Community Experience Distilled

Extend and enhance your Java applications with domain-specific scripting in Groovy

Groovy for Domain-specific Languages
Second Edition

A comprehensive tutorial on designing and developing mini Groovy-based domain-specific languages (DSLs), this book will guide you through the development of several mini DSLs that will help you gain all the skills needed to develop your own Groovy-based DSLs with confidence and ease.

Starting with the bare basics, this book will focus on how Groovy can be used to construct mini DSLs, and will go through the more complex metaprogramming features of Groovy, including using the abstract syntax tree (AST). Practical examples are used throughout this book to de-mystify these seemingly complex language features and to show how they can be used to create simple and elegant DSLs. Packed with examples, including several fully functional DSLs, this book will serve as a springboard for developing your own DSLs.

Who this book is written for
This book is for Java software developers who have an interest in building domain scripting into their Java applications. No knowledge of Groovy is required, although it will be helpful. This book does not teach Groovy, but quickly introduces the basic ideas of Groovy.

What you will learn from this book

- Familiarize yourself with Groovy scripting and work with Groovy closures
- Use the metaprogramming features in Groovy to build mini languages
- Employ Groovy markup and builders to simplify application development
- Familiarize yourself with Groovy markup and build your own Groovy builders
- Build effective DSLs with operator overloading, command chains, builders, and a host of other Groovy language features
- Integrate Groovy into your Java and JVM based applications

Who this book is written for
This book is for Java software developers who have an interest in building domain scripting into their Java applications. No knowledge of Groovy is required, although it will be helpful. This book does not teach Groovy, but quickly introduces the basic ideas of Groovy.

Prices do not include local sales tax or VAT

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction to DSLs and Groovy'
- A synopsis of the book’s content
- More information on Groovy for Domain-specific Languages Second Edition
Fergal Dearle is a seasoned software development professional with almost 30 years' experience in software product development across a wide variety of technologies. He is currently the principal consultant with his own software development consulting company, Dearle Technologies Ltd., engaged in design, development, and architecture of new software products for client companies. Recent projects have included the integration of the Telegraph (http://www.telegraph.co.uk) into Apple's new Apple News application for iOS 9 and the reengineering of the G-Cloud Digital Marketplace for the United Kingdom Cabinet Office (https://www.digitalmarketplace.service.gov.uk).

He is a committed mentor in his local CoderDojo in Wexford Town where he teaches Groovy to the young coding ninjas. He has been recently nominated as a CoderDojo Hero for his work.

In the past, Fergal has worked in lead architect and developer roles for Candle Corporation on the OMEGAMON product, which is now part of IBM's Tivoli product suite, and as the development manager for Unix implementations of Lotus 1-2-3. In the early 1990s, Fergal led the team at Glockenspiel that developed CommonView, the first object-oriented UI framework for Microsoft Windows. The team was awarded one of the first ever Jolt Productivity Awards by Dr. Dobbs Journal.
Preface

The Java virtual machine runs on everything from the largest mainframe to the smallest microchip and supports every conceivable application. But Java is a complex, and sometimes arcane, language to develop with. Groovy allows us to build targeted single-purpose mini languages, which can run directly on the JVM along with the regular Java code.

This book provides a comprehensive tutorial on designing and developing mini Groovy-based domain-specific languages (DSLs). It is a complete guide to the development of several mini DSLs with a lot of easy-to-understand examples. This book will help you gain all of the skills needed to develop your own Groovy-based DSLs.

*Groovy for Domain-specific Languages, Second Edition,* guides you from the basics through to the more complex metaprogramming features of Groovy. The focus is on how the Groovy language can be used to construct domain-specific mini languages.

Practical examples are used throughout to demystify these seemingly complex language features and to show how they can be used to create simple and elegant DSLs. The examples include a quick and simple Groovy DSL to interface with Twitter.

The book concludes with a chapter focusing on integrating a Groovy-based DSL in such a way as the scripts can be readily incorporated into your own Java applications. The overall goal of this book is to take developers through the skills and knowledge they need to start building effective Groovy-based DSLs to integrate into their own applications.
What this book covers

Chapter 1, Introduction to DSLs and Groovy, discusses how DSLs can be used in place of general-purpose languages to represent different parts of a system. You will see how adding DSLs to your applications can open up the development process to other stakeholders in the development process. You'll also see how, in extreme cases, the stakeholders themselves can even become co-developers of the system by using DSLs that let them represent their domain expertise in the code.

Chapter 2, Groovy Quick Start, covers the basics of installing Groovy and running simple Groovy scripts.

Chapter 3, Essential Groovy DSLs, covers two essential Groovy-based tools, Gradle and Spock. Gradle is a build, test, and deployment automation tool, which is powered by a Groovy DSL. Spock is a unit testing and specification framework built over JUnit. Both tools are used extensively throughout the book.

Chapter 4, The Groovy Language, covers a whistle-stop tour of the Groovy language. It also touches on most of the significant features of the language as a part of this tour.

Chapter 5, Groovy Closures, covers closures in some depth. It covers all of the important aspects of working with closures. You can explore the various ways to call a closure and the means of passing parameters. You will see how to pass closures as parameters to methods, and how this construct can allow the adding of mini DSL syntax to our code.

Chapter 6, Example DSL – GeeTwitter, focuses on how we can start with an existing Java-based API and evolve it into a simple user-friendly DSL that can be used by almost anybody. You'll learn the importance of removing boilerplate code and how you can structure our DSL in such a way that the boilerplate is invisible to our DSL users.

Chapter 7, Power Groovy DSL Features, covers all of the important features of the Groovy language, and looks in depth at how some of these features can be applied to developing DSLs.

Chapter 8, AST Transformations, covers how to use the Groovy abstract syntax tree (AST) transformations. AST transformations are a mechanism for us to hook into the Groovy compilation process. Here we look at compile time metaprogramming and see how we can use AST transformations to build code on the fly during the compilation process.

Chapter 9, Existing Groovy DSLs, discusses some existing Groovy DSLs that are in current use and are free to download.
Chapter 10, Building a Builder, explains how Groovy provides two useful support classes that make it much simpler to implement our own builders than if we used the MOP. You'll see how to use BuilderSupport and FactoryBuilderSupport to create our own builder classes.

Chapter 11, Implementing a Rules DSL, takes a look at Groovy bindings to see how they can be used in our DSL scripts. By placing closures strategically in the binding, you can emulate named blocks of code. You can also provide built-in methods and other shorthand by including closures and named Boolean values in the binding. These techniques can be used to great effect to write DSL scripts that can be read and understood by stakeholders outside of the programming audience.

Chapter 12, Integrating It All, takes all the knowledge from the previous chapters and builds a fully functioning web application based on a simple Game Engine DSL for Tic Tac Toe.
Introduction to DSLs and Groovy

It has been over 10 years since my first contact with the Groovy language. The occasion was an introductory talk about Groovy at JavaOne in the Moscone Centre, San Francisco, by James Strachan, the creator of the Groovy language. Java itself was just 10 years old at that time and Groovy was one of the very first languages other than Java to run on the Java Virtual Machine (JVM).

Just this year, Java celebrated its twentieth birthday. In 2005, there were an estimated 3 million Java developers worldwide. Now, in 2015, Wikipedia estimates it as 11 million. The Groovy language has also taken off. There were an estimated 5 million downloads of Groovy in the last year alone. So what are the benefits of Groovy and why should you consider using it?

The Groovy project site at http://www.groovy-lang.org explains this better than I ever could with six major benefits:

- A flat learning curve
- Powerful features
- Smooth Java integration
- Domain-specific languages
- A vibrant and rich ecosystem
- The scripting and testing glue

In this book, we will cover all the key benefits of the Groovy language. The main focus, however, is on how Groovy supports the development of domain-specific languages through its metaprogramming features.
One of the big benefits of Groovy is how its dynamic features support the development of domain-specific languages (DSLs) or "mini languages", which we can run directly on the JVM alongside your existing Java code. Groovy DSLs integrate seamlessly into the Groovy language itself in such a way that it's not always apparent where the regular Groovy code stops and the DSL starts.

In fact, large parts of almost any Groovy application are written using Groovy-based DSLs. For instance, a new developer starting out with Groovy might assume that the builder code he uses to output some XML is a part of the core Groovy language. But it is, in fact, a mini internal DSL implemented using the Groovy metaprogramming features.

If you are an Android developer, the chances are you may have programmed in Groovy already. Since 2013, the build system in the Android SDK has been a tool called Gradle (http://www.gradle.org). Gradle is a Groovy-based DSL for dependency management and build automation.

Whether you are one of the 11 million existing Java developers, looking to add DSL features to your application, or you are an existing Groovy developer looking to improve your knowledge of DSL writing, metaprogramming or AST transformations, this book is intended for you.

By the end of this book, I hope that you will have the knowledge and the confidence to start building your own DSLs with Groovy, and be able to integrate them into your Java applications. To begin with, in this chapter, we will take some baby steps. This chapter will give you a brief background on DSLs and their usage. We will also dip a toe into the Groovy language, and briefly touch on the features of the language that distinguish it from Java and make it a great tool for developing DSLs on top of the Java platform.

**DSL – a new name for an old idea**

I've mentioned domain-specific language (DSL) several times now, so what does this really mean? The term "DSL" describes a programming language that is dedicated to a specific problem domain. The idea is not new. DSLs have been around for a long time. One of the most exciting features of Unix has always been its mini languages. These include a rich set of typesetting languages (troff, eqn, pic, and so on), shell tools (awk, sed, and so on), and software development tools (make, yacc, and lex).
The Java platform has a multitude of mini DSLs in the form of XML config files for configuration of everything from EJBs to web applications. In many JEE applications, Enterprise Java Beans (EJB) can be configured using an XML configuration file, ejb-jar.xml. While the ejb-jar.xml file is written in the general-purpose language XML, the contents of the file need to conform to a document type definition (DTD) or XML schema, which describes the valid structure of the file.

XML configuration files can be found across a wide range of libraries and frameworks. Spring is configured by using a spring-config.xml file, and Struts with struts-config.xml. In each case, the DTD or schema defines the elements and tags, which are valid for the specific domain, be it EJB, Spring, or Struts. So, ejb-jar.xml can be considered a mini DSL for configuring EJB, spring-config.xml is a mini DSL for configuring Spring beans, and so on.

In essence, DSL is a fancy name for something that we use every day of our professional programming lives. There are not many applications that can be fully written in a single general-purpose language. As such, we are the everyday consumers of many different DSLs, each of which is specific to a particular purpose.

A typical day's work could involve working with Java code for program logic, CSS for styling a web page, JavaScript for providing some dynamic web content, and Ant, Maven, or Gradle to build the scripts that tie it all together. We are well used to consuming DSLs, but seldom consider producing new DSLs to implement our applications—which we should.

The evolution of programming languages

My own background is probably typical of many of my generation of old-school programmers. Back in 1986, I was a young software engineer fresh out of college. During my school and college years, I studied many different programming languages. I was fortunate in high school to have had a visionary Math teacher who taught us to program in BASIC, so I cut my teeth programming as early as 1974. Through various college courses, I came to know about Pascal, C, Fortran, Lisp, Assembler, and COBOL.

My school, college, and early professional career all reinforced a belief that programming languages were for the exclusive use of us programmers. We liked nothing better than spending hours locked away in dark rooms writing reams of arcane and impenetrable code. The more arcane and impenetrable the better! The hacker spirit prevailed, and annual competitions such as the International Obfuscated C Code Contest (IOCCC) were born.
The IOCCC runs to this day. The point of the contest is to write valid but impenetrable C code that works. Check out http://www.ioccc.org to see how not to write code.

General-purpose languages

All of the teaching in college in those days revolved around the general-purpose languages. I recall sitting in class and being taught about the "two" types of programming language: machine language, and high-level languages. Both were types of general-purpose languages, in which you could build any type of application, but each language had its own strengths and weaknesses. The notion of a DSL was not yet considered as part of the teaching program. Nor was the idea that anyone other than a cadre of trained professional programmers (hackers) would ever write programs for computers. These days, the word "hacker" has bad connotations of being synonymous with virus writers and the likes. In those days, a good "hack" was an elegant programming solution to a hard problem and being called a hacker by one’s peers was a badge of pride for most programmers.

The high-level programming language you used defined what type of an application programmer you were. COBOL was for business application programming, Fortran was for scientific programmers, and C was for hackers building Unix and PC software. Although COBOL and Fortran were designed to be used in a particular business domain, they were still considered general-purpose languages. You could still write a scientific application in COBOL or a business application in Fortran if you wanted to. However, you were unlikely to try any low-level device driver development in COBOL.

Although it was possible to build entire applications in assembly language (and many people did), high-level languages, such as C, BASIC, and COBOL, were much better suited to this task. The first version of the world-beating spreadsheet Lotus 1-2-3 was written entirely in 8086 assembly language, and ironically, it was the rewrite of this into the supposed high-level language C that nearly broke the company in the late 1980’s.

Languages such as C and C++ provide the low-level functionality in a high-level language, which enabled them to be used across a much greater range of domains, including those where assembly was utilized before. These days, Java, C# and C++ compete with each other like the Swiss Army knives of general-purpose languages. There are almost no application domains to which these languages have not been applied, from space exploration, through to enterprise business systems, and mobile phones.
Spreadsheets and 4GLs

Programs such as Lotus 1-2-3 and its precursor VisiCalc revolutionized people's view of who would program computers. A whole generation of accountants, financial analysts, scientists, and engineers came to realize that they can develop sophisticated turnkey solutions for themselves, armed only with a spreadsheet and a little knowledge of macros. Spreadsheet macros are probably one of the first DSLs to find their way out of the cloisters of the IT community and into the hands of the general business user.

Around this time, there was also much media attention paid to the new 4GL (fourth-generation language) systems. 4GLs were touted as being hugely more efficient for developing applications than traditional high-level languages, which then became known as third-generation language (3GL). From the hype in the media at the time, you would be forgiven for thinking that the age of the professional programmer was coming to an end and that an ordinary business user could use a 4GL to develop their own business applications. I viewed this claim with a degree of healthy skepticism—how could a non-programmer build software?

Like DSLs, 4GLs were, generally speaking, targeted at particular problem spaces, and tended to excel at providing solutions in those narrow target markets. The sophistication of most applications in those days was such that it was possible to build them with a few obvious constructs. 4GLs tended to be turnkey environments with integrated tools and runtime environments. You were restricted by the environment that the 4GL provided, but the applications that could be built with a 4GL could be built rapidly, and with a minimal amount of coding.

4GLs differ from our modern understanding of a DSL. We generally think of a DSL as being a mini language with a particular purpose, and they do not generally impose an entire runtime or tool set on their use. The best DSLs can be mixed and matched together, and used in conjunction with a general-purpose programming language, such as C++ or Java, to build our applications.

Language-oriented programming

Martin Fowler has spoken about the use of many mini DSLs in application development. He advocates building applications out of many mini DSLs, which are specific to the particular problem space, in a style of development called language-oriented programming. In a way, this style of programming is the norm for most developers these days, when we mix and match HTML, CSS, SQL, and Java together to build our applications.
Introduction to DSLs and Groovy

The thrust of language-oriented programming is that we should all be going beyond exploiting these generally available languages and implementing our own DSLs that represent the particular problem space that we are working on. With a language-oriented programming approach, we should be building DSLs that are as narrowly focused as the single application that we are currently working on. A DSL does not need to be generally applicable to be useful to us.

Who are DSLs for?

It's worth considering for a moment who the different types of users of a DSL might be. Most DSLs require some programming skills in order to get to grips with them, and are used by software and IT professionals in their daily chores, building, and maintaining and managing systems. They are specific to a particular technical aspect of system development. So the domain of CSS as a DSL is web development in general, and specifically page styling and layout. Many web developers start from a graphic design background and become proficient as coders of HTML, CSS, and JavaScript simply because it gives them better fine-grained control of the design process.

Many graphic designers, for this reason, eventually find themselves eschewing graphical tools such as Dreamweaver in favor of code. Hopefully, our goal in life will not be to turn everybody into a coder. Whereas most DSLs will remain in the realm of the programmer, there are many cases where a well-designed DSL can be used by other stakeholders in the development process other than professional developers. In some cases, DSLs can enable stakeholders to originate parts of the system by enabling them to write the code themselves. In other cases, the DSL can become a shared representation of the system. If the purpose of a particular DSL is to implement business rules then, ideally, that DSL should express the business rule in such a way that it can be clearly understood upon reading by both the business stakeholder who specified it and the programmer who wrote it.

A DSL for process engineers

My own introduction to the concept of DSLs came about in 1986 when I joined Computer Products Inc. (CPI) as a software engineer. In this case, the DSL in question was sophisticated enough to enable the stakeholders to develop large parts of a running system.
CPI developed a process control system, which was primarily sold to chemical and pharmaceutical industries. It was a genuinely distributed system when most process control systems were based on centralized mini or mainframe computers. It had its own real-time kernel, graphics, and a multitude of device drivers for all types of control and measurement devices. But the most innovative part of the system, which excited customers, was a scripting language called **EXTended Operations Language (EXTOL)**. EXTOL was a DSL in the purest sense because it drew the domain experts right into the development process, as originators of the running code.

With EXTOL, a chemical process engineer or chemist could write simple scripts to define the logic for controlling their plant. Each control block and measurement block in the system was addressable from EXTOL. Using EXTOL, a process engineer could write control logic in the same pseudo English that they used to describe the logic to their peers.

The following script could be deployed on a reactor vessel to control the act of half-filling the vessel with the reactant from VALVE001:

```
  drive VALVE001 to OPEN
  when LEVELSENSOR.level >= 50%
  drive VALVE001 to CLOSED
```

This was an incredibly powerful concept. Up to this point, most process control systems were programmed in a combination of high-level languages on the main process system, and relay logic on PLCs in the plant. Both tasks required specific programming skills, and could not generally be completed by the chemists or chemical engineers, who designed the high-level chemical processing undertaken at the plant. I recall a room full of white-coated chemists at one plant happily writing EXTOL scripts, as we commissioned the plant.

The proof of the pudding is always in the eating, and I don't recall a CPI engineer ever being called upon to write a single line of EXTOL code on behalf of a customer. Given an appropriate DSL that fit their needs, our customers could write all of the code that they need themselves, without having to be programmers.

This shows the power of DSLs at their best. At this extreme end of the spectrum, a DSL becomes a programming tool that a domain expert can use independently, and without recourse to the professional programmer. It's important to remember, however, that the domain experts in this case were mostly process engineers. Process engineers are already well accustomed to devising stepwise instructions, and building process flows. They will often use the same visual representations as a programmer, such as a flow chart to express a process that they are working on.
Introduction to DSLs and Groovy

When devising a DSL for a particular domain, we should always consider the stakeholders who need to be involved in using it. In the case of EXTOL, the DSL was targeted at a technical audience who could take the DSL and become part of the system development process. Not all of our stakeholders will be quite as technical as this. But, at the very least, the goal when designing a DSL should be to make the DSL understandable to nontechnical stakeholders.

Stakeholder participation

It's an unfortunate fact that with many DSLs, especially those based on XML, the code that represents a particular domain problem is often only legible to the programming staff. This leads to a disconnect between what the business analysts and domain experts define, and what eventually gets implemented in the system. For instance, a business rule is most likely to be described in plain English by a business analyst in a functional specification document. But these rules will most likely be translated by developers into an XML representation that is specific to the particular rules engine, which is then deployed as a part of the application. If the business analyst can't read the XML representation and understand it, then the original intent of the rule can easily be lost in translation.

With language-oriented programming, we should aim to build DSLs that can be read and understood by all stakeholders. As such, these DSLs should become the shared living specification of the system, even if in the end they must, by necessity, be written by a programmer with a technical understanding of the DSL.

DSL design and implementation

DSLs can take many different forms. Some DSLs, such as Unix mini languages, (sed, awk, and troff) have a syntactical structure, which is unique to that particular language. To implement such DSLs, we need to be able to parse this syntax out of the text files that contain the source code of that particular language. To implement our own DSL in this style involves implementing a mini compiler that uses lexing and parsing tools such as lex, yacc, or antlr.

Compiler writing is one particular skill that is outside the skill set of most application development teams. Writing your own parser or compiler grammar is a significant amount of effort to go to, unless the DSL is going to be used generally, and is beyond the scope of most application-specific DSLs.
EXTOL circumvented this problem by having its own syntax-sensitive editor. Users edited their EXTOL scripts from within the editor, and were prompted for the language constructs that they needed to use for each circumstance. This ensured that the scripts were always well-formed and syntactically correct. This also meant that the editor can save the scripts in an intermediate p-code form so that the scripts never existed as text-based program files, and therefore never needed to be compiled.

Many of the DSLs that we use are embedded within other languages. The multitude of XML configuration scripts in the Java platform are an example of this. These mini DSLs piggyback on the XML syntax, and can optionally use an XML DTD or schema definition to define their own particular syntax. These XML-based DSLs can be easily validated for "well-formedness" by using the DTD or schema.

**External versus internal DSLs**

We generally refer to DSLs that are implemented with their own unique syntax as external DSLs, and those that are implemented within the syntax of a host language as embedded or internal DSLs. Ideally, whenever building a new DSL, it would be best to give it its own unique and individual syntax. By designing our own unique syntax, we can provide language constructs, which are designed with both the problem domain and the target audience in mind.

If the intended user of the DSL is a non-programmer, then developing an XML-based syntax can be problematic. XML has its own particular rules about opening, closing, and properly terminating tags that appear arcane to anybody except a programmer. This is a natural constraint when working with DSLs that are embedded/internal to another language. An XML-based DSL cannot help being similar to XML.

Embedded/internal DSLs will never be as free-form as a custom external DSL due to the constraints of the host language. Fortunately, Groovy-based DSLs are capable of being structured in a more human-readable format. However, they always need to use well-formed Groovy syntax, and there are always going to be compromises when designing Groovy-based DSLs that are readable by your target audience.
Operator overloading

Some general-purpose languages, such as C++, Lisp, and now Groovy, have language features that assist in the development of mini language syntaxes. C++ was one of the earliest languages to implement the concept of operator overloading. By using operator overloading, we can make non-numeric objects behave like numeric values by implementing the appropriate operators. So, we can add a plus operator to a String object in order to support concatenation. When we implement a class that represents a numeric type, we can add the numeric operators again to make them behave like numeric primitives. We can implement a ComplexNumber class, which represents complex numbers, as follows:

```cpp
class ComplexNumber {
  public:
    double real, imag;
    ComplexNumber() { real = imag = 0; }
    ComplexNumber(double r, double i) { real = r; imag = i; }
    ComplexNumber& operator+(const ComplexNumber& num);
};
```

To add one complex number to another, we need to correctly add each of the real and imaginary parts together to generate the result. We implement an equality operator for ComplexNumber as follows:

```cpp
ComplexNumber& ComplexNumber::operator=(const ComplexNumber& num) {
  real = num.real;
  imag = num.imag;
  return *this;
}
```

This allows us then to add ComplexNumber objects together as if they were simple numeric values:

```cpp
int main(int argc, const char* argv[]) {
  ComplexNumber a(1, 2), b(3, 4);
  ComplexNumber sum;
  sum = a + b;
  cout << "sum is " << sum.real << " ; "
       << sum.imaginary << "i" << endl;
}
```

One of the criticisms of the operator overload feature in C++ is that when using operator overloading, there is no way to control what functionality is being implemented in the overloaded function. It is perfectly possible—but not very sensible—to make the + operator subtract values and the – operator add values. Misused operator overloading has the effect of obfuscating the code rather than simplifying it. However, sometimes this very obfuscation can be used to good effect.
The preceding example illustrates what could be considered as a classic case of obfuscation in C++. If your use of C++ predated the introduction of the standard C++ libraries and the streams libraries in particular, you will probably do a double take when looking at this code.

The example uses what has become commonly known as the stream operator $\ll$. This operator can be used to send a character stream to standard output, the logic being that it looks very much like how we stream output from one program to another in a Unix shell script. In fact, there really is no such thing as a stream operator in C++ and what has been overloaded here is the binary left shift operator $\ll$. I have to admit that my first encounter with a code like this left me perplexed. Why would anybody want to left shift the address of a string into another object was beyond me? Common use over the intervening years means that this is now a perfectly natural coding style for all C++ programmers. In effect, the streaming operator implements a mini internal DSL for representing streaming. It subverts the original language a little by using an operator out of context, but the end effect is perfectly understandable and makes sense.

During a fireside chat event at JavaOne some years ago, James Gosling was asked if he would ever consider operator overloading for the Java language, and the answer was a resolute no! Fortunately, we don't have to wait and see if Oracle will ever add operator overloading to Java. With Groovy, we can have it now. Groovy has an extensive set of features, including operator overloading that allow us to implement feature-rich DSLs from within the language. We'll take a look at some of those features that distinguish it from Java, now.

**Groovy**

In the later chapters of this book, we will discuss the Groovy language in detail, but let's begin with a brief introduction to the language and some of the features that make it a useful addition to the Java platform.

The Java platform has expanded over the years to cover almost all conceivable application niches—from Enterprise applications, to mobile and embedded applications. The core strengths of Java are its rich set of APIs across all of these problem domains and its standardized **virtual machine (VM)** interface. The standard VM interface has meant that the promise of "write once, run anywhere" has become a reality. The JVM has been implemented on every hardware architecture and operating system from the mightiest mainframe down to the humble Lego Mindstorms robotic kits for kids.
On top of this standard VM, the list of APIs that have been built extends into every conceivable domain. In addition to the standard APIs that are a part of JME, JSE, and JEE, which are extensive in themselves, there are literally thousands of open source component libraries and tools to choose from. All of this makes for a compelling argument for using Java for almost any software project that you can think of.

For many years of its evolution, the JVM was considered to be just that—a virtual machine for running Java programs. The JVM spec was designed originally by James Gosling to be used exclusively for the Java language. In recent years, there have been a number of open source projects that have started to introduce new languages on top of the JVM, such as JRuby (an implementation of the Ruby language), Jython (an implementation of the Python language and Groovy), Clojure, and Scala.

A natural fit with the JVM

Groovy differs from the preceding languages, as the Groovy language was designed specifically to be a new language to run on the JVM. Groovy is designed to be source compatible with the Java language as well as being binary-compatible at the byte code level.

James Strachan and Bob McWhirter started the Groovy project in August 2003 with the goal of providing a new dynamic and object-oriented language, which can run on the JVM. It took several existing dynamic languages, such as Ruby, Python, Dylan, and Smalltalk, as its inspiration. James had looked at the Python scripting language and had been impressed with the power that it had over Java. James and Bob wanted to design a language that had the powerful scripting features of Python, but stayed as close to the Java language as possible in terms of its syntax.

Groovy is code compatible with Java, and for this reason, it is possible in most cases to take an existing .java file and rename it to .groovy and it will continue to work. Groovy has its own compiler, groovyc, which generates Java byte code from Groovy source files just as the javac compiler does. Groovyc generates class files, which run directly on the JVM. Methods defined in a Groovy class can be called directly from Java and vice versa.

Groovy classes and interfaces are 100 percent binary compatible with their Java counterparts. Uniquely, this means that we can create a new Groovy class that extends a Java class or implements a Java interface. You can also create Java classes that extend Groovy classes or implement Groovy interfaces.
Groovy language features
Groovy adds a number of unique features that distinguish it from Java and allow developers to code at a higher level, and use a more abstract idiom, than is possible with Java. Placing all of these powerful features on top of a language that is code and API compatible with the Java platform is a powerful proposition.

Static and optional typing
In Java, as in other statically-typed languages, variables must first be declared with a type before they can have a value assigned to them. In Groovy, type can be left to be determined at the time of assignment. Groovy supports both static and optional typing as follows:

```groovy
String str1 = "I'm a String"
def str2 = "I'm also a String"
```

Both variables `str1` and `str2` are of the type `String`. The late binding of the type in the Groovy-style assignment allows for a much less verbose code.

Native support for lists and maps
One of the great bugbears of the Java language is the cumbersome interfaces required for list and map manipulation. Groovy adds native support for all of the Java collection types through a very intuitive and readable syntax. The following code:

```groovy
def authors = [ 'Shakespeare', 'Beckett', 'Joyce', 'Poe' ]
println authors
println authors[2]
```

Produces this output:

```plaintext
[Shakespeare, Beckett, Joyce, Poe]
Joyce
```
Maps are also declared with ease:

```groovy
println book
println book['title']
println book.title
```
This produces the following output:

```
def biggest = { number1, number2 ->
    number1<number2?number2:number1
}
// We can invoke the call method of the Closure class
def result = biggest.call(7, 1)
println result
// We can use the closure reference as if it were a method
result = biggest(3, 5)
println result
// And with optional parenthesis
result = biggest 13, 1
println result
```

Closures can contain multiple statements and can therefore be as complex as you like. In the following example, we iterate through a list looking for the biggest number, and return it when we are done:

```
def listBiggest = { list ->
    def biggest = list[0]
    for( i in list)
        if( i > biggest)
            biggest = i
    return biggest
}
def numberList = [ 8, 6, 7, 5, 3, 9]
println listBiggest( numberList)
```
Groovy operator overloading

Operator overloading is a powerful feature of the C++ language. Java inherited many of the features of the C++ language, but operator overloading was significantly left out. Groovy introduces operator overloading as a base language feature.

Any Groovy class can implement a full set of operators by implementing the appropriate corresponding method in the class. For example, the plus operator is implemented via the `plus()` method.

Regular expression support

Groovy builds regular expression handling right into the language via the `=~` operator and matcher objects. The following example creates a regular expression to match all multiple occurrences of the space character. This creates a matcher object from this expression and applies it to a string by using the `replaceAll` method:

```groovy
def lorem = "Lorem ipsum dolor sit amet, consectetur adipiscing elit"
println lorem
def matcher = lorem =~ " +"
def removed = matcher.replaceAll(" ")
println removed
```

Optional syntax

Optional typing means that variable type annotations are optional. This does not mean that variables have an unknown variable type. It means that the type will be determined at run time based on the value that gets assigned to the variable. All of the following are legal syntax in Groovy:

```groovy
int a = 3
def b = 2
String t = "hello"
def s = 'there'
```

Trailing semicolons at the end of statements are optional. The only time that you explicitly need to use a semicolon in Groovy is to separate statements that occur on the same line of code, as shown in the first and third lines in the following code:

```groovy
int a = 3; int b = 4;
def c = 2
def d = 5; def e = 6
```
Method call parentheses are also optional when the method being invoked has passed some parameters. We saw earlier, with closures, that we can invoke a closure through its reference as if it were a method call. When invoking a closure in this way, we can also drop the parentheses when passing parameters, as shown in the following code:

```groovy
println(a);
c = 2
print c
printit = { println it }
printit c
```

These make for a much looser programming style, which is closer to the scripting syntax of Ruby or Python. This is a big benefit when we are using Groovy to build DSLs. When our target audience is nontechnical, being able to drop parentheses and semicolons will make our code much more legible. Consider the following example, where we have two methods, or closures, to get an account by ID and then credit the account with some funds:

```groovy
Account account = getAccountById(234);
creditAccount(account, 100.00);
```

With optional types, such as parentheses and semicolons, this can be used to write code that is far more legible to our target audience:

```groovy
account = getAccountById 234
creditAccount account, 100.00
```

### Groovy markup

There are a number of builder classes built in Groovy. There are markup builders for HTML, XML, Ant build scripts, and for Swing GUI building. Markup builders allow us to write code to build a tree-based structure directly within our Groovy code. Unlike API-based approaches for building structures, the tree-like structure of the resulting output is immediately obvious from the structure of our Groovy markup code. Consider the following XML structure:

```xml
<?xml version="1.0"?>
<book>
  <author>Fergal Dearle</author>
  <title>Groovy for DSL</title>
</book>
```
In Groovy markup, this XML can be generated simply with the following code fragment:

```groovy
def builder = new groovy.xml.MarkupBuilder()
def closure = {
    author 'Fergal Dearle'
    title 'Groovy for DSL'
}
def closure // pass a closure to book method
builder.book(closure) // which can be written without parentheses
builder.book closure // or just inline the closure as a parameter
builder.book {
    ...
}
```

At first glance, this looks like strange special case syntax for markup. It's not! The structure of this code can be explained through the use of closures and the optional syntax that we've discussed in this chapter. We will go into this in great detail in Chapter 5, Groovy Closures, but it is interesting at this point to see how the clever use of some language features can yield a powerful DSL-like markup syntax.

Breaking down the preceding code a little, we can rewrite it as:

```groovy
def closure = {
    author 'Fergal Dearle'
    title 'Groovy for DSL'
}
// pass a closure to book method
builder.book(closure)
// which can be written without parentheses
builder.book closure // or just inline the closure as a parameter
builder.book {
    ...
}
```

In other words, the code between the curly braces is in fact a closure, which is passed to the book method of MarkupBuilder. Parentheses being optional, we can simply declare the closure inline after the method name, which gives the neat effect of seeming to mirror the markup structure that we expect in the output.

Similarly, author and title are just method invocations on MarkupBuilder with the optional parentheses missing. Extending this paradigm a little further, we can decide to have author take a closure parameter as well:

```groovy
def closure = {
    author {
        author 'Fergal Dearle'
    }
    title 'Groovy for DSL'
}
def closure
```

---

Chapter 1
Introduction to DSLs and Groovy

```java
    first_name 'Fergal'
surname 'Dearle'
}
title 'Groovy for DSL'
}
```

This will output the following nested XML structure:

```xml
<?xml version="1.0"?>
<book>
  <author>
    <first_name>Fergal</first_name>
    <surname>Dearle</surname>
  </author>
  <title>Groovy for DSL</title>
</book>
```

**Downloading the example code**

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

The method calls on `MarkupBuilder` start off by outputting an opening XML tag, after which they invoke the closure if one has been passed. Finally, the XML tag is properly terminated before the method exits. If we analyze what happens in sequence, we can see that `book` invokes a closure that contains a call to `author`. Additionally, the `author` tag contains a closure with calls to `first_name`, `surname`, and so on.

Before you go to the Groovy documentation for `MarkupBuilder` to look for the `book`, `author`, and `surname` methods in `MarkupBuilder`, let me save you the effort. They don't exist. These are what we call pretend methods. We will see later in the book how Groovy's metaprogramming features allow us to invoke methods on closure that don't really exist, but have them do something useful anyway.

Already, we are seeing how some of the features of the Groovy language can coalesce to allow the structuring of a very useful DSL. I use the term DSL here for Groovy builders because that is essentially what they are. What initially looks like special language syntax for markup is revealed as being regular closures with a little bit of clever metaprogramming. The result is an embedded or internal DSL for generating markup.
Summary

So, now we have a feel for DSLs and Groovy. We have seen how DSLs can be used in place of general-purpose languages to represent different parts of a system. We have also seen how adding DSLs to our applications can open up the development process to other stakeholders in the development process. We've also seen how, in extreme cases, the stakeholders themselves can even become co-developers of the system by using DSLs that let them represent their domain expertise in code.

We've seen how using a DSL that makes sense to a nontechnical audience means it can become a shared resource between programming staff and business stakeholders, representing parts of the system in a language that they all understand. So, we are beginning to understand the importance of usability when designing a DSL.

We have dipped a tentative toe in the water by looking at some Groovy code. We've gained an appreciation of how Groovy is a natural fit with the Java language due to its binary and class level compatibility. We have touched on the features of the Groovy language that make it unique from Java, and looked at how these unique features can be used as a basis for building on the base Groovy language with internal DSLs.

In the next chapter, we will go into more depth with the language itself and see how we can use these features to build programs. In subsequent chapters, we will dive deeper and see how the language can be exploited as an ideal platform for building DSLs on top of the Java platform.
Where to buy this book
You can buy Groovy for Domain-specific Languages Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.