On Closures in Computer Languages

# Version Information

Draft version. Alpha, 1.1

Hope to take it to a Beta soon.

Started: 8/19/23

Contents

[Version Information 1](#_Toc143364947)

[Goal/Note 1](#_Toc143364948)

[So, what is a closure: A function with state? 2](#_Toc143364949)

[Understand Ordinary, Regular Run-of-the-mill Functions 2](#_Toc143364950)

[Functions are stateless. 2](#_Toc143364951)

[Functions can have side effects. 2](#_Toc143364952)

[Function as an instance and an object on its own merit 2](#_Toc143364953)

[Stateful Functions 2](#_Toc143364954)

[How do you initialize the state for a function 2](#_Toc143364955)

[Two stateful functions of the same name are not the same instance 2](#_Toc143364956)

[Why are they called Closures? 2](#_Toc143364957)

[How are closures implemented in Object Oriented languages? 3](#_Toc143364958)

[Closures in various languages 3](#_Toc143364959)

[Python 3](#_Toc143364960)

[Javascript 3](#_Toc143364961)

[Java 3](#_Toc143364962)

[Julia 3](#_Toc143364963)

[Uses of Closures 3](#_Toc143364964)

# Goal/Note

1. With the introduction of functional programming languages closures are often used
2. Closures are not very familiar to programmers that have grown up with the older procedural or somewhat newer object-oriented languages.
3. Neither the name “closure” intuitively tell you what it is: is it an object? Is it a function? In either case how do you understand it?
4. Let’s explore closures from a variety of angles.
5. Also briefly touch on a variety of uses that this programming facility is applied towards.

# So, what is a closure: A function with state?

The easiest way to understand is to see it as a function with (hidden) state.

That must come across to you as a surprise.

Let’s talk about functions first.

# Understand Ordinary, Regular Run-of-the-mill Functions

I start with the premise that everyone in programming are familiar with regular functions. Without covering the basics let me touch upon a few key aspects.

## Functions are stateless.

Functions don’t have state. Functions are state-less. They have inputs and outputs but no state. Clearly.

Functions are idem potent, ideally. This just means you can call a function repeatedly any number of times, and the later invocations are not impacted by what happened in the prior invocations.

## Functions can have side effects.

Of course, unless the function is operating on a global state such as updating a record in a database.

These are sometimes called side effects. Side effects of calling the function.

In a pure function there are no side effects

## Function as an instance and an object on its own merit

Usually, a function is code written with a name and inputs and outputs. In that sense its structure and behavior, static. The functions are known by their names. Those names are directly used to invoke them (call them). They are so to say, in the name space. Like peoples names in real life.

In that sense they are not indirect pointers. When it is a pointer, then that pointer can point to any of a number of functions. This is an indirect reference. And sometimes called higher level functions.

Unlike a named function, when references (pointers) to functions are allowed in a programming language, they can be seen as instances of a function that can be passed around and applied like objects. Or instances.

When a function is stateless there is only one copy of that function even if it is passed around as a pointer (or reference).

# Stateful Functions

If a function needs to remember what happened during its last invocation, we get into trouble.

One way to do this is to maintain a global variable that is outside of this function.

This means that the global variable is visible to any number of functions.

And those functions can read or manipulate that variable or object.

This behavior is not desirable because one can get into ownership to that variable and updates by multiple threads leading to race conditions etc. Furthermore, that global variable will unnecessarily introduce a name into the name space.

So, there are two key goals for implementing stateful functions:

1. State needs to be maintained.
2. State needs to be hidden.
3. Function needs to have read and write access to the state

## How do you initialize the state for a function

Consider a function,

f(x): a

where “x” is the argument and “a” is the state.

The same function f(x) may be used for different initial states.

f(x): b, f(x): c

where “b” and “c” are different states that you want f(x) to manipulate.

So, one must be aware,

f(1), f(2), f(3), f(1), f(2), f(3)

is not enough clarity, because each f(x) may have been tied to a different initial state (a,b, or c).

So, suddenly, f(x) is now sensitive to its state context, so

f(1):a is different from f(1):b

This would not have been the case if f(x) is stateless. Then

f(1) is same f(1) had they been stateless

So, that leads the following observation,

*“Two stateful functions of the same name are not the same instance.”*

So, you cannot use function name for your invocation anymore! Instead, you have to do something like this:

fa = f(x):a

fb = f(x):b

//Then use them

fa(1)

fb(1)

//Where fa(x) and fb(x) are pointers obtained dynamically

The syntax in languages differs significantly from how the pointers to the function that are specific to their initial state are obtained.

## Managing state with Objects, a quick look

This would have been straight forward in OO languages using objects and their embedded functions

class SomeClass

{

internal-state y;

//Step 1:

//Creator function to initialize the state

SomeClass(state) { y = state }

//Step 2:

//Now the actual function to manipulate that state

f(x){ //do something with y}

}

//Step1: Create the function with state “a”

fa = create SomeClass(a);

//Step2: Start using it

fa.f(x)

fa.f(x)

fa.f(x)

//Create the function with state “b” and use it multiple times

fb = create SomeClass(b)

fb.f(x)

fb.f(x)

fb.f(x)

//Or some language syntactic sugar can be put in place to do

fa = create SomeClass(a)

//use it

fa(x)

fa(x)

fa(x)

//and…. for “b”

fb = create SomeClass(b)

fb(x); fb(x); fb(x); //etc.

## The Two-Shot need for Stateful functions

So when a function is needed to be aware of its state, there are two steps before that function can be used

1. Initialize the function and get a pointer to the initialized function with the desired state.
2. Use the pointer that is gotten from step 1 to use/manipulate that state.

Typically, there are two functions each representing step 1 and step 2 above.

The first one is similar to a constructor in OO languages and the second one is the actual function that has access to the state that is setup by the constructor. These functions may go by different names.

# Why are they called Closures?

Can Closures update state?

Can closures return some aspect of state to the caller?

Can two closures operate on the same state?

# How are closures implemented in Object Oriented languages?

# Closures in various languages

## Python

## Javascript

## Java

## Julia

# Uses of Closures

To create interfaces