Practically Groovy: Functional programming with curried closures

Groovy's everyday coding construct goes where no closure has gone before

Ken Barclay
John Savage
Andrew Glover
Scott Davis

August 23, 2005

Closures are everywhere in Groovy, and the only problem with Groovy closures is that they can start to seem, well, kind of bland when you use them every day. This month, guest authors Ken Barclay and John Savage show you how to spice up standard closures recipes like closure composition and the Visitor design pattern with just a hint of curry. The curry() method was invented by Haskell Curry and has been in the Groovy language since before the JSR-compliant releases.

View more content in this series

Since the inception of the Practically Groovy series almost a year ago, I've given you several opportunities to get to know closures. When I first wrote about Groovy as part of the alt.lang.jre series ("Feeling Groovy," August 2004), I introduced Groovy's closure syntax, and just last month I showed you the recent JSR-compliant updates to that same syntax. From your studies so far, you know that Groovy closures are code blocks that can be referenced, parameterized, passed as a method parameter, and delivered as the return value from a method call. What's more, they can also be parameters to and return values from other closures. Because closures are objects of type Closure, they can also be a property of a class or a member of a collection.

While none of this stuff is exactly bland, the closure techniques (or should I say recipes) you'll learn this month are definitely a little more spicy than what you've tried so far. Guest authors John Savage and Ken Barclay have been doing some interesting experiments with the curry() method in Groovy closures, and this month we're lucky enough to get a taste of what they've cooked up.

About this series

The key to incorporating any tool into your development practice is knowing when to use it and when to leave it in the box. Scripting languages can be an extremely powerful addition to
Not only will Barclay and Savage's curried closures rewhet your appetite for familiar operations such as composition and the Visitor design pattern, they'll also open the door to functional programming with Groovy. Think of it as fusion cooking, but with Groovy in the pot.

**Pass the curry, please!**

While spicy curries are usually found at good Indian restaurants, *curried functions* are more typically found in functional programming languages such as ML and Haskell (see Related topics). The term *curry* is taken from Haskell Curry, the mathematician who developed the concept of partial functions. Currying refers to taking multiple arguments into a function that takes many arguments, resulting in a new function that takes the remaining arguments and returns a result. The exciting news is that the current release of Groovy (version jsr-02 at the time of writing) supports the `curry()` method for closure objects -- which means that we, the citizens of Planet Groovy, now get to take advantage of some aspects of functional programming!

You've probably never cooked with `curry()` before, so we'll start from a simple, familiar base. Listing 1 shows a closure referenced as `multiply`. It has formal parameters `x` and `y` and returns the product of these two values. The code then demonstrates two methods for executing the `multiply` closure: explicitly (via the `call` notion) or implicitly, assuming no ambiguity exists. The latter style gives rise to a function-call notation.

**Listing 1. Simply closures**

```groovy
def multiply = { x, y -> return x * y } // closure
def p = multiply.call(3, 4)             // explicit call
def q = multiply(4, 5)                  // implicit call
println "p: \$p"                        // p is 12
println "q: \$q"                        // q is 20
```

This closure is all well and good, but we're going to curry it up just the same. When calling the `curry()` method you need not supply the full complement of actual parameters. The `curried` call gives rise to the partial application of the closure. The *partial application* of a closure is another Closure object in which some values have been fixed.

**Listing 2. Curried closures**

```groovy
def multiply = { x, y -> return x * y }  // closure
def triple = multiply.curry(3)          // triple = { y -> return 3 * y }
def quadruple = multiply.curry(4)       // quadruple = { y -> return 4 * y }
def p = triple.call(4)                  // explicit call
def q = quadruple(5)                    // implicit call
println "p: \$p"                        // p is 12
println "q: \$q"                        // q is 20
```

Listing 2 demonstrates the currying of the `multiply` closure. In the first example, the value for parameter `x` has been set as 3. Effectively the closure, referenced as `triple`, now has the definition of `triple = { y -> return 3 * y }`. 

Practically Groovy: Functional programming with curried closures
As you can see, the parameter `x` has been removed from the definition of `multiply` and all occurrences have been replaced with a value of 3.

**Curried math 101**

As you might know from basic math, the multiplication operator is *commutative* (in other words, `x * y = y * x`). The subtract operator, however, is not commutative; therefore, you need two operations to handle the value to be subtracted and the value from which the subtraction will take place. Listing 3 defines the closures `lSubtract` and `rSubtract` (respectively left and right) for this purpose and consequently, shows an interesting application of the `curry` function.

**Listing 3. Left and right operands**

```groovy
def lSubtract = { x, y -> return x - y }
def rSubtract = { y, x -> return x - y }
def dec = rSubtract.curry(1) // dec = { x -> return x - 1 }
def cent = lSubtract.curry(100) // cent = { y -> return 100 - y }
def p = dec.call(5)                      // explicit call
def q = cent(25)                         // implicit call
println "p: \$p"                        // p is 4
println "q: \$q"                        // q is 75
```

**Iteration and composition**

You’ll recall from previous articles in this series that closures are commonly used with *iterator methods* applied to `List` and `Map` collections. The iterator method `collect`, for example, applies a closure to every element in a collection and returns a new collection with the new values. Listing 4 illustrates applying the `collect` method to a `List` and a `Map`. The `List` referenced as `ages` is sent the `collect()` method with the single closure `{ element -> return element + 1 }` as a parameter. Note that where the final parameter to a method is a closure, Groovy permits that you can remove it from the list of actual parameters and place it immediately after the closing parenthesis. When there are no actual arguments then the parentheses can be omitted. This is shown with the `collect()` method called with the `Map` object referenced as `accounts`.

**Listing 4. Closures and collections**

```groovy
def ages = [20, 30, 40]
def accounts = ['ABC123' : 200, 'DEF456' : 300, 'GHI789' : 400]
def ages1 = ages.collect({ element -> return element + 1 })
def accounts1 = accounts.collect
```

The final example collects all the elements from the `List` referenced as `ages` and applies the `dec` closure (from Listing 3) to them.

Practically Groovy: Functional programming with curried closures
Closure composition

Perhaps one of the more important characteristics of closures is composition, wherein you can define one closure whose purpose is to combine other closures. Using composition, two or more simple closures can be combined to produce a more elaborate one.

Listing 5 introduces a nifty composition closure. Now hold on tight and read closely: The parameters f and g represent single-parameter closures. So far so good? Now, for some parameter value x, closure g is applied to it and the closure f is applied to the result produced. Whew! By currying the first two closure parameters you can effectively deliver a new closure that is the combination of the effects of these two.

Listing 5 combines closures triple and quadruple to produce the closure twelveTimes. Upon applying this to the actual parameter 3, the value 36 is returned.

Listing 5. Super closure composition, baby!

def multiply = { x, y -> return x * y }
// closure
def triple = multiply.curry(3)
// triple = { y -> return 3 * y }
def quadruple = multiply.curry(4)
// quadruple = { y -> return 4 * y }
def composition = { f, g, x -> return f(g(x)) }
def twelveTimes = composition.curry(triple, quadruple)
def threeDozen = twelveTimes(3)
println "threeDozen: ${threeDozen}"
// threeDozen: 36

Pretty slick, huh?

Five star computations

Now let’s explore some of the spicier aspects of closures. We’ll start with a mechanism by which you can express closures that embody a pattern of computation, which is a concept from functional programming. An example of a pattern of computation is where you transform every element of a List in some way. Because these patterns occur so frequently, we’ve developed a class entitled Functor to encapsulate them as static Closures. Listing 6 shows an extract.

Listing 6. Functor encapsulates a pattern of computation

package fp

abstract class Functor {
  // arithmetic (binary, left commute and right commute)
  public static Closure bMultiply     = { x, y -> return x * y }
public static Closure rMultiply     = { y, x -> return x * y }
public static Closure lMultiply     = { x, y -> return x * y }
  // ...
  // composition
  public static Closure composition   = { f, g, x -> return f(g(x)) }
  // lists
  public static Closure map    = { action, list -> return list.collect(action) }
public static Closure apply  = { action, list -> list.each(action) }


Practically Groovy: Functional programming with curried closures
Here you can see a closure named `map`, which is not to be confused with the `Map` interface. The `map` closure has a parameter `f` that represents a closure and a parameter `list` that represents, not surprisingly, a `List`. It returns a new `List` in which `f` has been mapped to every element in `list`. Of course, Groovy already has the `collect()` method for `Lists`, so we've used that in our implementation.

In Listing 7, we've taken things one step further by currying the `map` closure, resulting in a block that will multiply all the elements of a given list by 12.

### Listing 7. Add some curry and multiply by 12

```groovy
import fp.*

def twelveTimes = { x -> return 12 * x }
def twelveTimesAll = Functor.map.curry(twelveTimes)
def table = twelveTimesAll([1, 2, 3, 4])
println "table: \${table}"
// table: [12, 24, 36, 48]
```

Now that is what we call a five-star computation!

### Business rules a la curry

All this talk of closure cookery is well and good, but the more business-minded among you will better appreciate the next example. Consider the problem of computing the net price of a specific `Book` item, taking into account the shop discount and any governmental taxes such as a `value added tax`. If you were to include this logic as part of the `Book` class, the resulting solution would probably be a hard-wired one. Because the bookshop could change the value of its discount or apply it to only a selection of its stock, such a solution would likely be too rigid.

But guess what? Changing business rules are readily accommodated using curried closures. You can use a set of simple closures to represent individual business rules and you can combine them in various ways using compositions. Finally, you can map them to collections using computation patterns.

Listing 8 illustrates the bookshop example. The closure `rMultiply` is a partial application that adapts the binary multiplication to be a unary closure by using a constant second operand. The two book closures `calcDiscountedPrice` and `calcTax` are instances of the `rMultiply` closure with set values for the multiplier value. The closure `calcNetPrice` is the algorithm to compute the net price by first calculating the discounted price and then the sales tax on top of that. Finally, `calcNetPrice` is applied to the price of the book.
Listing 8. The book business object

import fp.*

class Book {
    String name
    String author
    BigDecimal price
    String category
}

def bk = new Book(name:'Groovy',
    author:'KenB',
    price:25,
    category:'CompSci')

// constants
def discountRate = 0.1
def taxRate = 0.17

// book closures
def calcDiscountedPrice = Functor.rMultiply.curry(1 - discountRate)
def calcTax = Functor.rMultiply.curry(1 + taxRate)
def calcNetPrice =
    Functor.composition.curry(calcTax, calcDiscountedPrice)

// now calculate net prices
def netPrice = calcNetPrice(bk.price)
println "netPrice: ${netPrice}"  // netPrice: 26.325

A groovier visitor

You've seen how you can apply curried closures to functional patterns, so now let's take a look at what happens when we revisit an important object-oriented design pattern using similar techniques. It's a common use case for an object-oriented system to have to traverse a collection of objects and perform some action against each element in the collection. Given a different scenario, the system would traverse the same collection but perform a different action. Usually you can meet this requirement with the Visitor design pattern (see Related topics). A Visitor interface introduces the action protocol for processing an element of the collection. Concrete subclasses define the different required behaviors. A method is then introduced that traverses the collection and applies the Visitor action against each element.

If you haven't guessed it by now, closures can be used to achieve the same effect. One attraction of this approach is that with closures you do not need to develop the visitor class hierarchy. Further, you can effectively use closure composition and mapping to define the action and effect traversal of the collection!

For example, consider a one-to-many relationship between the class Library and the class Book used to represent the inventory held by a library. You could use a List or a Map to implement the relationship; however, a Map offers the advantage that it can provide fast lookup given, say, the book catalog number as the key.

Listing 9 shows a simple one-to-many relationship using a Map. Note the two display methods in the Library class. Both are candidates for refactoring by introducing a visitor.

Listing 9. The library application

class Book {
    String title

String author
String catalogNumber
boolean onLoan = false

String toString() {
    return "Title: ${title}; author: ${author}"
}
}
class Library {
    String name
    Map stock = [:]

def addBook(title, author, catalogNumber) {
    def bk = new Book(title:title,
        author:author,
        catalogNumber:catalogNumber)
    stock[catalogNumber] = bk
}

def lendBook(catalogNumber) {
    stock[catalogNumber].onLoan = true
}

decl displayBooksOnLoan() {
    println "Library: ${name}"
    println "Books on loan" 
    stock.each { entry ->
        if(entry.value.onLoan == true) println entry.value
    }
}

decl displayBooksAvailableForLoan() {
    println "Library: ${name}"
    println "Books available for loan" 
    stock.each { entry ->
        if(entry.value.onLoan == false) println entry.value
    }
}

def lib = new Library(name : 'Napier')
lib.addBook('Groovy', 'KenB', 'CS123')
lib.addBook('Java', 'JohnS', 'CS456')
lib.addBook('UML', 'Ken and John', 'CS789')
lib.lendBook('CS123')
lib.displayBooksOnLoan() // Title: Groovy; author: KenB
lib.displayBooksAvailableForLoan() // Title: UML; author: Ken and John
lib.displayBooksAvailableForLoan() // Title: Java; author: JohnS

Listing 10 includes several closures in the Library class that mimic the use of a visitor. The action closure (somewhat similar to the map closure) applies an action closure to every element of a List. The closure displayLoanedBook displays a book if it is on loan and the closure displayAvailableBook displays a book if it is not on loan. Both act as the visitors and associated actions. Currying the apply closure with displayLoanedBook results in the closure displayLoanedBooks, which is prepared to process the collection of books. A similar scheme is used to produce a display of the books available for lending, as Listing 10 shows.
Listing 10. The library visitor revisited

import fp.*

class Book {
   String title
   String author
   String catalogNumber
   boolean onLoan = false

   String toString() {
      return "Title: ${title}; author: ${author}"
   }
}

class Library {
   String name
   Map stock = [:]

   def addBook(title, author, catalogNumber) {
      def bk = new Book(title:title,
         author:author,
         catalogNumber:catalogNumber)
      stock[catalogNumber] = bk
   }

   def lendBook(catalogNumber) {
      stock[catalogNumber].onLoan = true
   }

   // now uses private displayLoanedBooks() closure
   def displayBooksOnLoan() {
      println "Library: ${name}"
      println "Books on loan"
      displayLoanedBooks(stock.values())
   }

   // now uses private displayAvailableBooks() closure
   def displayBooksAvailableForLoan() {
      println "Library: ${name}"
      println "Books available for loan"
      displayAvailableBooks(stock.values())
   }

   private displayLoanedBook = { bk ->
      if(bk.onLoan == true){
         println bk
      }
   }

   private displayAvailableBook = { bk ->
      if(bk.onLoan == false){
         println bk
      }
   }

   private displayLoanedBooks = Functor.apply.curry(displayLoanedBook)
   private displayAvailableBooks = Functor.apply.curry(displayAvailableBook)
}

def lib = new Library(name : 'Napier')
lib.addBook('Groovy', 'KenB', 'CS123')
lib.addBook('Java', 'JohnS', 'CS456')
lib.addBook('UML', 'Ken and John', 'CS789')
lib.lendBook('CS123')
lib.displayBooksOnLoan()
Testing with closures

Before we wrap up, let's look at one additional use of Groovy closures. Consider an application modeled as a `Company` with many `Employee`s. A recursive relation establishes a further one-to-many aggregation between a single `Employee` (the team leader) and many `Employee`s (the team members). Figure 1 is a class diagram for such an organization.

**Figure 1. The Company application**

You can use closures to make statements about the architectural integrity of your models. For example, in this case, you might want to ensure that every employee has been assigned a manager. The simple closure `hasManager` expresses the requirement for a single employee:

```groovy
def hasManager = { employee -> return (employee.manager != null) }.
```

The partial application of the `forall` closure from the `Functor` class above in Listing 6 can describe the architectural requirement:

```groovy
def everyEmployeeHasManager = Functor.forAll.curry(hasManager).
```

Listing 11 demonstrates the application of curried closures to put the heat on (that is, test) a system's architectural integrity.

**Listing 11. Closures for testing architectural integrity**

```groovy
import fp.*
/**
 * A company with any number of employees.
 * Each employee is responsible to a team leader who, in turn, manages a team of staff.
 */
class Employee {
    int id
    String name
    Map staff = [:]
    def manager = null

    String toString() {
        return "Employee: ${id} ${name}"
    }

    def addToTeam(employee) {
        staff[employee.id] = employee
        employee.manager = this
    }
}
class Company {
    String name
    Map employees = [:]
    def hireEmployee(employee) {
        employees[employee.id] = employee
    }
    def displayStaff() {
        println "Company: $name"
        println "===================="
        employees.each { entry ->
            println "${entry.value}"
        }
    }
}

def co = new Company(name: 'Napier')
def emp1 = new Employee(id: 123, name: 'KenB')
def emp2 = new Employee(id: 456, name: 'JohnS')
def emp3 = new Employee(id: 789, name: 'JonK')
co.hireEmployee(emp1)
co.hireEmployee(emp2)
co.hireEmployee(emp3)
emp3.addToTeam(emp1)
emp3.addToTeam(emp2)
co.displayStaff()
// Architectural closures
def hasManager = { employee -> return (employee.manager != null) }
def everyEmployeeHasManager = Functor.forAll.curry(hasManager)
def staff = new ArrayList(co.employees.values())
println "Every employee has a manager?: ${everyEmployeeHasManager.call(staff)}"

That curry was excellent

You've seen a lot of closures this month, but hopefully with enough of a nouveau twist to keep you hungry for more. As you learned with the multiplication examples, curried closures make it surprisingly easy to implement functional patterns of computation. Once you've got a handle on these patterns, you can deploy them to common enterprise scenarios, such as when we applied them to business rules in the bookstore example. Applying closures to functional patterns is exciting, and once you've done that, it's not too big a stretch to apply them to object-oriented design patterns. Curried closures can be used to mimic the essential elements of the Visitor pattern, as we showed in the Library example. They can also be useful for carrying out integrity checks during software testing, as we showed with the Company example.

All of the examples you've seen this month are common use cases for enterprise systems. It's heartening to see how fluidly Groovy closures and the curry method can be applied to numerous programming scenarios, and to both functional and object-oriented patterns. Haskell Curry would surely find this terribly groovy!
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code</td>
<td>j-pg08235.zip</td>
<td>2KB</td>
</tr>
</tbody>
</table>
Related topics

- "Groovy's growth spurt" (Andrew Glover, developerWorks, July 2005): A look at many of the JSR-compliant changes to Groovy's syntax (including closures).
- "Functional programming in the Java language" (Abhijit Belapurkar, developerWorks, July 2004): A primer on using functional programming constructs such as closures and higher-order functions in the Java language.
- "Beginning Haskell" (David Mertz, developerWorks, September 2001): A tutorial for those who want to learn more about functional programming.
- Practically Groovy: Read the complete set of articles, which build on each other as the series progresses.
- Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, et.al., Addison Wesley, 1995): Learn more about the object-oriented design patterns discussed in this article.