Ruby Programming Language
Closures

Sapna Jain

Nov 30, 2004

Ruby Programming Language

Ruby is an object oriented programming language. Every bit of data is an object, even the primitive data types are also objects.

Ruby is a multi-paradigm programming language, it allows to program procedurally, object-orientated or functionally.

Ruby support blocks and closures.

Blocks

1. Blocks are nameless functions. Basically we can pass nameless function to another function, and then that function can invoke the nameless function.
2. In C the same functionality is provided by function pointers. We can pass a function pointer as parameter to other function, which can invoke function using function pointers.

Blocks Vs Function Pointers

With function pointers, we have to explicitly specify that a function will accept function pointer as argument.

But, in Ruby block is considered as an implicit argument, and thus any function can call block using *y*ield keyword.

Use of Blocks

1. Blocks must follow a method invocation:
   invocation do ... end
   invocation ...
2. Blocks remember their variable context, and are full closures.
3. Blocks are invoked via yield and may be passed arguments.
4. Brace form has higher precidence and will bind to the last parameter if invocation made w/o parens.
5. do/end form has lower precidence and will bind to the invocation even without parens.

Closures

In programming languages, a closure is an abstraction representing a function, plus the lexical environment (see static scoping) in which the function was created, and its application to arguments. A closure results in a fully closed term: one with no free variables left.

Closures typically appear in languages that allow functions to be "first-class" values in other words, such languages allow functions to be passed as arguments, returned from function calls, bound to variable names, etc., just like simpler types such as strings and integers.

What makes a block a closure?

A closure object has code to run, the executable, and state around the code, the scope. So you capture the environment, namely the local variables, in the closure. As a result, you can refer to the local variables inside a closure. Even after the function has returned, and its local scope has been destroyed, the local variables remain in existence as part of the closure object. When no one refers to the closure anymore, it's garbage collected, and the local variables go away.

So the local variables are basically being shared between the closure and the method. If the closure updates the variable, the method sees it. And if the method updates the variable, the closure sees it.

Benefits of closure

You can reconvert a closure back into a block, so a closure can be used anywhere a block can be used. Often, closures are used to store the status of a block into an instance variable, because once you convert a block into a closure, it is an object that can by referenced by a variable. And of course closures can be used like they are used in other languages, such as passing around the object to customize behavior of methods. If you want to pass some code to customize a method, you can of course just pass a block. But if you want to pass the same code to more than two methods – this is a very rare case, but if you really want to do that – you can convert the block into a closure, and pass that same closure object to multiple methods.

Example of closure

Let we have a user interface with two buttons. bStart =

Button.new("Start")
bPause = Button.new("Pause")
#...

What happens when the user presses one of our buttons? In the Button class, the hardware folks rigged things so that a callback method, buttonPressed, will be invoked. The obvious way of adding functionality to these buttons is to create subclasses of Button and have each subclass implement its own buttonPressed method.

Example Continued...

```
class StartButton ¡ Button
    def initialize
       super(”Start”) #invoke Button’s initialize
    end
    def buttonPressed
       do start actions...
    end
end
bStart = StartButton.new
```

Example Continued...

Problem with previous code

There are two problems here.

1. This will lead to a large number of subclasses. If the interface to Button changes, this could involve us in a lot of maintenance.
2. The actions performed when a button is pressed are expressed at the wrong level; they are not a feature of the button, but are a feature of the jukebox that uses the buttons. We can fix both of these problems using blocks.

Example Continued...

```
class JukeboxButton ¡ Button
    def initialize(label, &action)
       super(label)
       @action = action
    end
    def buttonPressed
       @action.call(self)
    end
end
bStart = JukeboxButton.new(”Start”)  songList.start
bPause = JukeboxButton.new(”Pause”)  songList.pause
```

Example Continued...

The key to all this is the second parameter to JukeboxButton#initialize. If the last parameter in a method definition is prefixed with an ampersand (such as &action), Ruby looks for a code block whenever that method is called. That code block is converted to an object of class Proc and assigned to the parameter. You can then treat the parameter as any other variable. In our example, we assigned it to the instance variable @action. When the callback method buttonPressed is invoked, we use the Proc#call method on that object to invoke the block. So what

exactly do we have when we create a Proc object? The interesting thing is that it's more than just a chunk of code. Associated with a block (and hence a Proc object) is all the context in which the block was defined: the value of self, and the methods, variables, and constants in scope. Part of the magic of Ruby is that the block can still use all this original scope information even if the environment in which it was defined would otherwise have disappeared. This facility is called a closure.

Example of method proc

This example uses the method proc, which converts a block to a Proc object.

```
def nTimes(aThing)
    return proc —n— aThing * n
end
p1 = nTimes(23)
p1.call(3)  69
p1.call(4)  92
p2 = nTimes(”Hello ”)
p2.call(3)  ”Hello Hello Hello ”
```

The method nTimes returns a Proc object that references the method's parameter, aThing. Even though that parameter is out of scope by the time the block is called, the parameter remains accessible to the block.

References

- Artima Developers
  http://www.artima.com/intv/closures2.html
- Programming Ruby
  http://www.rubycentral.com/book/tut_containers.html
- Ruby Programming Language
  http://en.wikipedia.org/wiki/Ruby_programming_language