Closures

Procs and lambdas also have another special attribute. When you create a Ruby proc, it captures the current execution scope with it.

They don’t carry the actual values, but a reference to them, so if the variables change after the proc is created, the proc will always have the latest version.

```
def call_proc(my_proc)
  count = 500
  my_proc.call
end

count   = 1
my_proc = Proc.new { puts count }

p call_proc(my_proc)
```

It would seem like 500 is the most logical conclusion, but because of the ‘closure’ effect this will print 1.

Binding Class

The Proc class defines a method named binding. Calling this method on a proc or lambda returns a Binding object that represents the bindings in effect for that closure.

```ruby
def return_binding
  foo = 100
  binding
end

puts return_binding.class
puts return_binding.eval('foo')

# If you try to print foo directly you will get an error.
# The reason is that foo was never defined outside of the method.
puts foo
```

When you create a proc or a lambda, the resulting

Proc object holds not just the executable block but also bindings for all the variables used by the block.

You already know that blocks can use local variables and method arguments that are defined outside the block. In the following code, for example, the block associated with the iterator uses the method argument :

```
# Return a lambda that retains or "closes over" the argument n
def multiplier(n)
lambda {|data| data.collect{|x| x*n } }
end
doubler = multiplier(2)
 # Get a lambda that knows how to double
puts doubler.call([1,2,3]) # Prints 2,4,6
```

Another way to

say this is that the variables used in a lambda or proc are not statically bound when the lambda or proc is created. Instead, the bindings are dynamic, and the values of the variables are looked up when the lambda or proc is executed.

As an example, the following code defines a method that returns two lambdas. Because the lambdas are defined in the same scope, they share access to the variables in that scope. When one lambda alters the value of a shared variable, the new value is available

to the other lambda:

```
# Return a pair of lambdas that share access to a local variable.


# Return a pair of lambdas that share access to a local variable.
def accessor_pair(initialValue=nil)
value = initialValue # A local variable shared by the returned lambdas.
getter = lambda { value }
 # Return value of local variable.
setter = lambda {|x| value = x }
 # Change value of local variable.
return getter,setter
 # Return pair of lambdas to caller.
end
getX, setX = accessor_pair(0) # Create accessor lambdas for initial value 0.
puts getX[]
 # Prints 0. Note square brackets instead of call.
setX[10]
```

```
 # Change the value through one closure.
puts getX[]
 # Prints 10. The change is visible through the other.
```