Closures

A closure refers to the lexical context a function was declared in and the variables it has access to. Closures allow us to dynamically create functions, implement encapsulation, and create interfaces to interact with our code. Learn how to use them to perform each of these tasks.

Dynamically Generating Functions

Let's say we want to write a function that lets us add `10` to any number.

```
function add10(num) {
        return num + 10;
}
```

We want another function that lets us add 20 to any, and another that lets us add 30, and another that lets us add 40, and 50. Let's write them.

```
function add20() {
    return num + 20;
}

function add30() {
        return num + 30;
}

...
```

Writing all of these function becomes tiresome and requires a lot of code duplication, something we always want to avoid. We want an easier way to write these functions.

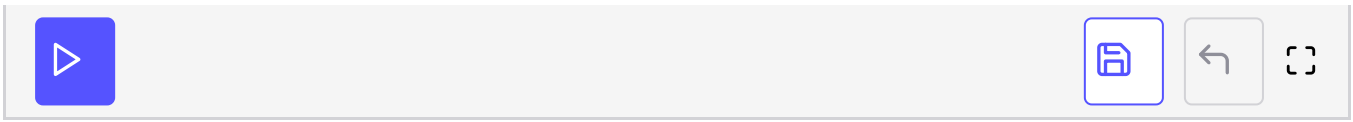
Scope Persistence

When a function is created, it retains access to the scope that it was created in. If it's created inside another function, it retains access to the scope of that outer function even after that outer function returns.

```
function fnGenerator(str) {
    const stringToLog = 'The string I was given is "' + str + '"';


    return function() {
        console.log(stringToLog);
    }
}

const fnReturned = fnGenerator('Bat-Man');
fnReturned(); // -> The string I was given is "Bat-Man"
```



We've just created a closure. MDN defines a closure as

> The combination of a function and the lexical environment within which that function was declared.

Even though `fnGenerator` has finished executing, its scope remains in memory and the returned function retains access to this scope. This is the concept we'll use to solve our code repetition problem.

The Solution

Knowing that functions can retain access to the scoped variables of their outer functions, we can write an exciting new type of function: generator functions, or factory functions.

A generator/factory function is a function that creates and returns functions based on what is passed in to it.

```
function addFactory(storedNum) {
    return function(num2) {
        return storedNum + num2;
    }
}
```

Let's walk through what this function is doing.

To invoke `addFactory`, we pass in a number and are returned a brand new function. This brand new function permanently keeps access to the `storedNum` we passed in. The new function itself takes a parameter and adds it to that

first number when invoked.

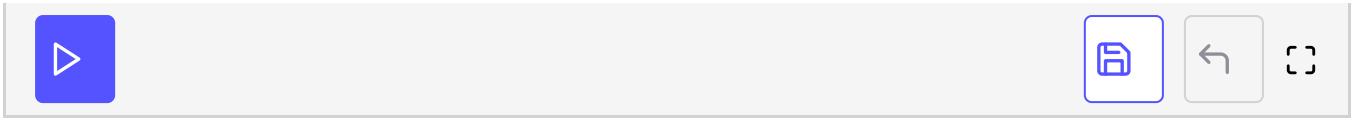
Every time we invoke `addFactory`, we create a brand new scope with the parameter we pass in. The function returned by `addFactory` has access to that specific scope. This is why the system works - **each returned function will have access to a different scope with a different `storedNum`. Each returned function will retain access to the variable that was given to `addFactory` at the time `addFactory` was invoked.**

Now we can dynamically generate those functions we were trying to write earlier.

```
function addFactory(storedNum) {
    return function(num2) {
        return storedNum + num2;
    }
}


const add10 = addFactory(10);
const add20 = addFactory(20);
const add30 = addFactory(30);

console.log(add10(5)); // -> 15
console.log(add20(6)); // -> 26
console.log(add30(7)); // -> 37
```



`add10` retains access to `10` and adds it to whatever we give it. Similarly, `add20` and `add30` retain access to `20` and `30` respectively.

We've gotten rid of the need to write all of those functions ourselves. We call our factory function and get new functions each time, each one doing a slightly different thing.

If we need functions that are mostly similar but slightly different, this is often a great technique to use.

Encapsulation

With this idea of scope persistence, we can write functions that hide data from someone who uses the function. This is called encapsulation. We might do this

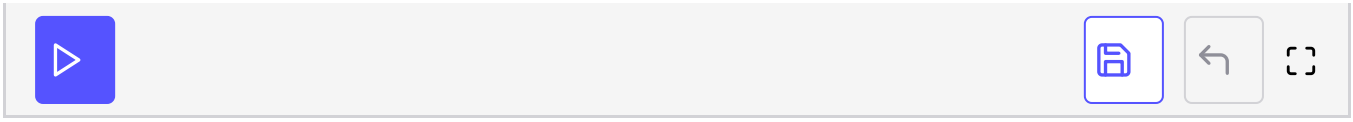
because we don't want a user manipulating a value in a way they're not intended to.

Say we're writing a function that keeps track of how many times it's been invoked. We'll need to make a `counter` variable and increment it by 1 every time it's called. We never want the user to change `counter` themselves - we want it only changeable by calling the function. Using this idea of scope persistence, we can make it so the only way a user can change `counter` is the way we want them to.

```
function counterGenerator() {
    let counter = 1;

    return function() {
        return counter++;
    }
}

const incrementCounter = counterGenerator();
console.log(incrementCounter()); // -> 1
console.log(incrementCounter()); // -> 2
counter = 100; // <- sets a new global variable 'counter';
               // the one inside counterGenerator is unchanged
console.log(incrementCounter()); // -> 3
```



We can give the user the `incrementCounter` function and hide the `counterGenerator` function. The only way to interact with the `counter` variable is to call `incrementCounter()`.

We might want to do this if we were creating a library. By using this concept, we can create an interface that lets the user interact with our code base without letting them break anything.

Consider driving a car. There are several different types of engines - gasoline, electric, even hydrogen-powered. Every car, no matter the engine, has the same controls - a steering wheel, turn signals, an air conditioner, window controls, etc.

The engine is hidden behind walls of plastic and metal. It's something the driver shouldn't have to worry about, so there's no point in showing it or letting the driver tinker with it. They might break something. No matter what type of engine it has, the car's interface, the controls, remain the same.

We try to write code the same way. If we ever wanted to update our library because we found a bug or we found a way to make a function work faster, we could push the changes to our code base. As long as we don't change the interface that the developer using our library interacts with, they won't have to change their code. Our library will silently update and their code will work the same.

That's it for closures.

Exercise

This is an extremely important concept. Feel free to try this exercise to see if you grasp the concept.

```
// Make this function take in a parameter.
// Make this function return a function.
// Make the returned function return whatever was
// originally passed in to func when it was called.

function func() {
    // Your code here
}
```