AkiraChix
SHE BUILDS • SHE SERVES • SHE LEADS

Outline

- Functions As Values
- Immediately Invoked Function Expressions (IIFEs)
- Closure
  - Modules

Functions As Values

Can a function be treated as a value?

Recall a function declaration syntax as follows:

```
function foo( ) {
    // ..
}
```

foo is basically a variable in the outer enclosing scope that's given a reference to the function being declared. i.e. the function itself is a value, just like 11, "hello", [1,2,3]

A function itself can be a value that's assigned to variables, passed to or returned from other functions.

As such, a function value should be thought of as an expression.

Consider:

```
// Anonymous function expression, since it has no name
var foo = function() {
        // ..
};

// Named function expression, assigned to x variable
var x = function bar(){
        // ..
};
```

Named function expressions are generally more preferable, though anonymous function expressions are still extremely common.

AkiraChix
SHE BUILDS • SHE SERVES • SHE LEADS

Immediately Invoked Function Expressions (IIFEs)

For a function to be executed, it has to be called. For instance, to execute the previous functions we call foo() or x()

However, there is a way to execute a function expression without calling it.

This is referred to as Immediately Invoked Function Expression (IIFE).

```
( function IIFE(){ .. } )
```

Where:

- the outer (..) is just a nuance of JS grammar to needed to prevent it from being treated as a normal function declaration.
- the final ( ) on the end of the expression is what actually executes the function expression referenced immediately before it.

Normal function expression

```
function foo() { .. }

// `foo` function reference expression,

// then `()` executes it

foo();
```

Its equivalent IIFE function expression

```
// `IIFE` function expression,

// then `()` executes it

( function IIFE(){ .. } )( );
```

Consider:

```
var a = 20;

(function IIFE(){
    var a = 10;
    console.log( a );    // 10
})( );
console.log( a );        // 20
```

Since IIFE maintain their own scope, they can be used to declare variables that won't affect the global scope variables.

IIFEs can also have return values

```
var x = (function IIFE(){
        return 20;
})();
x;              // 20
```

Closure

One of the most important concept in JS.

Closure is a way to "remember" and continue to access a function's scope (its variables) even once the function has finished running.

Consider:

```
function outer(x) {
        // parameter `x` is a variable within outer function

        // inner function `inner()` uses `x`, so
        // it has a "closure" over it
        function inner(y) {
                return y + x;
        };

        return inner;
}
```

The reference to the inner function gets returned with each call to the outer function

Closure is the ability of the inner function to remember whatever x value was passed in to the outer().

For example:

```
var plusOne = outer(1);

var plusTwo = outer(2);

plusOne(3);     // 4

plusTwo(4);     // 6
```

Explanation:

1. When we call outer(1), we get back a reference to inner(..) that remembers x as 1. We call this function reference plusOne(..).
2. When we call plusOne(3), it adds 3 (its inner y) to the 1 (remembered by x), and we get 4 as the result.

Modules

The most common usage of closure in JS is the module pattern.

Modules let you define private implementation details (variables, functions) that are hidden from the outside world, as well as a public API that is accessible from the outside.

Consider:

```
function User(){
        var username, password;
        function doLogin(user,pw) {
                username = user;
                password = pw;
                // do the rest of the login work
        }
        var publicAPI = {
                login: doLogin
        };
        return publicAPI;
}
```

```
// create a `User` module instance

var janeDoe = User();


// user to access login API

janeDoe.login( "janeDoe", "mypwd123" );
```

The User() function serves as an outer scope that holds the variables username and password, as well as the inner doLogin() function.

These are all private inner details of this User module that cannot be accessed from the outside world.

Each time we execute User(), a new instance of the User module (with its scope) is created.

The inner doLogin() function has a closure over username and password, meaning it will retain its access to them even after the User() function finishes running.

publicAPI is an object with one property/method on it, login, which is a reference to the inner doLogin() function.

When we return publicAPI from User(), it becomes the instance we call janeDoe.

Assignment

Write a program Kiosk( ) that will allow you to add a fruit and retrieve a list of fruits to/from a kiosk using setter and getter methods.

For instance:

```
var kiosk = Kiosk( )
kiosk.setFruit("mango")
kiosk.getFruits() will return a list of fruits  ["mango"]
```

Next class

- *this* Identifier
- Classes
- Prototypes
- Old & New
  - Polyfilling
  - Transpiling

Course material: https://github.com/getify/You-Dont-Know-JS/blob/master/up%20%26%20going/ch2.md

Video of the week - https://www.youtube.com/watch?v=tJKIDTX9oXg