Function Scope

In this lesson, we'll explain the behavior of functions and their scopes.

WE'LL COVER THE FOLLOWING Inside the Function Body Redefining a Function Binding Global Values to Functions Altering Higher-Scope Values

Inside the Function Body

In Reason, data created inside the body of a function will not exist outside it. Only the value we return is accessible. For example, this code will produce an error:

```
let a = 10;
let b = 5;

let product = (a, b) => {
    let c = 20;
    a * b * c;
};

/* Accessing c outside the function */
c;
```

Redefining a Function

Because of let bindings, we can use the same identifier to define different functions. The latest definition will be selected by the compiler.

```
Js.log(foo(10));

let foo = (n) => n + 2;
Js.log(foo(10));
```

The cool thing to note is that, if we call the <code>foo()</code> function within the second definition, the compiler will use the **old definition** to compute a value. This is because the older definition is the only one that's available when the newer definition is being compiled:

```
let foo = (n) => n - 2;
let foo = (n) => foo(n) + 2;
Js.log(foo(10));
```

Instead of **recursively** computing (10 + 2) + 2, it computes (10 - 2) + 2. This confirms what we talked about in the <u>Identifiers</u> section: *older Let bindings are never lost when new ones are defined*.

One may think that this goes against the principle of recursion, but soon, we'll see that Reason supports recursion through another approach.

Binding Global Values to Functions

Since ReasonML is a static language, once a global variable has been referenced in a function, the function will remember that particular definition of the variable, even if it is redefined later on:

```
let str = "Hello";

let strTuple = (s: string) => {
    let t = (str, s);
    t;
};

Js.log(strTuple("World"));

/* Redefining str */
let str = "Educative";
```



In the second call, even though str has been reset to Educative, the function remembers the original value because that is the one which was present when the function was compiled.

Altering Higher-Scope Values

Since let bindings are immutable, a function cannot really affect the values of bindings which are outside its scope.

However, a function can change the contents of an array outside its scope. This is because the mutable array is being passed to the function.

Here's an example:

```
let arr = [| 10, 20, 30, 40 |];
let foo = (n) => {
    arr[0] = arr[0]* n;
}

Js.log(arr);
foo(10);
Js.log(arr);
```

In the next lesson, we will understand the art of **currying**.