Reference vs. Primitive Types

At this point you've worked with many different data types - booleans, numbers, strings, arrays, objects, etc. It's now time to to go a little more in depth into the differences between these data types.

When you finish this reading, you should be able to:

- Identify whether a data type is a Primitive type or a Reference type.

Primitives vs. Objects

JavaScript has many data types, six of which you've encountered so far:

Five Primitive Types:

1. `Boolean`- `true`and `false`
2. `Null`- represents the intentional absence of value.
3. `Undefined`- default return value for many things in JavaScript.
4. `Number`- like the numbers we usually use (`15`, `4`, `42`)
5. `String`- ordered collection of characters ('apple')

One Reference Type:

1. `Object`- (an array is also a kind of object)!

You might be wondering about why we separated these data types into two categories - Reference & Primitive. Let's talk about the one of the main ways*Reference Types*and *Primitive Types*differ:

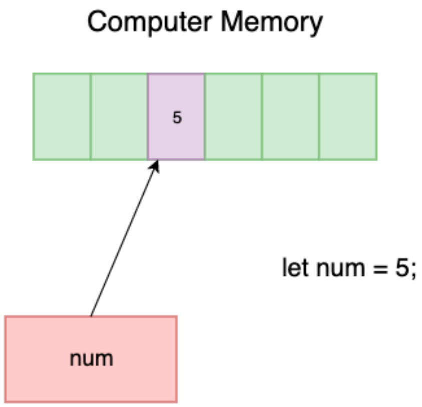
- *Primitive types*are **immutable**. Meaning they cannot change.

Immutability

When we talk about primitive types the first thing we mentioned was*mutability*. Primitives are **immutable**meaning they can not be directly changed. Let's look at an example:

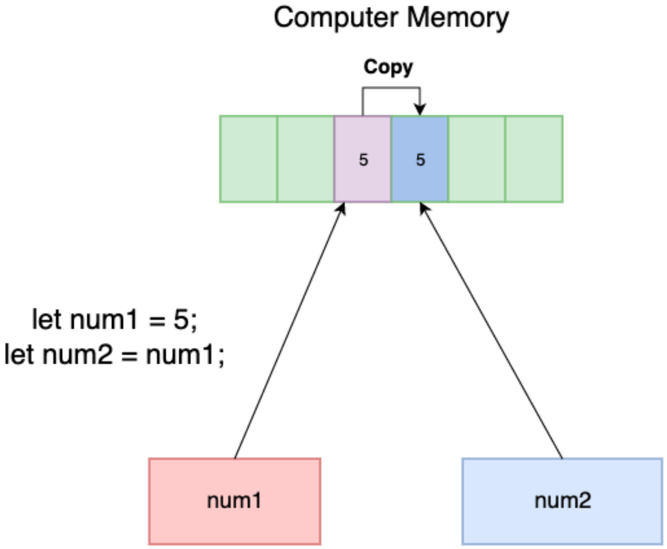
```
let num1 = 5;
// here we assign num2 to point at the value of the number variable
let num2 = num1;

// here we *reassign* the num1 variable
num1 = num1 + 3;

console.log(num1); // 8
console.log(num2); // 5
```

Whoa wait whaaaat? Let's break this down was just happened with some visuals. We start by assigning `num1`. JavaScript already knows that the number `5` is a primitive number value. So when we are assigning `num1` to the value of 5 we are actually telling the `num1`variable to point to the place the number 5 takes up in our computer's memory:



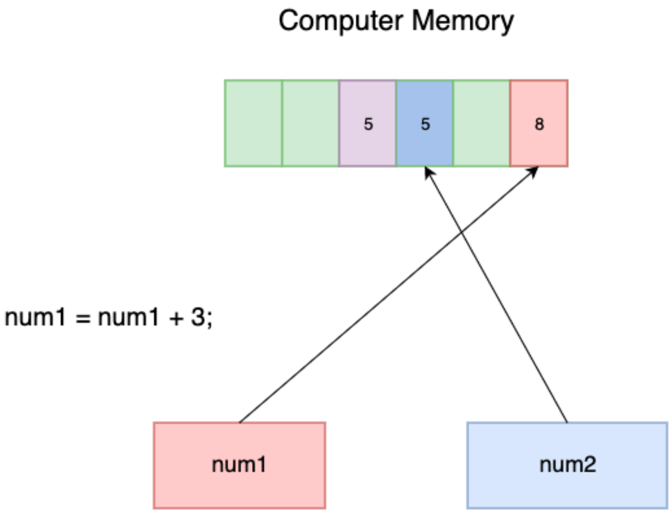
```
let num = 5;
```

Next we assign `num2` to the **value** of `num1`. What effectively happens when we do this is we are *copying*the value of `num1`and then pointing `num2` at that copy:



```
let num1 = 5;
let num2 = num1;
```

Now here is where it gets really *interesting*. We cannot change the 5 our computer has in memory - because it is a **primitive**data type. Meaning if we want `num1` to equal 8 we need to **reassign**the value of the `num1`variable. When we are talking about *primitives*reassignment breaks down into simply having your variable point somewhere else in memory:



```
num1 = num1 + 3;
```

All this comes together in `num1`now pointing at a new value in our computer's memory. Where does this leave `num2`? Well because we never reassigned `num2` it is still pointing at the value it originally copied from `num1`and pointing to 5 in memory.

So that in essence is **immutability**, you can not change the values in memory only reassign where your variables are pointing.

Let's do another quick example using booleans:

```
let first = true;
let second = first;

first = false;

// first and second point to different places in memory
console.log(first); // false
console.log(second); // true
```

Mutability

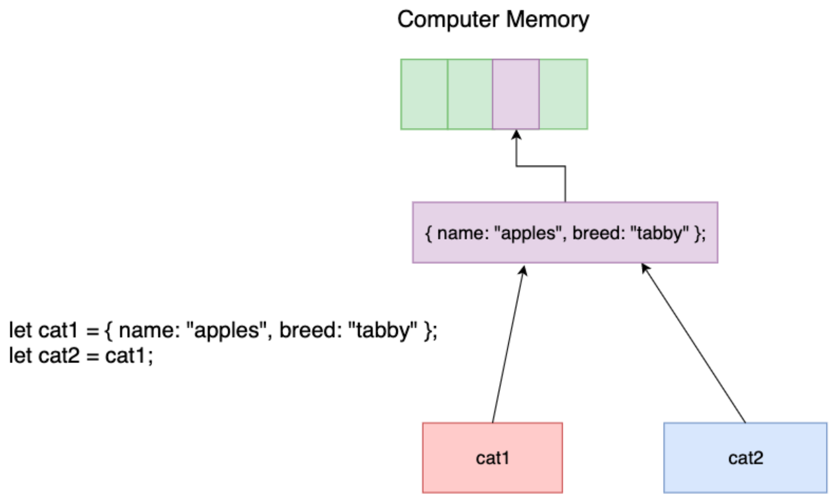
Let's now talk about the inverse of immutability: mutability.

Let's take a look at what we call **reference**values which **are**mutable. When you assign a reference value from one variable to a second variable, the value stored in the first variable is also copied into the location of the second variable.

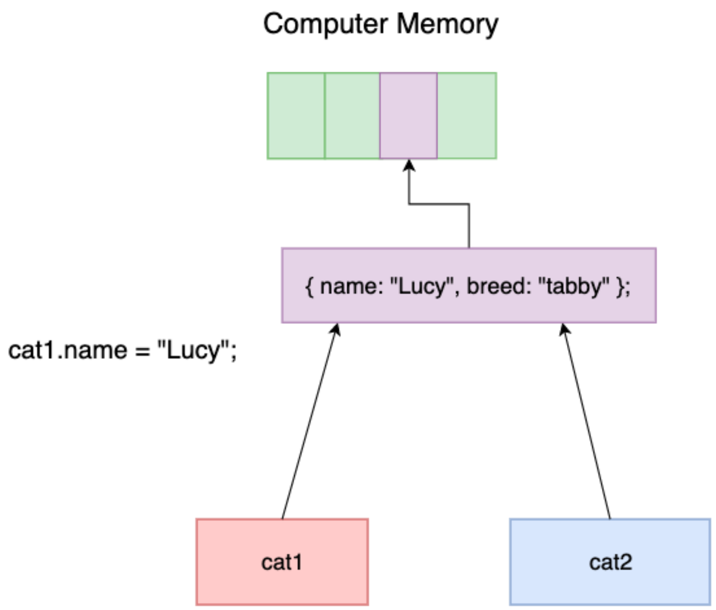
Let's look at an example using objects:

```
let cat1 = { name: "apples", breed: "tabby" };
let cat2 = cat1;

cat1.name = "Lucy";

console.log(cat1); // => {name: "Lucy", breed: "tabby"}
console.log(cat2); // => {name: "Lucy", breed: "tabby"}
```

Here is a visualization of what happened above. First we create `cat1`then assign `cat2` to the value of `cat1`. This means that both `cat1`and `cat2`are pointing to the **same object**in our computer's memory:


Computer Memory
{ name: "apples", breed: "tabby" };
let cat1 = { name: "apples", breed: "tabby" };
let cat2 = cat1;
cat1
cat2

Now looking at the code above we can see what when we change **either**`cat1`or `cat2`, since they are both pointing to the same place in memory, **both**will change:



This holds true of arrays as well. Arrays are a kind of object - though obviously different. We'll go a lot deeper into this when we start talking about classes - but for now concentrate on the fact that arrays are also a *Reference Type*.

See below for an example:

```
let array1 = [14, "potato"];
let array2 = array1;

array1[0] = "banana";

console.log(array1); // => ["banana", "potato"]
console.log(array2); // => ["banana", "potato"]
```

If we change `array1`we also change `array2`because both are pointing to the same *reference*in the computer's memory.

What you learned

- How to work with variables that are both Primitive types and Reference types.