var versus let / const

> Besides `var`, we now have access to two new identifiers for storing values â`let` and `const`. Unlike `var`, `let` and `const` statements are not hoisted to the top of their enclosing scope.

An example of using `var`:

```
var snack = 'Meow Mix';

function getFood(food) {
    if (food) {
        var snack = 'Friskies';
        return snack;
    }
    return snack;
}

getFood(false); // undefined
```

However, observe what happens when we replace `var` using `let`:

```
let snack = 'Meow Mix';

function getFood(food) {
    if (food) {
        let snack = 'Friskies';
        return snack;
    }
    return snack;
}

getFood(false); // 'Meow Mix'
```

This change in behavior highlights that we need to be careful when refactoring legacy code which uses `var`. Blindly replacing instances of `var` with `let` may lead to unexpected behavior.

> **Note**: `let` and `const` are block scoped. Therefore, referencing block-scoped identifiers before they are defined will produce a `ReferenceError`.

```
console.log(x);

let x = 'hi'; // ReferenceError: x is not defined
```

> **Best Practice**: Leave `var` declarations inside of legacy code to denote that it needs to be carefully refactored. When working on a new codebase, use `let` for variables that will change their value over time, and `const` for variables which cannot be reassigned.

Replacing IIFEs with Blocks

> A common use of **Immediately Invoked Function Expressions** is to enclose values within its scope. In ES6, we now have the ability to create block-based scopes and therefore are not limited purely to function-based scope.

```
(function () {
    var food = 'Meow Mix';
}());

console.log(food); // Reference Error
```

Using ES6 Blocks:

```
{
    let food = 'Meow Mix';
}

console.log(food); // Reference Error
```