Mutability

Before you move on to reading about the idea of References, please take a minute to read this first.

We saw that object values can be modified. The types of values discussed in earlier chapters, such as numbers, strings, and Booleans, are all *immutable*—it is impossible to change values of those types. You can combine them and derive new values from them, but when you take a specific string value, that value will always remain the same. The text inside it cannot be changed. If you have a string that contains "cat", it is not possible for other code to change a character in your string to make it spell "rat".

Objects work differently. You *can* change their properties, causing a single object value to have different content at different times.

When we have two numbers, 120 and 120, we can consider them precisely the same number, whether or not they refer to the same physical bits. With objects, there is a difference between having two references to the same object and having two different objects that contain the same properties. Consider the following code:

```
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};

console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false

object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

The object1 and object2 bindings grasp the *same* object, which is why changing object1 also changes the value of object2. They are said to have the same *identity*. The binding object3 points to a different object, which initially contains the same properties as object1 but lives a separate life.

Bindings can also be changeable or constant, but this is separate from the way their values behave. Even though number values don't change, you can use a let binding to keep track of a changing number by changing the value the binding points at. Similarly, though a const binding to an object can itself not be changed and will continue to point at the same object, the *contents* of that object might change.

```
const score = {visitors: 0, home: 0};
// This is okay
score.visitors = 1;
// This isn't allowed
score = {visitors: 1, home: 1};
```

When you compare objects with JavaScript's == operator, it compares by identity: it will produce true only if both objects are precisely the same value. Comparing different objects will return false, even if they have identical properties. There is no "deep" comparison operation built into JavaScript, which compares objects by contents, but it is possible to write it yourself.