Object and Reference Types Overview

In this lesson we're going to talk a little bit about data storage and access. One of the wonderful things about JavaScript is just how versatile its data storage methods are. In order to understand how they work, I'm going to give you a quick refresher about how reference types work and how they're different from primitive types.

First of all, the thing to remember is that all data types inside of JavaScript inherit from the object type. Arrays and functions are sub-types of objects that inherit reference behavior. And a reference type can expand to any size, and their properties can be modified, unlike primitive types that have a fixed size in memory. Primitive types, such as number and boolean, have a fixed size in memory. This makes it convenient for JavaScript to pass them by value, meaning that each copy of a primitive variable is independent and can change its value without altering other copies.

Reference types, on the other hand, can expand to be any size that they want to be. In order to make that efficient in memory, JavaScript stores them as references to one unique location in memory. This means that changing one can change the value of another because all of them refer to the same shared location in memory. What this means in practice is that certain JavaScript types can be copied from one place to another and other data types are referenced from the same location.

Reference and Primitive Data Types

I'll show you how that looks. We'll start by making some primitives.

```
// PRIMITIVE
var number1 = 123;
var number2 = number1;
console.log(number1); // 123
console.log(number2); // 123
number2 = 456;
console.log(number1); // 123
console.log(number2); // 456
var toy1 = {"color":"red", "size":5, "soft":true};
var toy2 = toy1;
console.log(toy1.color); // "red"
console.log(toy2.color); // "red"
toy2.color = "blue";
console.log(toy2.color); // "blue"
console.log(toy1.color); // "blue"
```

Each one of these values are independent and doesn't refer to a shared location. However, if we have reference variable types, we would see a very different behavior:

```
// REFERENCE
var toy1 = {"color":"red", "size":5, "soft":true};
var toy2 = toy1;
console.log(toy1.color); // "red"
console.log(toy2.color); // "red"
toy2.color = "blue";
console.log(toy2.color); // "blue"
console.log(toy1.color); // "blue"
```

That's the behavior of reference types. Understanding this behavior is fundamental to being able to use JavaScript effectively to store and manage data.

Testing For Equality In Reference Types

In addition to understanding primitive and reference types, it's also critical to understand the different equality tests that are available in JavaScript and avoid coercion if you don't want coercion.

Primitive types are compared by value, and they can be coerced into the same type during comparison, by using a double equals sign for comparisons. Reference types, however, are compared by their unique location in memory.

We'll start using the same primitive type.

```
//SAME PRIMITIVE TYPE
var fruits = "Apples, Pears, and Tomatoes";
var foods = "Apples, Pears, and Tomatoes";
console.log(fruits == foods); //true
console.log(fruits === foods); //true

//PRIMITIVE TYPE COERCION
var quantities = 123;
var quantityString = "123";
console.log(quantities == quantityString); //true
console.log(quantities === quantityString); //false
```

When you use coercion to compare two primitives by value, they're coerced into strings. If we use a triple equal sign, which forces comparison without coercion, we'll get false, and that's because the actual value of `quantities` is not being coerced into a string.

Let's check if we have the same reference type:

```
//SAME REFERENCE TYPE
var fruitsObject = {Apples:3, Pears:2, Tomatoes:5};
var foodsObject = {Apples:3, Pears:2, Tomatoes:5};
console.log(fruitsObject == foodsObject); //false
console.log(fruitsObject === foodsObject); //false
```

We get `false` because these are separate locations in memory, they are not equal to each other. Even using the coercion equality test, if we clear and run, we still get false. Two reference types that are separate in memory don't equal each other even if their contents are exactly the same.