A Drip of JavaScript

An Introduction to IIFEs - Immediately Invoked Function Expressions

Originally published in the A Drip of JavaScript newsletter.

It doesn't take very long working with JavaScript before you come across this pattern:

```
(function () {
    // logic here
})();
```

Your first encounter is likely to be quite confusing. But fortunately the concept itself is simple. The pattern is called an immediately invoked function expression, or IIFE (pronounced "iffy").

In JavaScript functions can be created either through a function declaration or a function expression. A function declaration is the "normal" way of creating a named function.

```
// Named function declaration
function myFunction () { /* logic here */ }
```

On the other hand, if you are assigning a function to a variable or property, you are dealing with a function expression.

```
// Assignment of a function expression to a variable
var myFunction = function () { /* logic here */ };

// Assignment of a function expression to a property
var myObj = {
    myFunction: function () { /* logic here */ }
};
```

A function created in the context of an expression is also a function expression. For example:

```
// Anything within the parentheses is part of an expression
(function () { /* logic here */ });

// Anything after the not operator is part of an expression
!function () { /* logic here */ };
```

The key thing about JavaScript expressions is that they return values. In both cases above the return value of the expression is the function.

That means that if we want to invoke the function expression right away we just need to tackle a couple of parentheses on the end. Which brings us back to the first bit of code we looked at.

```
(function () {
    // logic here
})();
```

Now we know what the code is doing, but the question "Why?" still remains.

The primary reason to use an IIFE is to obtain data privacy. Because JavaScript's `var` scopes variables to their containing function, any variables declared within the IIFE cannot be accessed by the outside world.

```
(function () {
    var foo = "bar";

    // Outputs: "bar"
    console.log(foo);
})();

// ReferenceError: foo is not defined
console.log(foo);
```

Of course, you could explicitly name and then invoke a function to achieve the same ends.

```
function myImmediateFunction () {
    var foo = "bar";

    // Outputs: "bar"
    console.log(foo);
}

myImmediateFunction();

// ReferenceError: foo is not defined
console.log(foo);
```

However, this approach has a few downsides. First, it unnecessarily takes up a name in the global namespace, increasing the possibility of name collisions. Second, the intentions of this code aren't as self-documenting as an IIFE. And third, because it is named and isn't self-documenting it might accidentally be invoked more than once.

It is worth pointing out that you can easily pass arguments into the IIFE as well.

```
var foo = "foo";

(function (innerFoo) {
    // Outputs: "foo"
```

```
    console.log(innerFoo);
})(foo);
```

And that's the story behind IIFEs. Soon we'll be building on this by taking a look at the module pattern in JavaScript.

Thanks for reading!

Joshua Clanton
