using `===` instead (at least sometimes). Let's take a closer look at the difference between these two operators, and why you might use one over the other. We'll begin by looking at an example of these two operators in action:

INTERACTIVE SESSION:

```
> null == undefined
true
> null === undefined
false
```

The equality operator, ==, attempts to do implied typecasting *before* it compares two values. So, in the first expression above, JavaScript sees that you're trying to compare values of two different types, and so, tries to convert one type to the other in order to do the comparison. In this case, JavaScript could either convert `undefined` to *null*, or `null` to *undefined* (JavaScript can do it either way), and then the two values are equal.

However, the strict equality operator, ===, does *not* do implied typecasting. Instead, it compares the two values as they are. If the types of the two operands are different, then the result is false immediately. Let's see what happens when we use *strict equality* on our previous comparison of 88 with "88":

INTERACTIVE SESSION:

```
> var myNum = 88;
undefined
> var myString = "88";
undefined
> if (myNum === myString) {
      console.log("My number is equal to my string!");
  } else {
      console.log("A number shouldn't really be equal to a
string!");
  }
A number shouldn't really be equal to a string!
< undefined
```

Let's try *strict equality* on a falsey value, like 0:

INTERACTIVE SESSION:

```
> var zero = 0;
undefined
> if (zero == false) {
      console.log("zero is a falsey value!");
  }
zero is a falsey value!
< undefined
> if (zero === false) {
```

```
      console.log("zero is a falsey value!");
  } else {
      console.log("Now we don't convert zero to false");
  }
Now we don't convert zero to false
< undefined
```

So, strict equality prevents conversion of a falsey value, like 0, to false.

Just like the `==` equality operator has a negative version, `!=` (meaning *not equal to*), the strict equality operator also has a negative version, `!==`. The only difference between `!=` and `!==` is that `!=` attempts to typecast its operands to the same type, while `!==` does not.

Do some experimenting with the four operators: `==`, `!=`, `===` and `!==`. You might be surprised by what you find.

Some programmers *always* use strict equality (and strict inequality) rather than equality (and inequality). However, sometimes you might want to take advantage of JavaScript's ability to do typecasting. If you do, be cautious and make sure you know exactly how that typecasting is going to work on the types of values you expect.

You can get all the gory details of the algorithms used by the equality operator and the strict equality operator (also known as the *identity operator*) in the ECMAScript specification.

Objects and Truthy-ness

So far, we've been looking at the truthy-ness and falsey-ness of primitive values like numbers, strings, null and undefined. What about objects?

INTERACTIVE SESSION:

```
> var o = { name: "object" };
undefined
> o
Object {name: "object"}
> if (o) {
     console.log("This object is truthy!");
  }
This object is truthy!
< undefined
```

So it looks like objects are truthy. What about empty objects? (Remember that empty strings are falsey, so you might expect that empty objects are also falsey.)

INTERACTIVE SESSION:

```
> var p = {};
undefined
> if (p) {
      console.log("This object is truthy!");
  } else {
      console.log("This object is falsey!");
  }
This object is truthy!
< undefined
```

Interesting. Even a completely empty object, like `p`, is still truthy. But...

INTERACTIVE SESSION:

```
> p == true
false
> p == false
false
```

Even though `p` might be truthy, it's not equal to true (or false) using the equality operator, so no type conversion is happening here.

Objects and Equality

Let's do a few more tests in the console and compare our empty object, `p`, to some other values:

INTERACTIVE SESSION:

```
> var p = {};
undefined
> p == 0
false
> p == null
false
> p == undefined
false
> p == "{}"
false
> p == {}
false
```

In this example, we use the equality operator, `==`, because we want JavaScript to try to typecast the values for comparison. As you can see, all of the results are false, which means that even if JavaScript is able to typecast, the comparison is still false.

Most of these results are probably expected, but that last one sure isn't! We know that `p` is an empty object, `{ }`, so why isn't `p` equal to another empty object? Aren't they the same thing?

Go ahead and create a file with two objects so we can experiment:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Comparing objects </title>
  <meta charset="utf-8">
  <script>
        var book1 = {
            title: "Harry Potter",
            author: "JK Rowling",
            published: 1999,
            hasMovie: true
        };
        var book2 = {
            title: "Harry Potter",
            author: "JK Rowling",
            published: 1999,
            hasMovie: true
        };

        if (book1 == book2) {
            console.log("The two books are the same");
        } else {
            console.log("The two books are different");
        }
  </script>
</head>
<body>
</body>
</html>
```

Here, we create two book objects using object literals, and then test to see if the books are equal. Save this file in your work folder as *objectsTest.html*, and load it in your browser. In the console, you see the message, "The two books are different."

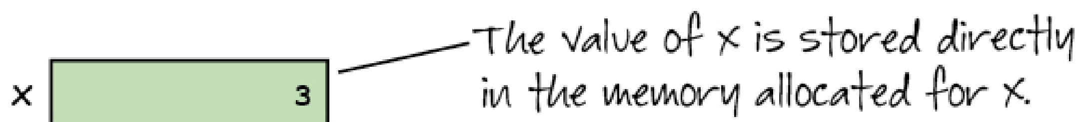
The two books, `book1` and `book2`, are exactly the same, that is they have the same properties. All the property names are the same, the property values are the same, and the number of properties is the same. So why aren't they equal? (Note that we're using the equality operator here to test equality. We know the types of the two objects are the same, so we don't have to worry about any typecasting happening).

The two objects are not equal because of an important difference in how primitive values are stored and how objects are stored in the computer's memory. When you create a primitive value, let's say:

OBSERVE:

```
var x = 3;
```

The computer allocates a bit of memory, gives it the name "x", and saves the value 3 in that bit of memory:



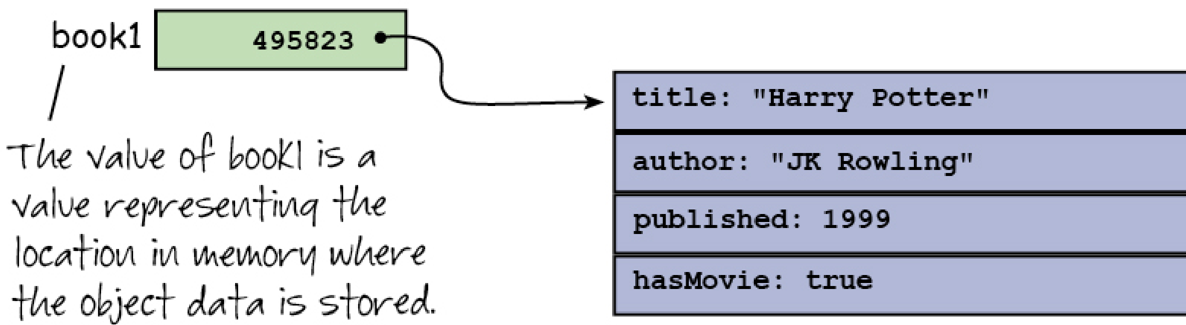
Now, compare that to what happens when you create an object, let's say `book1` from the example above:

OBSERVE:

```
var book1 = {
    title: "Harry Potter",
    author: "JK Rowling",
    published: 1999,
    hasMovie: true
};
```

In this case, the computer names and allocates some memory for each of the properties in the object, and then allocates a separate bit of memory for the variable name, and in that memory stores a value that points to the place in memory where the object is actually stored. This is called an *object reference*.

So the variable `book1` doesn't contain the object itself. It actually contains a *reference* to the object. Like this:



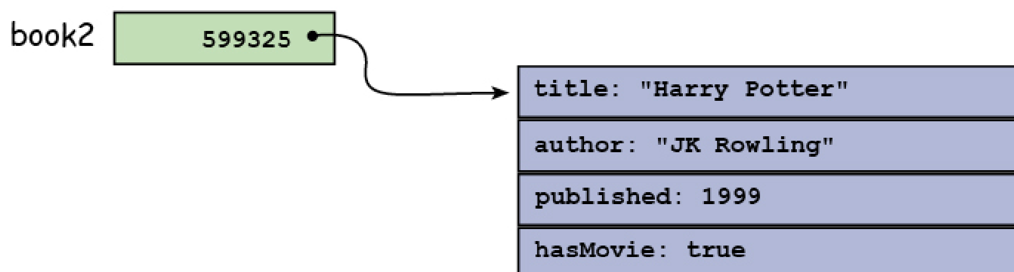
In this case, the object value (that is, all the properties in the object, plus a few other things about the object) is stored at memory location 495823 (I just made that up for this example, but you get the idea), and the variable `book1` contains that location (in the same way that `x` contains 3).

Now let's see what happens when we create the second book object, `book2`:

OBSERVE:

```
var book2 = {
    title: "Harry Potter",
    author: "JK Rowling",
    published: 1999,
    hasMovie: true
};
```

Even though the properties are exactly the same, a completely separate book object is created and stored in a completely different part of memory:



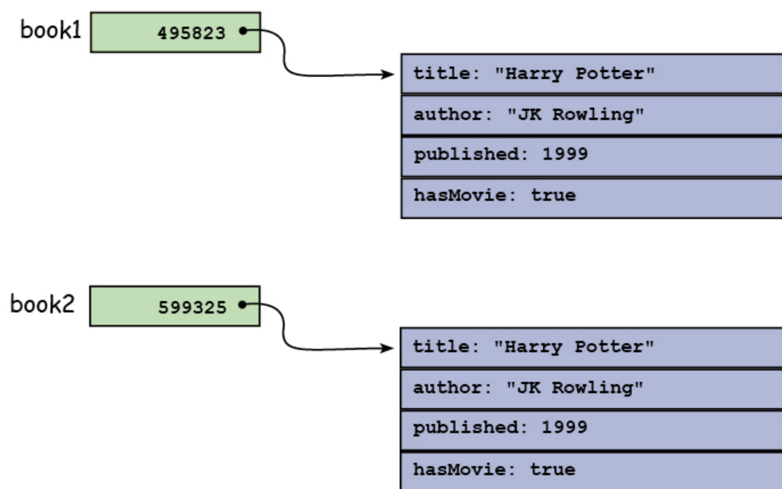
The variable `book2` contains the memory location of this second object, and the memory location is *different* from the memory location for `book1`.

So when we compare `book1` and `book2`:

OBSERVE:

```
if (book1 == book2) {
    console.log("The two books are the same");
} else {
    console.log("The two books are different");
}
```

the values that are compared are the memory locations of the two objects. They are *not* equal, so we see the message "The two books are different."

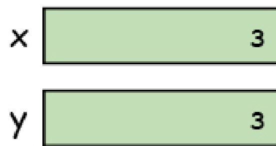


When you compare book1 and book2, again JavaScript looks at the value in memory for book1 and compares it to the value in memory for book2. In this case, they are different, because the data for each object is stored in a different place in memory!

Compare this to what happens when we compare primitive values:

INTERACTIVE SESSION:

```
> var x = 3;
undefined
> var y = 3;
undefined
> if (x == y) { console.log("x and y are the same"); }
x and y are the same
< undefined
```



When you compare x and y, JavaScript looks at the value in the memory for x and compares it to the value in the memory for y and if they are the same value, then x and y are the same.

In the example above where we compared two book objects, we used *literal* objects for the books. Do you think the result would be the same when if we used an object constructor? Let's see:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Comparing objects, take two </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
        this.title = title;
        this.author = author;
        this.published = published;
        this.hasMovie = hasMovie;
    }
    var book1 = new Book("Harry Potter", "JK Rowling", 1999, true);
    var book2 = new Book("Harry Potter", "JK Rowling", 1999, true);
    if (book1 == book2) {
        console.log("book1 is equal to book2");
    } else {
        console.log("book1 is NOT equal to book2");
    }
  </script>
</head>
<body>
</body>
</html>
```

Now we use a constructor function, `Book()`, to create two books, and then test to see if they are equal. Save this file in your work folder as *objectsTest2.html*, and open it in your browser. In the console, you see the message, `book1 is NOT equal to book2.`

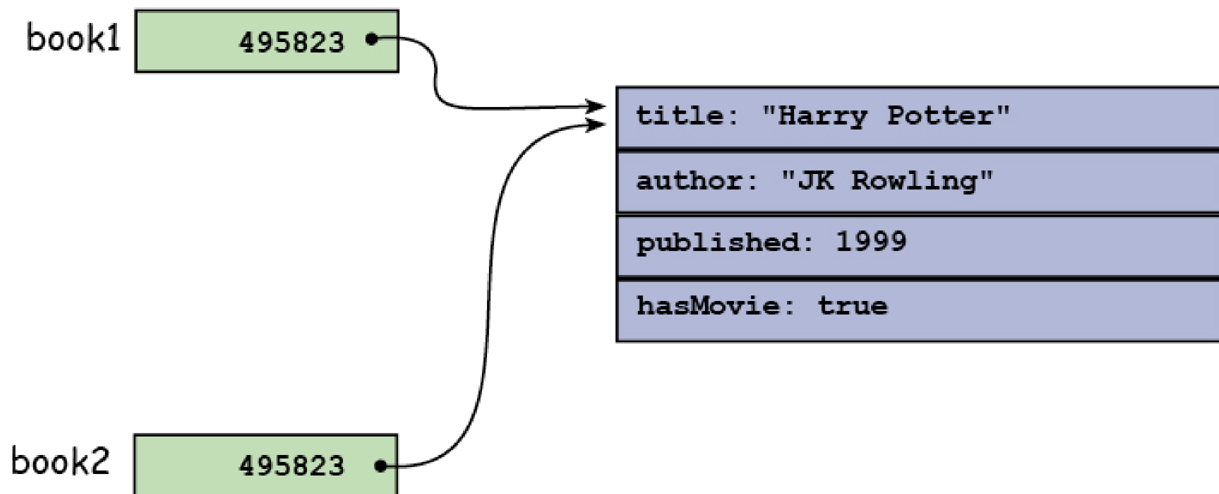
The result is the same because we are creating two completely different book objects, even though they use the same constructor and have the same properties.

Okay, so knowing what you know about how objects are stored in memory, what do you think happens when we change the program like this:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Comparing objects, take two </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
        this.title = title;
        this.author = author;
        this.published = published;
        this.hasMovie = hasMovie;
    }
    var book1 = new Book("Harry Potter", "JK Rowling", 1999,
true);
    var book2 = new Book("Harry Potter", "JK Rowling", 1999,
true);
    var book2 = book1;
    if (book1 == book2) {
        console.log("book1 is equal to book2");
    } else {
        console.log("book1 is NOT equal to book2");
    }
  </script>
</head>
<body>
</body>
</html>
```

Save these changes to *objectsTest2.html*, and open or refresh the page in your browser. In the console, you see the message `book1 is equal to book2`. Why? Because when we assign the value of book1 to book2 (`var book2 = book1`), we store the *memory location* of the data in `book1` into `book2`, like this:



So now, when we compare the values of `book1` and `book2`, they are the same: they are both values that point to the same memory location. Let's update the program one more time:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Comparing objects, take two </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
        this.title = title;
        this.author = author;
        this.published = published;
        this.hasMovie = hasMovie;
    }
    var book1 = new Book("Harry Potter", "JK Rowling", 1999,
true);
    var book2 = book1;
    if (book1 == book2) {
        console.log("book1 is equal to book2");
    } else {
        console.log("book1 is NOT equal to book2");
    }
    book1.star = "Harry";
    console.log(book1);
    console.log(book2);
  </script>
</head>
```

```
<body>
</body>
</html>
```

Save your changes, and open or refresh the file in your browser. Take a look at the two book objects that we display in the console. Notice anything interesting?

OBSERVE:

```
Book {title: "Harry Potter", author: "JK Rowling", published:
1999, movie: true, star: "Harry"}
Book {title: "Harry Potter", author: "JK Rowling", published:
1999, movie: true, star: "Harry"}
```

In the code we added a new property, `star` to `book1`, and set its value to "Harry." Yet when you look at the two objects in the console, you can see that *both* `book1` and `book2` now have the property star, with the value "Harry". How did this happen!?

Well, remember that `book2` points to the same location in memory that `book1` does. So if we change the data in `book1`, we're also changing the data in `book2` because they are the *same* object.

What do you think would happen if we changed the title of `book2`? Try changing the title of `book2` to "Harry Potter and the Sorcerer's Stone." What do you see when you display `book1` and `book2`?

Now, you might be asking, "If I can't compare two different objects using the equality operator to see if they are the same (that is, that they have the same properties and values), how do I know if two objects are the same?"

The answer is that you have to look at each property of an object separately. This isn't too hard to do if the properties in an object are all primitive values (numbers, strings, booleans). However, if your objects have nested objects and/or methods, then it gets a bit trickier because then the solution depends on what you mean by "equality" in the case of two objects. What do you think it means for one object to be "equal" to an other? That’s a good topic for you to think about.

Various JavaScript libraries have tackled this question by implementing functions that check equality of objects. Be cautious though because different libraries may have different ideas about what equality of objects means. Make sure the library function works as you expect. For example, you can use the Underscore.js library's `isEqual()` function to test the equality of objects.

We covered a lot of ground in this lesson, including truthy and falsey values, implied typecasting and what can happen when you compare two values, two different kinds of equality operators, and the difference in comparing primitive values and object values. Whew! That's lots of detail, some of which you may not have encountered before, but that you'll need to know as you get into more advanced JavaScript programming.

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.