Checkpoint 11

References

Outline

In this module so far, you've had a chance to revisit some familiar concepts and expand your skills. But in this checkpoint, you will start exploring something new. It's time for you to learn about an extremely important concept in programming: *references*.

At this point, you will learn the basics of references, and you'll discover how primitive data types are different from reference data types. *This is probably one of the most important checkpoints you have encountered so far.* It's essential that you understand this material deeply, so dedicate ample time to reviewing and working through the ideas in this checkpoint. Take your time with it. Don't gloss over it.

By the end of this checkpoint, you will be able to do the following:

- Evaluate code that uses primitive and reference data types

Primitive data types

JavaScript has several data types that are passed by *value*. These are often called *primitive data types*, or simply primitives, and you've already learned a bit about them! Primitive data types in JavaScript include strings, numbers, and booleans, as well as `null` and `undefined`. When a primitive data type is assigned to a variable, that variable gets its own copy.

Take a look at the following line of code:

```
const title = "Mort";
```

In this example, the variable `title` *contains* the string `"Mort"`. If you reassign the value contained by `title` to another variable, such as `name`, both variables will contain their own copy of that value. You can see this below.

```
let title = "Mort";
const name = title;
title = "Equal Rites";
console.log(title, name); //> "Equal Rites", "Mort"
```

The above code is very important. Although it may seem obvious, you should take a moment to understand what's happening here. When `title` is reassigned to `"Equal Rites"`, notice that the `name` variable still holds its own copy of `"Mort"`.

Now, take a look at another example. As you read over this, what do you expect to be logged?

```
const price = 999;
let salePrice = price;
salePrice -= 100;
console.log(price, salePrice); //> ???
```

In the code above, the `price` variable will still be the number `999`, while the `salePrice` variable will be `899`. Despite the modification of `salePrice`, `price` retains its own copy of the value `999`.

Reference data types

But there's another kind of data type: reference data types. *Reference data types*, sometimes just called references, in JavaScript include functions, objects, and arrays. When one of these data types is assigned to a variable, the variable will contain a *reference* (which is also called a *pointer*) to the data.

Outline

Take a look at the following example. What do you expect to happen here?

```
const book = { title: "Mort", author: "Terry Pratchet
const mort = book;
book.price = 789;
console.log(mort); //> { title: "Mort", author: "Ter
```

Are you surprised by the result? Both the `book` variable and the `mort` variable point toward the same *refere* the object through one variable, the other variable will have that same modification.

This is further illustrated by the following two examples. Here's the first:

```
const author = {};
const book = {};
console.log(author === book); //> false
```

In the above `console.log()` statement, two empty objects, `author` and `book`, are compared with one another. When the `===` sign is used, it compares these objects' references.

These objects may look the same. However, the statement resolves to `false`. That is because `author` and `book` each store their own reference to *different* objects.

Now, check out this example:

Outline

```
const sourcery = { title: "Sourcery", author: "Terry
const favoriteBook = sourcery;
console.log(sourcery === favoriteBook); //> true
```

In the above `console.log()` statement, `sourcery` and `favoriteBook` contain references to the *same* object. When the `===` sign is used, it compares these references and resolves to `true`.

This works the same way for arrays, as well. Take a look:

```
const books = ["Mort", "Sou
const series = books;
series.push("Guards! Guards!");
console.log(books); //> [ "Mort", "Sourcery", "
```

In the above example, `push()` changes the array to include a new title. This *mutates* the original array, which both `books` and `series` are pointing toward.

Note: In the examples above, `const` is used instead of `let` to store an array or object. And yet, the values *inside* of these arrays and objects can be changed. Ultimately, `const` only stops reassignment, without changing the values inside of the reference itself (in other words, the array or object).

The assignment below includes a quiz to test your knowledge. Here is the answer key for you to check your answers.

Outline

Checkpoint

This checkpoint will be autograded. Please click the link below to open your assignment in a new tab. Once you complete the assignment, you will see a button allowing you to submit your answers and move on to the next checkpoint.

Your work

03.29.21 **Approved**

Completed Next checkpoint

How would you rate this content?

Report a typo or other issue

Go to Overview

Outline