Null and Undefined Types

Two of the critical values to know about for variables are `null` and `undefined`. We've encountered both of these already, but let's get into a little more detail about what they are and how they work.

Undefined Variables

First of all, `undefined` variables are variables that have been declared but have not yet been set to a value. When a variable has not yet been defined, `undefined` is actually the type of that variable, and the boolean value of an `undefined` variable is `false`.

```
var newVariable;

console.log(typeof(newVariable));
console.log(Boolean(newVariable));
```

This makes sense, because JavaScript assigns types to variables based on the value of their content, and an undefined variable has no content yet.

Null Variables

Setting a variable to `null` unsets its value so that it essentially stops existing. Because `null` is an unknown value, JavaScript reports the type of a `null` variable as `object`.

```
var newVariable = 1;

console.log(typeof(newVariable));
console.log(Boolean(newVariable));

newVariable = null;

console.log(typeof(newVariable));
console.log(Boolean(newVariable));
```

`null` variables are considered `false` when trying to determine their boolean value. But does JavaScript consider `null` and `undefined` to be the same?

Null and Undefined - Not Exact Matches

`null` and `undefined` are treated as matches when you're doing comparisons, but they're not exact matches:

```
var notInitialized;
var setToNull = null;

console.log(notInitialized == setToNull);
console.log(notInitialized === setToNull);
```

Knowing the difference between these return values, and how JavaScript treats them when comparing them, is going to come in handy when you're

confronted with a bug that you don't understand and you're trying to figure out why a comparison is coming back different from the way that you would expect it to.