Published in Functional JavaScript

Santosh Rajan Follow
Mar 24, 2014 · 11 min read · Listen

Introduction to Functional JavaScript

History, programming paradigms, abstractions, basics

History

JavaScript was a functional programming language even before it got its name! Back in 1995 Netscape Communications the makers of the Netscape Browser realized that their browser needed a scripting language embedded in HTML. They hired Brendan Eich to embed the Scheme programming language in HTML. Scheme was a full fledged functional programming language.

To his dismay “the powers that be” encouraged Brendan to come up with a more traditional programming language. His first cut was a language called “Mocha” in May 1995, and it retained that name till December 1995. It was first renamed “LiveScript” and soon after that Netscape and Sun did a license agreement and it was called “JavaScript”. But by then Brendan Eich had sneaked in some “functional programming” features into JavaScript.

The story gets more complicated, and we will delve into it. Because this story will tell you why JavaScript is what it is today. Brendan Eich claims that hiring him to embed the Scheme language, might actually have been a “bait and switch operation”. But at that point in time Netscape was also negotiating with Sun to embed Java in the browser. Note that JavaScript was for embedding in HTML while Java was for embedding in the Browser. The idea was that Java would be used for component development while JavaScript would be used for lightweight scripting within HTML.

We don’t know what actually transpired, but the orders from above to Brendan where clear. The new scripting language must “look like Java” and must be “object based”. Any hopes Brendan might have harboured for Scheme, where now out of the window. We will see why.

Curly brace language

```
for (int i = 0; i < 10; i++) {
 System.out.println("Hello");
 System.out.println("World");
}
```

The same for-loop in C.

```
int i;
for (i = 0; i < 10; i++) {
 printf("Hello\n");
 printf("World\n");
}
```

And the for-loop in JavaScript.

```
for (var i = 0; i < 10; i++) {
 console.log("Hello")
 console.log("World")
}
```

Programming Paradigms

Indeed JavaScript does look like Java. And C too. However curly braces were not the only implications for a language to "look like Java". Java is an imperative/object oriented style programming language. JavaScript also had to be an imperative/object oriented style language.

Programming languages are made up of operators, conditional statements, loop statements and functions. Having conditional statements and loop statements are hallmarks of an "imperative" style programming language. Functional style languages tend to support operators and functions only.

It is interesting that none of the three languages, Java, C++ and C, were functional programming languages. While C was an imperative programming language, C++ and Java were Imperative/Objected Oriented programming languages. By now you would have guessed that the three programming paradigms (styles) were imperative, object oriented and functional. There is one more, the declarative paradigm.

The differences between these paradigms are because of the foundations on which they were based




logic". In this chapter we will look at the differences between, imperative, object oriented and functional programming at a more practical level.

Imperative language

An imperative programming language is one in which program state change is achieved by executing a series of statements, and does flow control primarily using conditional statements, loop statements and function calls. The program given below is a simple implementation of the JavaScript Array.join method in an imperative manner.

```
function simpleJoin(stringArray) {
 var accumulator = ''
 for (var i=0, l=stringArray.length; i < l; i++) {
   accumulator = accumulator + stringArray[i]
 }
 return accumulator
}
```

Object oriented language

The code above is straight forward. We iterate through an array and add each string element to the accumulator and return the accumulator. We will now rewrite this function in an object oriented manner. Since JavaScript has an Array class, we will add this method to the Array class, so that every instance of this class has access to this function. JavaScript use prototypal inheritance and so we add this function to the Array prototype.

```
Array.prototype.simpleJoin = function() {
 var accumulator = ""
 for (var i=0, l=this.length; i < l; i++) {
   accumulator = accumulator + this[i]
 }
 return accumulator
}
```

As we can see, the object oriented version is quite like the imperative version, except that the function (method) is now a method of the class. Object oriented languages tend to be imperative languages also.

Functional language




```
function simpleJoin(stringArray, i, accumulator) {
 if (i === stringArray.length) {
   return accumulator
 } else {
   return simpleJoin(
            stringArray, i+1, accumulator+stringArray[i])
 }
}
```

The first thing to note is that we are not using the for loop here for iteration. Instead we use recursion for iteration. Recursion happens when the function calls itself from within itself. Indeed this is one of the characteristics of a functional programming language. eg. Scheme does not have any loop statements. Instead it uses recursion for iteration. The function is called for the first time with the given array in stringArray, **i** set to **0**, and accumulator set to **“”**. The second time around the function is called from within itself with the same stringArray, **i** set to **i + 1**, and accumulator set to **accumulator + stringArray[i]**. And we continue the same way until **i === stringArray.length** when we return the accumulator. We will discuss recursion in detail in a later chapter. Just remember we used recursion for doing iteration here.

There is still something imperative about this function. The if statement. Functional languages tend to use expressions that evaluate to some value, instead of statements that don’t evaluate to anything. So let us rewrite the function, to make it as functional as possible in JavaScript.

```
function simpleJoin(stringArray, i, accumulator) {
 return (i === stringArray.length) ? accumulator :
   simpleJoin(stringArray, i + 1, accumulator + stringArray[i])
}
```

Now this is as functional as you can get with JavaScript. Instead of the **if** statement we return the value evaluated by the conditional operator **?**. The conditional operator **?** Takes a conditional expression and returns the value of one of the two expressions based the condition being true or false. The value of the first expression is returned if true and the second if false.

We can see that the functional version is concise. Indeed one of the advantages of functional programming is that it lends itself to lesser code to accomplish the same thing, leading to better readability and maintainability.




around this problem in the chapter on recursion. As of writing this book tail call optimization is expected in ecmascript 6.

Multi paradigm language

So is JavaScript an imperative language, or an object oriented language, or a functional language? It is a multi paradigm language. It does not have all the functional features implemented. But it is slowly getting there. This is also true of most other languages. Most languages (other than functional languages to begin with) have added functional features to various degrees over the years. A good example of the multi paradigm nature of JavaScript is the Array.forEach method. Here is a possible simple implementation. Note that all modern browsers have already implemented this.

```
Array.prototype.forEach = function(callback) {
 for (var i = 0, len = this.length; i < len; ++i) {
   callback(this[i], i, this)
 }
}
```

In the code above the for-loop part of the code is imperative. Adding to the array prototype and usage of **this** is object oriented. Passing a function as an argument to another function (callback) is functional and is a feature of functional programming known as “higher order function”. In JavaScript, we take this for granted, passing functions as an argument. Surprisingly this was not a feature found in the most popular languages until recently. eg. You cannot pass functions as arguments in Java, though you can do it indirectly via Interfaces. Same is the case with C, though you can do it indirectly using pointers.

Programming Abstractions

What if JavaScript did not have the “passing functions as arguments feature”? The answer to this question is crucial to understanding what functional programming brings to the table. For one, we would not be able to write the **Array.forEach** function above. And even worse, every time we had to iterate over an array, we would have to write the same for-loop again and again. If we had **Array.foreach** we need to think only about writing the callback. We need to think only of what to do with each element of the array, and need not concern ourselves with iterating over the array. In other words we have “abstracted” away the iteration part of the problem, and freed ourselves to concentrate on each element of the array.

Abstractions are about hiding away implementation details, thereby reducing and factoring out

functional programming we use functional features like first class functions, nested functions, anonymous functions and closures to abstract away code and sometimes even data. Monads are an esoteric feature of functional programming that can even abstract away program structure!

Macro's are another feature that allows code abstraction. However macros are not a feature of functional programming. But they are [illegible]tional languages like Lisp, Scheme, Clojure etc. There are attempts to bring Macro's to JavaScript and at the moment it is very much early days.

A good example of the power of functional abstractions is **jQuery**. **jQuery** is the mother of all functional abstractions. It has abstracted away the JavaScript language itself! So much so that you wouldn't be surprised, if you found a jQuery programmer who knows very little or no JavaScript. As of Feb 2013 there was one publisher who had 32 jQuery books listed, and not a single JavaScript book! And jQuery has achieved so much mostly using only two functional features, functions as arguments and closures.

First Class Functions

The functional programming feature that Brendan Eich implemented in JavaScript was "first class functions". Functions are first class if they are treated as "first class citizens" of that language. Which implies functions are treated just like all other variables. ie. You can pass them as arguments to functions, you can return them as values from other functions, or you can assign them to variables or data structures. We have seen "passing functions are arguments" earlier. Here is an example of assigning a function to a variable.

```
function greet(name) {
 console.log('Hello ' + name)
}

greet("John") // "Hello John"

var sayHello = greet
sayHello("Alex") // "Hello Alex"
```

Some programming language theorists consider "anonymous functions" as first class functions. Not to be outdone, Brendan Eich threw anonymous functions into the mix. This is like letting the cat among the pigeons so to speak. But not for Brendan Eich, he knew the solution to the problem. Here is an anonymous function in JavaScript




```
}
```

If you noticed we did not give this function a name. After all, it is an anonymous function. If you try to run the code above, you will get an error. Something to the effect “you cannot run the code in this context”. And rightly so. They can only be assigned to something, or passed as arguments to a function.

```
var sayHello = function(name) {
 console.log("Hello " + name)
}
sayHello(“Jane”) // “Hello Jane”
```

What if we wanted to change the greeting? Sometimes we would like to say “Hi” instead of “Hello”. We could create a generic “createGreeting” function, which would in turn “compose” another function for you, and return the new composed function. So if we wanted to say “Hi” it would return a function, and if we wanted to say “Hello” it would return another function that says “Hello”. We can do all that because JavaScript supports first class functions, and we can return functions from other functions. Here is the code.

```
function createGreeting(greeting) {
 return function(name) {
   console.log(greeting + “ “ + name)
 }
}
var sayHi = createGreeting(“Hi”)
sayHi(“Jack”) // “Hi Jack”
var sayHello = createGreeting(“Hello”)
sayHello(“Jack”) // “Hello Jack”
```

The **createGreeting** function takes a greeting as its argument. The function returns a newly created anonymous function. However the newly created anonymous function was created inside another function **createGreeting**. So it is also a nested function now. Now since our language supports anonymous functions it will also have to support nested functions. And when we return nested functions from our function we run into another problem. We will look at that in more detail.

Scope




not true of the variable **greeting**. It is defined in another function called createGreeting and hence is “non local” to the anonymous function. However the anonymous function can access the variable greeting due to something called lexical scoping.

The “scope” of a variable is its “visibility” within a program. “Lexical scope” means that visibility is limited to all the text (code). So when we say “local variables are lexically scoped” within a function, it means that the function’s local variables are visible to all the text (code) in the function, even if the code is within another nested function. This also means that when you run the nested function outside the lexically scoped environment, the nested functions non local variable will not be visible. And there lies the problem of returning nested functions from another function. And indeed thats what we are doing here.

```
var sayHi = createGreeting(“Hi”)
```

In the line above we assign the returned anonymous function to variable sayHi. And call the function in the next line.

```
sayHi(“Jack”)
```

We are calling sayHi outside of createGreeting. And the **greeting** variable is not available outside of createGreeting. The variables it may access in the scope where it was defined, may not be available in the scope where it was actually called. Thats why languages like C don’t support nested functions. For this to work the language needs to support another functional programming feature called “closures”. JavaScript supports closures. As matter of fact it has to support closures. Any language that supports first class functions and nested functions has to support closures.

Closure

A function’s closure is a reference to all its non local variables. In the previous example **greeting** was the non local variable and **name** was the local variable. A closure is a table of references to all of a functions non local variables. This allows the function to continue to refer to the non local variable even when the function is out of the scope of the variables.

Creating Closures is a very important pattern used in functional programming. It hides away parts of




When programming in a functional style, creating a closure is something you do by default. It should come to you naturally. You will see creation of closures in almost every chapter of this book. Recursion, currying, composition, memoization, modules all create closures.