JavaScript Scope
The `var` Keyword

Learning objective: By the end of this lesson, students will understand the scope, behavior, and limitations of the `var` keyword in JavaScript, including how it differs from `let` and `const`.

In addition to defining variables with `let` and `const`, there is also a *third* way to define variables in JavaScript: by using the `var` keyword.

When JavaScript was created, `var` was the only way to define variables. It's the least restrictive of the three keywords, and it allows us to redeclare variables and declare variables without intializing them.

This lack of restriction can cause a lot of confusion and unintended behavior in our code, and is one of the reasons we don't use `var` (and you shouldn't either).

Regardless of whether they are defined within a block, variables declared with `var` always have function scope. This means they're accessible from anywhere within the function in which they were declared.

The following code from MDN's docs about let and docs about var demonstrates the differences between `let` and `var`:

```
const varTest = () => {
  var x = 1
  if (true) {
    var x = 2 // same variable!
    console.log(x) // 2
  }
  console.log(x) // 2
}

const letTest = () => {
  let x = 1
  if (true) {
    let x = 2 // different variable
    console.log(x) // 2
  }
  console.log(x) // 1
}
```

and another example of their differences:

```
if (true) {
  var x = "yes"
}
console.log(x) // "yes"

if (true) {
  let y = "yes"
}
console.log(y) // ReferenceError: y is not defined
```

`var` hoisting

We can call function declarations before they have been defined thanks to hoisting.

Similarly, a variable's **declaration** (but not its assignment) is hoisted to the top of the function when it's declared using `var` (but not when it's declared using `let` or `const`).

Hoisting in JavaScript moves all variable declarations to the top of their scope before code execution. This means that no matter where we declare a variable in our code, it will be accessible as if it were declared at the top of the scope.

For example, when we write code like this:

```
const hoist = () => {
  console.log(x)  // prints: undefined, not a ReferenceError
  var x = 25
  console.log(x)  // prints: 25
}
```

Internally, the JS engine actually sees this:

```
const hoist = () => {
  var x
  console.log(x) // prints: undefined, not a ReferenceError
  x = 25
  console.log(x) // prints: 25
}
```

Here are some things to remember about hoisting:

- Only variable declarations are hoisted, not function declarations.
- Hoisting moves variable declarations to the top of the scope, but **does not** initialize them. This means variables that are hoisted will have a value of `undefined` until they're initialized.
- If you're not careful, hoisting can cause **unexpected behavior**.