Extracted from:

Programming Groovy 2

Dynamic Productivity for the Java Developer

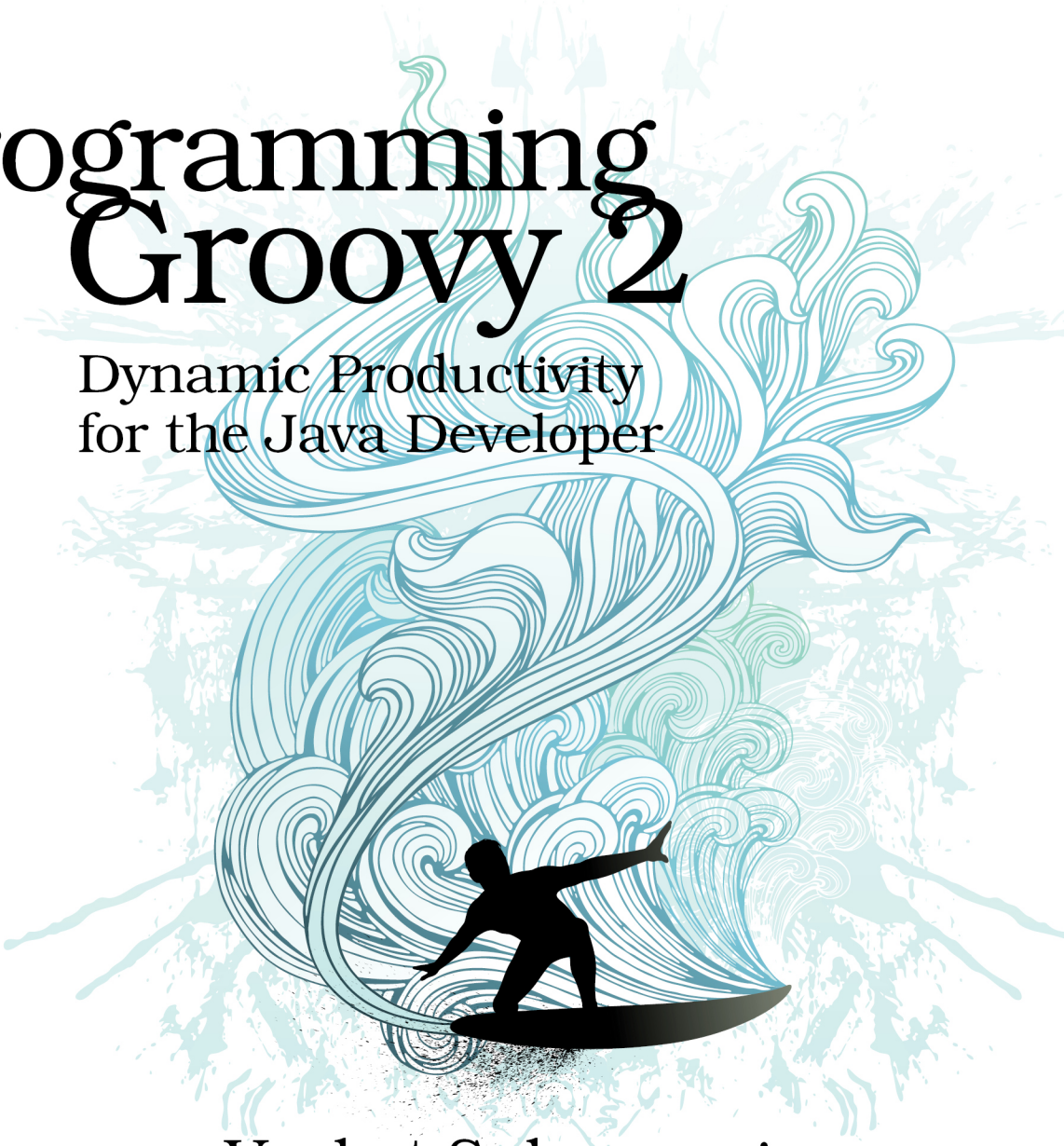
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Groovy 2

Dynamic Productivity for the Java Developer



Venkat Subramaniam

Foreword by Guillaume Laforge

Edited by Brian P. Hogan

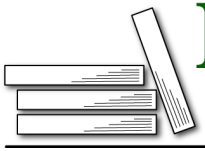
Programming Groovy 2

Dynamic Productivity for the Java Developer

Venkat Subramaniam

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-30-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2013

To Mythili and Balu—for being much more
than an aunt and an uncle—for being there
when I needed them most.

We create anonymous inner classes in Java, where we define method arguments to register event handlers and provide short local glue code. Back when introduced in Java 1.1, anonymous inner classes seemed like a nice idea, but soon we realized that they become verbose, especially for really short implementations of single-method interfaces. Closures in Groovy are short anonymous methods that remove that verbosity.

Closures are lightweight, short, concise, and one of the features we'll employ the most in Groovy. Where we used to pass instances of anonymous classes, now we can pass closures.

Closures are derived from the lambda expressions from functional programming, and "a lambda expression specifies the parameter and the mapping of a function"—see Robert Sebesta's *Concepts of Programming Languages [Seb04]*. Closures are one of the most powerful features in Groovy, yet they are syntactically elegant. Or as the computer scientist and functional-programming pioneer Peter J. Landin put it, "A little bit of syntax sugar helps you to swallow the λ calculus."

We'll use closures extensively through the Groovy JDK (GDK), which has extended the Java Development Kit (JDK) with fluent and convenient methods that take closures. Rather than being forced to create interfaces and a number of small classes, we can design applications with small chunks of low-ceremony code. This means less code, less clutter, and more reuse.

In this chapter you'll learn to create and use closures. We'll cover how to use them to elegantly implement some design patterns. You'll learn that closures don't simply stand in as anonymous methods, but can turn into a versatile tool to solve problems with high memory demands. So let's roll up our sleeves and get down to some code.

4.1 The Convenience of Closures

Closures in Groovy totally remove verbosity in code and help create lightweight reusable pieces of code. To understand the convenience they offer, let's contrast them with familiar traditional solutions for common tasks.

The Traditional Way

Let's consider a simple example—assume we want to find the sum of even values from 1 to a certain number, `n`.

Here is the traditional approach:

UsingClosures/UsingEvenNumbers.groovy

```
def sum(n) {
```

```
  total = 0
  for(int i = 2; i <= n; i += 2) {
    total += i
  }
  total
}
println "Sum of even numbers from 1 to 10 is ${sum(10)}"
```

In the method sum(), we're running a for loop that iterates over even numbers and sums them. Now, suppose instead of that we want to find the product of even numbers from 1 to n.

UsingClosures/UsingEvenNumbers.groovy

```
def product(n) {
  prod = 1
  for(int i = 2; i <= n; i += 2) {
    prod *= i
  }
  prod
}
println "Product of even numbers from 1 to 10 is ${product(10)}"
```

We again iterate over even numbers, this time computing their product. Now, what if we want to get a collection of squares of these values? The code that returns an array of squared values might look like the following:

UsingClosures/UsingEvenNumbers.groovy

```
def sqr(n) {
  squared = []
  for(int i = 2; i <= n; i += 2) {
    squared << i ** 2
  }
  squared
}
println "Squares of even numbers from 1 to 10 is ${sqr(10)}"
```

The code that does the looping is the same (and duplicated) in each of the previous code examples. What's different is the part dealing with the sum, product, or squares. If we want to perform some other operation over the even numbers, we'd be duplicating the code that traverses the numbers. Let's find ways to remove that duplication.

The Groovy Way

Each of the previous three examples produced different results, but all three of them have a common task—picking even numbers from a given collection. Let's start with a function for that common task. Instead of returning a list of even numbers, let's write the function so that when an even number is

picked, the function immediately sends it to a code block for processing. Let the code block simply print that number for now:

UsingClosures/PickEven.groovy
```
def pickEven(n, block) {
  for(int i = 2; i <= n; i += 2) {
    block(i)
  }
}

pickEven(10, { println it } )
```

The pickEven() method is a *higher-order function*—a function that takes functions as arguments or returns a function as a result.[1] The method is iterating over values (like before), but this time it yields, or sends, the value over to a block of code. In Groovy we refer to the anonymous code block as a *closure*—Groovy programmers use a relaxed definition of the term.[2]

The variable block holds a reference to a closure. Much like the way we can pass objects around, we can pass closures around. The variable name does not have to be named block; it can be any legal variable name. When calling the method pickEven(), we can now send a code block as shown in the earlier code. The block of code (the code within {}) is passed for the parameter block, like the value 10 for the variable n. In Groovy, we can pass as many closures as we want. So, the first, third, and last arguments for a method call, for example, may be closures. If a closure is the last argument, there is an elegant syntax, as we see here:

UsingClosures/PickEven.groovy
```
pickEven(10) { println it }
```

If the closure is the last argument to a method call, we can attach the closure to the method call. The code block, in this case, appears like a parasite to the method call. Unlike Java code blocks, Groovy closures can't stand alone; they're either attached to a method or assigned to a variable.

What's that it in the block? If we're passing only one parameter to the code block, then we can refer to it with a special variable name it. We can give an alternate name for that variable if we like, as in the next example:

UsingClosures/PickEven.groovy
```
pickEven(10) { evenNumber -> println evenNumber }
```

1. See http://c2.com/cgi/wiki?HigherOrderFunction.
2. See http://groovy.codehaus.org/Closures+-+Formal+Definition.

The variable evenNumber now refers to the argument that's passed to this closure from within the pickEven() method.

Now let's revisit the computations on even numbers. We can use pickEven() to compute the sum, like so:

UsingClosures/PickEven.groovy
```
total = 0
pickEven(10) { total += it }
println "Sum of even numbers from 1 to 10 is ${total}"
```

We started out simply printing the even numbers generated by pickEven(), but now we're totaling those values, without any change to the function. Rather than duplicating the code, as in the traditional-way examples, we have concise code with greater reuse. The function is not limited to totaling the values; we can use it, for example, to compute the product, as in the next code:

UsingClosures/PickEven.groovy
```
product = 1
pickEven(10) { product *= it }
println "Product of even numbers from 1 to 10 is ${product}"
```

Other than the syntactic elegance, closures provide a simple and easy way for a function to delegate part of its implementation logic.

The block of code in the previous example does something more than the block of code we saw earlier. It stretches its hands and reaches out to the variable product in the scope of the caller of pickEven(). This is an interesting characteristic of closures. A closure is a function with variables bound to a context or environment in which it executes.

We know how to create closures; next let's discuss how to use them in applications.

4.2 Programming with Closures

We're talking about the power and elegance of closures in this chapter, but first let's discuss how to approach them in our projects. We need to decide whether we want to implement a certain functionality or task as a regular function/method or whether we should use a closure.

Closures augment, refine, or enhance another piece of code. For example, a closure may be useful to express a predicate or condition that will refine the selection of objects. We can use closures to take advantage of coroutines such as the control flow like in iterators or loops.

Closures are very helpful in two specific areas. They can help manage resource cleanup (see Section 4.5, *Using Closures for Resource Cleanup*, on page ?)

and they can help create internal domain-specific languages (DSLs)—see Chapter 19, *Creating DSLs in Groovy*, on page ?.

To implement a certain well-identified task, a regular function is better than a closure. A good time to introduce closures is during refactoring.

Once we get the code working, we can revisit it to see whether closures would make it better and more elegant. Let a closure emerge from this effort rather than forcing a use to begin with.

We should keep closures small and cohesive. These are intended to be small chunks of code, only a few lines, that are attached to method calls. When writing a method that uses a closure, it's better not to overuse dynamic properties of closures, like determining the number and types of parameters at runtime. It must be very simple and obvious to implement a closure when calling methods.

We saw the convenience and benefits of using closures. Next, let's look at a couple of different ways to use closures.