Chapter 1

Introduction to Scripting

The main topic of this book is the synergy of scripting technologies and the Java platform. I describe projects Java developers can use to create a more powerful development environment, and some of the practices that make scripting useful.

Before I start to discuss the application of scripting in the Java world, I summarize some of the theory behind scripting in general and its use in information technology infrastructure. This is the topic of the first two chapters of the book, and it gives us a better perspective of scripting technology as well as how this technology can be useful within the Java platform.

To begin, we must define what scripting languages are and describe their characteristics. Their characteristics greatly determine the roles in which they could (should) be used. In this chapter, I explain what the term *scripting language* means and discuss their basic characteristics.

At the end of this chapter, I discuss the differences between scripting and system-programming languages and how these differences make them suitable for certain roles in development.

Background

The definition of a scripting language is fuzzy and sometimes inconsistent with how scripting languages are used in the real world, so it is a good idea to summarize some of the basic concepts about programming and computing in general. This summary provides a foundation necessary to define scripting languages and discuss their characteristics.

Let's start from the beginning. Processors execute *machine instructions*, which operate on data either in the processors' registers or in external memory. Put simply, a machine instruction is made up of a sequence of binary digits (0s and 1s) and is specific to the particular processor on which it runs. Machine instructions consist of the *operation code* telling the processor what operation it should perform, and *operands* representing the data on which the operation should be performed.

For example, consider the simple operation of adding a value contained in one register to the value contained in another. Now let's imagine a simple processor with an 8-bit instruction set, where the first 5 bits represent the operation code (say, 00111 for register value addition), and the registers are addressed by a 3-bit pattern. We can write this simple example as follows:

```
00111 001 010
```

In this example, I used `001` and `010` to address registers number one and two (`R1` and `R2`, respectively) of the processor.

This basic method of computing has been well known for decades, and I'm sure you are familiar with it. Various kinds of processors have different strategies regarding how their instruction sets should look (RISC or CISC architecture), but from the software developer's point of view, the only important fact is the processor is capable of executing only binary instructions. No matter what programming language is used, the resulting application is a sequence of machine instructions executed by the processor.

What has been changing over time is how people create the order in which the machine instructions are executed. This ordered sequence of machine instructions is called a *computer program*. As hardware is becoming more affordable and more powerful, users' expectations rise. The whole purpose of software development as a science discipline is to provide mechanisms enabling developers to craft more complex applications with the same (or even less) effort as before.

A specific processor's instruction set is called its *machine language*. Machine languages are classified as first-generation programming languages. Programs written in this way are usually very fast because they are optimized for the particular processor's architecture. But despite this benefit, it is hard (if not impossible) for humans to write large and secure applications in machine languages because humans are not good at dealing with large sequences of 0s and 1s.

In an attempt to solve this problem, developers began creating symbols for certain binary patterns, and with this, *assembly languages* were introduced. Assembly languages are *second-generation programming languages*. The instructions in assembly languages are just one level above machine instructions, in that they replace binary digits with easy-to-remember keywords such as `ADD`, `SUB`, and so on. As such, you can rewrite the preceding simple instruction example in assembly language as follows:

```
ADD R1, R2
```

In this example, the `ADD` keyword represents the operation code of the instruction, and `R1` and `R2` define the registers involved in the operation. Even if you observe just this simple example, it is obvious assembly languages made programs easier for humans to read and thus enabled creation of more complex applications.

Although they are much more human-oriented, however, second-generation languages do not extend processor capabilities by any means.

Enter *high-level languages*, which allow developers to express themselves in higher-level, semantic forms. As you might have guessed, these languages are referred to as *third-generation programming languages*. High-level languages provide various powerful loops, data structures, objects, and so on, making it much easier to craft many applications with them.

Over time, a diverse array of high-level programming languages were introduced, and their characteristics varied a great deal. Some of these characteristics categorize programming languages as scripting (or dynamic) languages, as we see in the coming sections.

Also, there is a difference in how programming languages are executed on the host machine. Usually, *compilers* translate high-level language constructs into machine instructions that reside in memory. Although programs written in this way initially were slightly less efficient than programs written in assembly language because of early compilers' inability to use system resources efficiently, as time passed compilers and machines improved, making system-programming languages superior to assembly languages. Eventually, high-level languages became popular in a wide range of development areas, from business applications and games to communications software and operating system implementations.

But there is another way to transform high-level semantic constructs into machine instructions, and that is to interpret them as they are executed. This way, your applications reside in scripts, in their original form, and the constructs are transformed at runtime by a program called an *interpreter*. Basically, you are executing the interpreter that reads statements of your application and then executes them. Called *scripting* or *dynamic languages*, such languages offer an even higher level of abstraction than that offered by system-programming languages, and we discuss them in detail later in this chapter.

Languages with these characteristics are a natural fit for certain tasks, such as process automation, system administration, and gluing existing software components together; in short, anywhere the strict syntax and constraints introduced by system-programming languages were getting in the way

between developers and their jobs. A description of the usual roles of scripting languages is a focus of Chapter 2, "Appropriate Applications for Scripting Languages."

But what does all this have to do with you as a Java developer? To answer this question, let's first briefly summarize the history of the Java platform. As platforms became more diverse, it became increasingly difficult for developers to write software that can run on the majority of available systems. This is when Sun Microsystems developed Java, which offers "write once, run anywhere" simplicity.

The main idea behind the Java platform was to implement a virtual processor as a software component, called a *virtual machine*. When we have such a virtual machine, we can write and compile the code for that processor, instead of the specific hardware platform or operating system. The output of this compilation process is called *bytecode*, and it practically represents the machine code of the targeted virtual machine. When the application is executed, the virtual machine is started, and the bytecode is interpreted. It is obvious an application developed in this way can run on any platform with an appropriate virtual machine installed. This approach to software development found many interesting uses.

The main motivation for the invention of the Java platform was to create an environment for the development of easy, portable, network-aware client software. But mostly due to performance penalties introduced by the virtual machine, Java is now best suited in the area of server software development. It is clear as personal computers increase in speed, more desktop applications are being written in Java. This trend only continues.

One of the basic requirements of a scripting language is to have an interpreter or some kind of virtual machine. The Java platform comes with the Java Virtual Machine (JVM), which enables it to be a host to various scripting languages. There is a growing interest in this area today in the Java community. Few projects exist that are trying to provide Java developers with the same power developers of traditional scripting languages have. Also, there is a way to execute your existing application written in a dynamic language such as Python inside the JVM and integrate it with another Java application or module.

This is what we discuss in this book. We take a scripting approach to programming, while discussing all the strengths and weaknesses of this approach, how to best use scripts in an application architecture, and what tools are available today inside the JVM.

Definition of a Scripting Language

There are many definitions of the term *scripting language*, and every definition you can find does not fully match some of the languages known to be representatives of scripting languages. Some people categorize languages by their purpose and others by their features and the concepts they introduce. In this chapter, we discuss all the characteristics defining a scripting language. In Chapter 2, we categorize scripting languages based on their role in the development process.

Compilers Versus Interpreters

Strictly speaking, an *interpreter* is a computer program that executes other high-level programs line by line. Languages executed only by interpreters are called *interpreted languages*.

To better understand the differences between compilers and interpreters, let's take a brief look at compiler architecture (see Figure 1.1).

As you can see in Figure 1.1, translating source code to machine code involves several steps:

1. First, the source code (which is in textual form) is read character by character. The scanner groups individual characters into valid language constructs (such as variables, reserved words, and so on), called *tokens*.
2. The tokens are passed to the parser, which checks that the correct language syntax is being used in the program. In this step, the program is converted to its parse tree representation.
3. Semantic analysis performs type checking. Type checking validates that all variables, functions, and so on, in

the source program have been used consistently with their definitions. The result of this phase is intermediate representation (IR) code.

4. Next, the optimizer (optionally) tries to make equivalent but improved IR code.
5. In the final step, the code generator creates target machine code from the optimized IR code. The generated machine code is written as an object file.

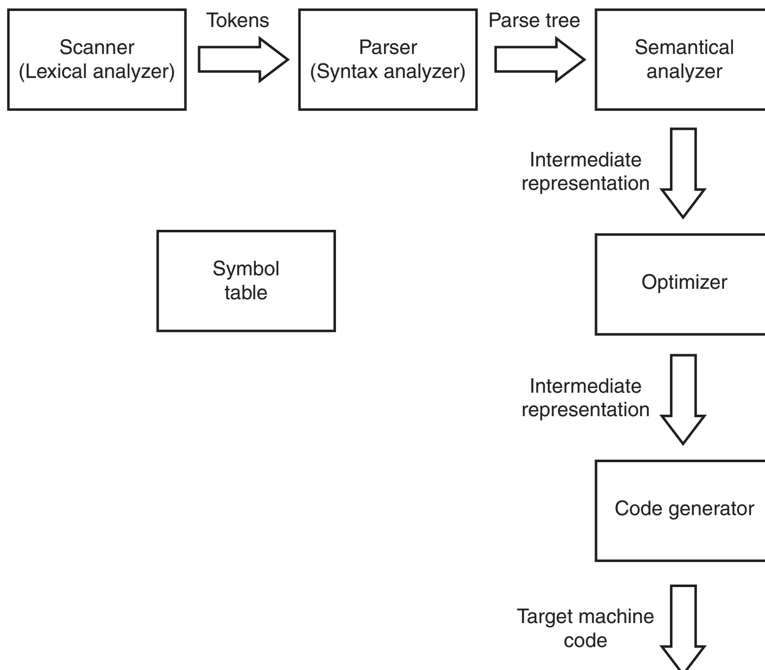

FIGURE 1.1 Compiler architecture

To create one executable file, a linking phase is necessary. The linker takes several object files and libraries, resolves all external references, and creates one executable object file. When such a compiled program is executed, it has complete control of its execution.

Unlike compilers, interpreters handle programs as data that can be manipulated in any suitable way (see Figure 1.2).

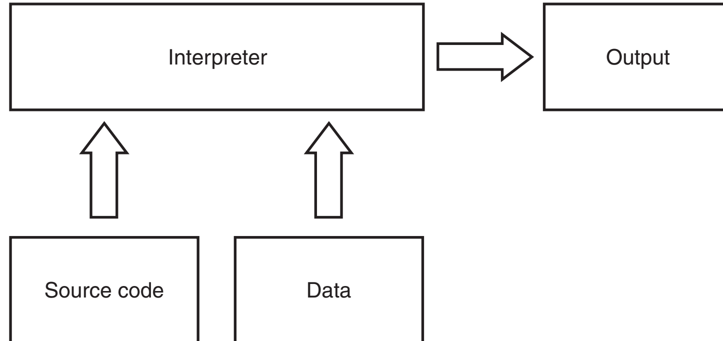

FIGURE 1.2 Interpreter architecture

As you can see in Figure 1.2, the interpreter, not the user program, controls program execution. Thus, we can say the user program is passive in this case. So, to run an interpreted program on a host, both the source code and a suitable interpreter must be available. The presence of the program source (script) is the reason why some developers associate interpreted languages with scripting languages. In the same manner, compiled languages are usually associated with system-programming languages.

Interpreters usually support two modes of operation. In the first mode, the script file (with the source code) is passed to the interpreter. This is the most common way of distributing scripted programs. In the second, the interpreter is run in interactive mode. This mode enables the developer to enter program statements line by line, seeing the result of the execution after every statement. Source code is not saved to the file. This mode is important for initial system debugging, as we see later in the book.

In the following sections, I provide more details on the strengths and weaknesses of using compilers and interpreters. For now, here are some clear drawbacks of both approaches important for our further discussion:

- It is obvious compiled programs usually run faster than interpreted ones. This is because with compiled programs, no high-level code analysis is being done during runtime.

- An interpreter enables the modification of a user program as it runs, which enables interactive debugging capability. In general, interpreted programs are much easier to debug because most interpreters point directly to errors in the source code.
- Interpreters introduce a certain level of machine independence because no specific machine code is generated.
- The important thing from a scripting point of view, as we see in a moment, is interpreters allow the variable type to change dynamically. Because the user program is reexamined constantly during execution, variables do not need to have fixed types. This is much harder to accomplish with compilers because semantic analysis is done at compile time.

From this list, we can conclude interpreters are better suited for the development process, and compiled programs are better suited for production use. Because of this, for some languages, you can find both an interpreter and a compiler. This means you can reap all the benefits of interpreters in the development phase and then compile a final version of the program for a specific platform to gain better performance.

Many of today's interpreted languages are not interpreted purely. Rather, they use a hybrid compiler-interpreter approach, as shown in Figure 1.3.

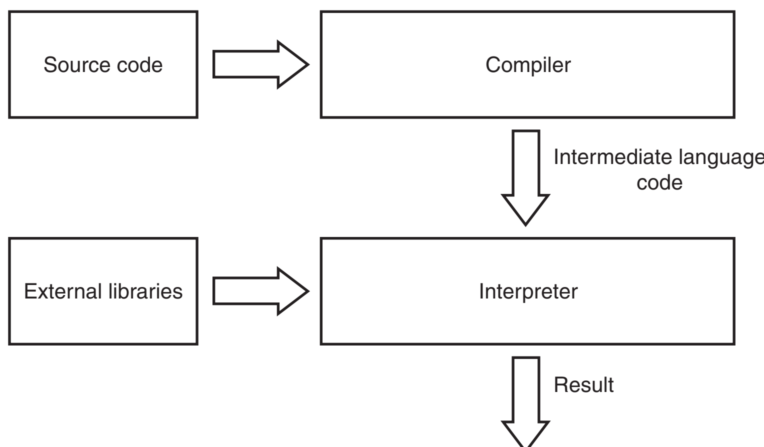

FIGURE 1.3 Hybrid compiler-interpreter architecture

In this model, the source code is first compiled to some intermediate code (such as Java bytecode), which is then interpreted. This intermediate code is usually designed to be very compact (it has been compressed and optimized). Also, this language is not tied to any specific machine. It is designed for some kind of virtual machine, which could be implemented in software. Basically, the virtual machine represents some kind of processor, whereas this intermediate code (bytecode) could be seen as a machine language for this processor.

This hybrid approach is a compromise between pure interpreted and compiled languages, due to the following characteristics:

- Because the bytecode is optimized and compact, interpreting overhead is minimized compared with purely interpreted languages.
- The platform independence of interpreted languages is inherited from purely interpreted languages because the intermediate code could be executed on any host with a suitable virtual machine.

Lately, just-in-time compiler technology has been introduced, which allows developers to compile bytecode to machine-specific code to gain performance similar to compiled languages. I mention this technology throughout the book, where applicable.

Source Code in Production

As some people have pointed out, you should use a scripting language to write user-readable and modifiable programs that perform simple operations and control the execution of other programs. In this scenario, source code should be available in the production system at runtime, so programs are delivered not in object code, but in plain text files (scripts) in their original source. From our previous discussion of interpreters, it is obvious this holds true for purely interpreted languages. Because scripting languages are interpreted, we can say this rule applies to them as well. But because some of them use a hybrid compilation-interpretation strategy, it is possible to deliver the

program in intermediate bytecode form. The presence of the bytecode improves execution speed because no compilation process is required. The usual approach is to deliver necessary libraries in the bytecode and not the program itself. This way, execution speed is improved, and the program source is still readable in production. Some of the compiler-interpreter languages cache in the file the bytecode for the script on its first execution. On every following script execution, if the source hasn't been changed, the interpreter uses the cached bytecode, improving the startup speed required to execute the script.

As such, the presence of source code in the production environment is one of the characteristics of scripting languages, although you can omit it for performance reasons or if you want to keep your source code secret.

Typing Strategies

Before I start a discussion on typing strategies implemented in different programming languages, I have to explain what types are.

There is no simple way to explain what typing is because its definition depends on the context in which it is used. Also, a whole branch of mathematics is dedicated to this issue. It is called *type theory*, and its proponents have the following saying, which emphasizes their attitude toward the importance of this topic:

Design the type system correctly, and the language will design itself.

To put it simply, types are metadata that describe the data stored in some variable. Types specify what values can be stored in certain variables, as well as the operations that can be performed on them.

Type constraints determine how we can handle and operate a certain variable. For example, what happens when you add the values of one variable to those of another depends on whether the variables are integers, floats, Booleans, or strings. A programming language's type system could classify the value

`hello` as a string and the value `7` as a number. Whether you can mix strings with numbers in this language depends on the language's type policy.

Some types are *native* (or *primitive*), meaning they are built into the language. The usual representatives of this type category are Booleans, integers, floats, characters, and even strings in some languages. These types have no visible internal structure.

Other types are *composite,* and are constructed of primitive types. In this category, we have structures and various so-called container types, such as lists, maps, and sets. In some languages, *string* is defined as a list of characters, so it can be categorized as a composite type.

In object-oriented languages, developers got the opportunity to create their own types, also known as *classes*. This type category is called *user-defined types*. The big difference between structures and classes is with classes, you define not just the structure of your complex data, but also the behavior and possible operations you can perform with it. This categorizes every class as a single type, where structures (in C, for example) are one type.

Type systems provide the following major benefits:

- **Safety**—Type systems are designed to catch the majority of type-misuse mistakes made by developers. In other words, types make it practically impossible to code some operations that cannot be valid in a certain context.
- **Optimization**—As I already mentioned, languages that employ static typing result in programs with better-optimized machine code. That is because early type checks provide useful information to the compiler, making it easier to allocate optimized space in memory for a certain variable. For example, there is a great difference in memory usage when you are dealing with a Boolean variable versus a variable containing some random text.
- **Abstraction**—Types allow developers to make better abstractions in their code, enabling them to think about programs at a higher level of abstraction, not bothering

with low-level implementation of those types. The most obvious example of this is in the way developers deal with strings. It is much more useful to think of a string as a text value rather than as a byte array.

- **Modularity**—Types allow developers to create application programming interfaces (APIs) for the subsystems used to build applications. Typing localizes the definitions required for interoperability of subsystems and prevents inconsistencies when those subsystems communicate.
- **Documentation**—Use of types in languages can improve the overall documentation of the code. For example, a declaration that some method's arguments are of a specific type documents how that method can be used. The same is true for return values of methods and variables.

Now that we know the basic concepts of types and typing systems, we can discuss the type strategies implemented in various languages. We also discuss how the choice of implemented typing system defines languages as either scripting (dynamic) or static.

DYNAMIC TYPING

The type-checking process verifies that the constraints introduced by types are being respected. System-programming languages traditionally used to do type checking at compile time. This is referred to as *static typing.*

Scripting languages force another approach to typing. With this approach, type checking is done at runtime. One obvious consequence of runtime checking is all errors caused by inappropriate use of a type are triggered at runtime. Consider the following example:

```
x = 7
y = “hello world”
z = x + y
```

This code snippet defines an integer variable, `x`, and a string variable, `y`, and then tries to assign a value for the `z` variable that is the sum of the `x` and `y` values. If the language has not

defined an operator, +, for these two types, different things happen depending on whether the language is statically or dynamically typed. If the language was statically typed, this problem would be discovered at compile time, so the developer would be notified of it and forced to fix it before even being able to run the program. If the language was dynamically typed, the program would be executable, but when it tried to execute this problematic line, a runtime error would be triggered.

Dynamic typing usually allows a variable to change type during program execution. For example, the following code would generate a compile-time error in most statically typed programming languages:

```
x = 7
x = "Hello world"
```

On the other hand, this code would be legal in a purely dynamic typing language. This is simply because the type is not being misused here.

Dynamic typing is usually implemented by tagging the variables. For example, in our previous code snippet, the value of variable `x` after the first line would be internally represented as a pair (`7`, `number`). After the second line, the value would be internally represented as a pair (`"Hello world"`, `string`). When the operation is executed on the variable, the type is checked and a runtime error is triggered if the misuse is discovered. Because no misuse is detected in the previous example, the code snippet runs without raising any errors.

I comprehensively discuss the pros and cons of these approaches later in this chapter, but for now, it is important to note a key benefit of dynamic typing from the developer's point of view. Programs written in dynamically typed languages tend to be much shorter than equivalent solutions written in statically typed languages. This is an implication of the fact that developers have much more freedom in terms of expressing their ideas when they are not constrained by a strict type system.

WEAK TYPING

There is yet another categorization of programming-language typing strategy. Some languages raise an error when a programmer tries to execute an operation on variables whose types are not suitable for that operation (type misuse). These languages are called *strongly typed languages*. On the other hand, *weakly typed languages* implicitly cast (convert) a variable to a suitable type before the operation takes place.

To clarify this, let's take a look at our first example of summing a number and string variable. In a strongly typed environment, which most system-programming languages deploy, this operation results in a compile-time error if no operator is defined for these types. In a weakly typed language, the integer value usually would be converted to its string representative (`7` in this case) and concatenated to the other string value (supposing that the + operator represents string concatenation in this case). The result would be a `z` variable with the "`7HelloWorld`" value and the string type.

Most scripting languages tend to be dynamic and weakly typed, but not all of them use these policies. For example, Python, a popular scripting language, employs dynamic typing, but it is strongly typed. We discuss in more detail the strengths and weaknesses of these typing approaches, and how they can fit into the overall system architecture, later in this chapter and in Chapter 2.

Data Structures

For successful completion of common programming tasks, developers usually need to use different complex data structures. The presence of language mechanisms for easy handling of complex data structures is in direct connection to developers' efficiency.

Scripting languages generally provide more powerful and flexible built-in data types than traditional system-programming languages. It is natural to see data structures such as lists, sets, maps, and so on, as native data types in such languages.

Of course, it is possible to implement an arbitrary data structure in any language, but the point is these data structures are embedded natively in language syntax making them much easier to learn and use. Also, without this standard implementation, novice developers are often tempted to create their own solution that is usually not robust enough for production use.

As an example, let's look at Python, a popular dynamic language with lists and maps (also called dictionaries) as its native language type. You can use these structures with other language constructs, such as a `for` loop, for instance. Look at the following example of defining and iterating a simple list:

```
list = ["Mike", "Joe", "Bruce"]
for item in list :
    print item
```

As you can see, the Python code used in this example to define a list is short and natural. But more important is the `for` loop, which is designed to naturally traverse this kind of data. Both of these features make for a comfortable programming environment and thus save some time for developers.

Java developers may argue that Java collections provide the same capability, but prior to J2SE 1.5, the equivalent Java code would look like this:

```
String[] arr = new String[]{"Mike", "Joe", "Bruce"};
List list = Arrays.asList(arr);
for (Iterator it = list.iterator(); it.hasNext(); ) {
      System.out.println(it.next());
}
```

Even for this simple example, the Java code is almost twice as long as and is much harder to read than the equivalent Python code. In J2SE 1.5, Java got some features that brought it closer to these scripting concepts. With the more flexible `for` loop, you could rewrite the preceding example as follows:

```
String[] arr = new String[]{"Mike", "Joe", "Bruce"};
List list = Arrays.asList(arr);
for (String item : list) {
      System.out.println(item);
}
```

With this in mind, we can conclude data structures are an important part of programming, and therefore native language support for commonly used structures could improve developers' productivity. Many scripting languages come with flexible, built-in data structures, which is one of the reasons why they are often categorized as "human-oriented."

Code as Data

The code and data in compiled system programming languages are two distinct concepts. Scripting languages, however, attempt to make them more similar. As I said earlier, programs (code) in scripting languages are kept in plain text form. Language interpreters naturally treat them as ordinary strings.

EVALUATION

It is not unusual for the commands (built-in functions) in scripting languages to evaluate a string (data) as language expression (code). For example, in Python, you can use the `eval()` function for this purpose:

```
x = 9
eval("print x + 7")
```

This code prints `16` on execution, meaning the value of the variable `x` is embedded into the string, which is evaluated as a regular Python program.

More important is the fact that scripted programs can generate new programs and execute them "on the fly". Look at the following Python example:

```
temp = open("temp.py", "w")
temp.write("print x + 7")
temp.close()
x = 9
execfile("temp.py")
```

In this example, we created a file called `temp.py`, and we wrote a Python expression in it. At the end of the snippet, the `execfile()` command executed the file, at which point `16` was displayed on the console.

This concept is natural to interpreted languages because the interpreter is already running on the given host executing the current script. Evaluation of the script generated at runtime is not different from evaluation of other regular programs. On the other hand, for compiled languages this could be a challenging task. That is because a compile/link phase is introduced during conversion of the source code to the executable program. With interpreted languages, the interpreter must be present in the production environment, and with compiled languages, the compiler (and linker) is usually not part of the production environment.

CLOSURES

Scripting languages also introduce a mechanism for passing blocks of code as method arguments. This mechanism is called a *closure*. A good way to demonstrate closures is to use methods to select items in a list that meet certain criteria.

Imagine a list of integer values. We want to select only those values greater than some threshold value. In Ruby, a scripting language that supports closures, we can write something like this:

```
threshold = 10
newList = orig.select {|item| item > threshold}
```

The `select()` method of the collection object accepts a closure, defined between the `{}`, as an argument. If parameters must be passed, they can be defined between the `||`. In this example, the `select()` method iterates over the collection, passing each item to the closure (as an `item` parameter) and returning a collection of items for which the closure returned `true`.

Another thing worth noting in this example is closures can refer to variables visible in the scope in which the closure is created. That's why we could use the global `threshold` value in the closure.

Closures in scripting languages are not different from any other data type, meaning methods can accept them as parameters and return them as results.

FUNCTIONS AS METHOD ARGUMENTS

Many scripting languages, even object-oriented ones, introduce standalone functions as so-called "first-class language citizens." Even if you do not have true support for closures, you can pass your functions as method arguments.

The Python language, for example, defines a `filter()` function that accepts a list and the function to be executed on every item in the list:

```
def over(item) :
    threshold = 10
    return item > threshold

newList = filter(over, orig)
```

In this example, we defined the `over()` function, which basically does the same job as our closure from the previous example. Next, we called the `filter()` function and passed the `over()` function as the second argument. Even though this mechanism is not as convenient as closures are, it serves its purpose well (and that is to pass blocks of code as data around the application).

Of course, you can achieve similar functionality in other nonscripting languages. For example, Java developers have the concept of anonymous inner classes serving the same purpose. Let's implement a similar solution using this approach:

```
package net.scriptinginjava.ch1;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

interface IFilter {
    public boolean filter(Integer item);
}

public class Filter {

    private static List select(List list, IFilter filter) {
        List result = new ArrayList();
        for (Iterator it = list.iterator(); it.hasNext();) {
            Integer item = (Integer)it.next();
            if (filter.filter(item)) {
```

```
                result.add(item);
            }
        }
        return result;
    }

    public static void main(String[] args) {
        Integer[] arr = new Integer[]{
                    new Integer(5),
                    new Integer(7),
                    new Integer(13),
                    new Integer(32)
                };
        List orig = Arrays.asList(arr);
        List newList = select(orig,
                    new IFilter() {
                        private Integer threshold
                            = new Integer(10);
                        public boolean filter(Integer item) {
                            return item.compareTo(threshold) > 0;
                        }
                    }
        );
        System.out.println(newList);
    }

}
```

> **NOTE**
>
> Some closure proponents say that the existence of this "named" interface breaks the anonymous concept at the beginning.

First we defined the IFilter interface with a filter() method that returns a Boolean value indicating whether the condition is satisfied.

Our Filter class contains a select() method equal to the methods we saw in the earlier Ruby and Python examples. It accepts a list to be handled and the implementation of the IFilter interface that filters the values we want in our new list. At the end, we implement the IFilter interface as the anonymous inner class in the select() method call.

As a result, the program prints this result list to the screen:

```
[13, 32]
```

From this example, we can see even though a similar concept is possible in system-programming languages, the syntax is much more complex. This is an important difference because the natural syntax for some functionality leads to its frequent use, in practice. Closures have simple syntax for passing the

code around the application. That is why you see closures used more often in languages that naturally support them than you see similar structures in other languages (anonymous inner classes in Java, for example).

Hopefully, closures will be added in Java SE 7, which will move Java one step closer to the flexibility of scripting languages.

Summary

In this section of the chapter, I discussed some basic functional characteristics of scripting languages. Many experts tend to categorize a language as scripting or system programming, not by these functional characteristics but by the programming style and the role the language plays in the system. However, these two categorizations are not independent, so to understand how scripting can fit into your development process, it is important to know the functional characteristics of the scripting language and the implications of its design. The differences between system-programming and scripting languages are described later in this chapter, helping us to understand how these two approaches can work together to create systems that feature the strengths of both programming styles.

It is important to note that the characteristics we've discussed thus far are not independent among each other. For example, whether to use static or dynamic typing depends on when the type checking is done. It is hard to implement dynamic typing in a strictly compiled environment. Thus, interpreter and dynamic typing somehow fit naturally together and are usually employed in scripting environments. The same is true for the compiler and static typing found in system-programming environments.

The similar is true for the generation and execution of other programs, which is a natural thing to do in interpreted environments and is not very easy (and thus is rarely done) in compiled environments.

To summarize, these characteristics are usually found in scripting programming environments. Not all languages support all the features described earlier, which is a decision driven by

the primary domain for which the language is used. For example, although Python is a dynamic language, it introduces strong typing, making it more resistible to type misuse and more convenient for development of larger applications.

These characteristics should serve only as a marker when exploring certain languages and their possible use in your development process. More important is the language's programming style, a topic we discuss shortly.

Scripting Languages and Virtual Machines

A recent trend in programming language design is the presence of a virtual machine as one of the vital elements of programming platforms. One of the main elements of the Java Runtime Environment (JRE) is the virtual machine that interprets bytecode and serves as a layer between the application and operating systems. A virtual machine serves as a layer between the application and operating systems in Microsoft's .NET platform as well.

Let's now summarize briefly how the JRE works. Java programs contained in `java` extension source files are compiled to bytecode (files with a `class` extension). As I said earlier, the purpose of bytecode is to provide a compact format for intermediate code and support for platform independence. The JVM is a virtual processor, and like all other processors, it interprets code—bytecode in this case. This is a short description of the JRE, but it is needed for our further discussion. You can find a more comprehensive description at the beginning of Chapter 3, "Scripting Languages Inside the JVM."

Following this, we can say Java is a hybrid compiled-interpreted language. But even with this model, Java cannot be characterized as a scripting language because it lacks all the other features mentioned earlier.

At this point, you are probably asking what this discussion has to do with scripting languages. The point is many modern scripting languages follow the same hybrid concept. Although programs are distributed in script form and are interpreted at

runtime, the things going on in the background are pretty much the same.

Let's look at Python, for example. The Python interpreter consists of a compiler that compiles source code to the intermediate bytecode, and the Python Virtual Machine (PVM) that interprets this code. This process is being done in the background, leaving the impression that the pure Python source code has been interpreted. If the Python interpreter has write privileges on the host system, it caches the generated bytecode in files with a pyc extension (the py extension is used for the scripts or source code). If that script had not been modified since its previous execution, the compilation process would be skipped and the virtual machine could start interpreting the bytecode at once. This could greatly improve the Python script's startup speed. Even if the Python interpreter has no write privileges on the system and the bytecode was not written in files, this compilation process would still be performed. In this case, the bytecode would be kept in memory.

> **NOTE**
> Python programs can be distributed in bytecode format, keeping the source code out of the production environment.

From this discussion, we can conclude virtual machines are one of the standard parts of modern scripting languages. So our original dilemma remains. Should we use languages that enforce a certain programming paradigm, and if so, how do we use them? The dynamic and weak typing, closures, complex built-in data structures, and so on, could be implemented in a runtime environment with the virtual machine.

There is nothing to restrict the use of a dynamic (scripting) language on the virtual machines designed for languages such as Java and C#. As long as we implement the compiler appropriate for the target virtual machine's intermediate bytecode, we will receive all the features of the scripting language in this environment. Doing this, we could benefit from the strengths of both the system-programming approach of Java, and the scripting programming model in our software development process.

We focus on projects that bring scripting languages closer to the Java platform later in this book. Also, we discuss where it's appropriate to apply the scripting style of development with traditional Java programming. Before we cover these topics, though, let's take a look at how scripting and system programming compare.

A Comparison of Scripting and System Programming

Every decision made during the language design process is directly related to the programming style used in that language and its usability in the development process.

In this section, I do not intend to imply one style is better than the other is. Instead, my objective is to summarize the strengths and weaknesses of both approaches so that we can proceed to Chapter 2, where I discuss how best to incorporate them into the development process.

Runtime Performance

It is clear programs written in system-programming languages have better runtime performance than equivalent scripts in most cases, for a few reasons:

- The most obvious reason is the runtime presence of the interpreter in scripting languages. Source code analysis and transformation during runtime introduces additional overhead in terms of program execution.
- Another factor influencing runtime performance is typing. Because system-programming languages force strong static typing, machine code created by the compiler is more compact and optimized for the target machine.

The fact that the script could be compiled to intermediate bytecode makes these interpreter performance penalties more acceptable. But the machine code is definitely more optimized than the intermediate code.

We have to take another point of view when talking about runtime performance, however. Many people approach runtime performance by asking which solution is faster. The more important question, which is often neglected, is whether a particular solution is fast enough.

You must take into consideration the tradeoffs between the benefits and the runtime performance that each approach

provides when you are thinking about applying a certain technology in your project. If the solution brings quality to your development process and still is fast enough, you should consider using it.

A recent development trend supports this point of view. Many experts state you should not analyze performance without comparing it to measurements and goals. This leads to debate concerning whether to perform premature or prudent optimization. The latter approach assumes you have a flexible system, and only after you've conducted the performance tests and found the system bottlenecks should you optimize those parts of your code.

Deciding whether scripting is suitable for some tasks in your development process must be driven by the same question. For instance, say you need to load a large amount of data from a file, and developing a system-programming solution to accomplish the task would take twice as long as developing a scripting approach. If both the system-programming and scripting solutions need 1 second to load the data and the interpreter required an additional 0.1 second to compile the script to the bytecode, you should consider scripting to be a fast enough solution for this task. As we see in a moment, scripts are much faster to write (because of the higher level of abstraction they introduce), and the end users of your project probably wouldn't even notice the performance advantage of the system-programming solution that took twice as much time to develop.

If we take another point of view, we can conclude the startup cost of executing programs written in dynamic languages could be close to their compiled alternatives. The first important thing to note is the fact that bytecode is usually smaller than its equivalent machine code. Experts who support this point of view stress that processors have increased in speed much faster than disks have. This leads to the thinking that the in-memory operations of the just-in-time compilers (compiling the bytecode to the machine code) are not much more expensive than the operation of loading the large sequence of machine code from the disk into memory.

To summarize, it is clear system-programming languages are faster than scripting languages. But if you don't need to be

restricted by only one programming language, you should ask yourself another question: What is the best tool for this task? If the development speed is more important and the runtime performance of the scripting solution is acceptable, there is your answer.

Development Speed

I already mentioned dynamic languages lead to faster development processes. A few facts support this assertion.

For one, a statement in a system-programming language executes about five machine instructions. However, a statement in a scripting language executes hundreds or even thousands of instructions. Certainly, this increase is partially due to the presence of the interpreter, but more important is the fact that primitive operations in scripting languages have greater functionality. For example, operations for matching certain patterns in text with regular expressions are as easy to perform as multiplying two integers.

These more powerful statements and built-in data structures lead to a higher level of abstraction that language can provide, as well as much shorter code.

Of course, dynamic typing plays an important role here too. The need to define each variable explicitly with its type requires a lot of typing, and this is time consuming from a developer's perspective. This higher level of abstraction and dynamic typing allows developers to spend more time writing the actual business logic of the application than dealing with the language issues.

Another thing speeding up the scripting development process is the lack of a compile (and linking) phase. Compilation of large programs could be time consuming. Every change in a program written in a system-programming language requires a new compile/link process, which could slow down development a great deal. In scripting, on the other hand, immediately after the code is written or changed, it can be executed (interpreted), leaving more time for the developer to actually write the code.

As you can see, all the things that increase runtime performance, such as compilation and static typing, tend to slow

down development and increase the amount of time needed to build the solution. That is why you hear scripting languages are more human oriented than machine oriented (which isn't the case with system-programming languages).

To emphasize this point further, here is a snippet from David Ascher's article titled "Dynamic Languages—ready for the next challenges, by design" (www.activestate.com/Company/NewsRoom/whitepapers_ADL.plex), which reflects the paradigm of scripting language design:

The driving forces for the creation of each major dynamic language centered on making tasks easier for people, with raw computer performance a secondary concern. As the language implementations have matured, they have enabled programmers to build very efficient software, but that was never their primary focus. Getting the job done fast is typically prioritized above getting the job done so that it runs faster. This approach makes sense when one considers that many programs are run only periodically, and take effectively no time to execute, but can take days, weeks, or months to write. When considering networked applications, where network latency or database accesses tend to be the bottlenecks, the folly of hyper-optimizing the execution time of the wrong parts of the program is even clearer. A notable consequence of this difference in priority is seen in the different types of competition among languages. While system languages compete like CPU manufacturers on performance measured by numeric benchmarks such as LINPACK, dynamic languages compete, less formally, on productivity arguments and, through an indirect measure of productivity, on how "fun" a language is. It is apparently widely believed that fun languages correspond to more productive programmers—a hypothesis that would be interesting to test.

Robustness

Many proponents of the system-programming approach say dynamic typing introduces more bugs in programs because there is no type checking at compile time. From this point of view, it is always good to detect programming errors as soon as

possible. This is certainly true, but as we discuss in a moment, static typing introduces some drawbacks, and programs written in dynamically typed languages could be as solid as programs written in purely statically typed environments. This way of thinking leads to the theory that dynamically typed languages are good for building prototypes quickly, but they are not robust enough for industrial-strength systems.

On the other side stand proponents of dynamic typing. From that point of view, type errors are just one source of bugs in an application, and programs free of type-error problems are not guaranteed to be free of bugs. Their attitude is static typing leads to code much longer and much harder to maintain. Also, static typing requires the developer to spend more of his time and energy working around the limitations of that kind of typing.

Another implication we can glean from this is the importance of testing. Because a successful compilation does not guarantee your program will behave correctly, appropriate testing must be done in both environments. Or as best-selling Java author Bruce Eckel wrote in his book *Thinking in Java* (Prentice Hall):

If it's not tested, it's broken.

Because dynamic typing allows you to implement functionality faster, more time remains for testing. Those fine-grained tests could include testing program behavior for type misuse.

Despite all the hype about type checking, type errors are not common in practice, and they are discovered quickly in the development process. Look at the most obvious example. With no types declared for method parameters, you could easily find yourself calling a method with the wrong order of parameters. But these kinds of errors are obvious and are detected immediately the next time the script is executed. It is highly unlikely this kind of error would make it to distribution if it was tested appropriately.

Another extreme point of view says even statically typed languages are not typed. To clarify this statement, look at the following Java code:

```
List list = new ArrayList();
list.add(new String("Hello"));
list.add(new Integer(77));

Iterator it = list.iterator();
while (it.hasNext()) {
    String item = (String)it.next();
}
```

This code snippet would be compiled with no errors, but at execution time, it would throw a `java.lang.ClassCastException`. This is a classic example of a runtime type error. So what is the problem?

The problem is objects lose their type information when they are going through more-generic structures. In Java, all objects in the container are of type `java.lang.Object`, and they must be converted to the appropriate type (class) as soon as they are released from the container. This is when inappropriate object casting could result in runtime type errors. Because many objects in the application are actually contained in a more-generic structure, this is not an irrelevant issue.

Of course, there is a workaround for this problem in statically typed languages. One solution recently introduced in Java is called *generics*. With generics, you would write the preceding example as follows:

```
List list<String> = new ArrayList<String>();
list.add(new String("Hello"));
list.add(new Integer(77));

Iterator<String> it = list.iterator();
while (it.hasNext()) {
    String item = it.next();
}
```

This way, you are telling the compiler only `String` objects can be placed in this container. An attempt to add an `Integer` object would result in a compilation error. This is a solution to this problem, but like all workarounds, it is not a natural approach.

The fact that scripting programs are smaller and more readable by humans makes them more suitable for code review by a

development team, which is one more way to ensure your application is correct. Guido van Rossum, the creator of the Python language, supported this view when he was asked in an interview whether he would fly an airplane controlled by software written in Python (www.artima.com/intv/strongweakP.html):

You'll never get all the bugs out. Making the code easier to read and write, and more transparent to the team of human readers who will review the source code, may be much more valuable than the narrow-focused type checking that some other compiler offers. There have been reported anecdotes about spacecraft or aircraft crashing because of type-related software bugs, where the compilers weren't enough to save you from the problems.

This discussion is intended just to emphasize one thing: Type errors are just one kind of bug in a program. Early type checking is a good thing, but it is certainly not enough, so conducting appropriate quality assurance procedures (including unit testing) is the only way to build stable and robust systems.

Many huge projects written purely in Python prove the fact that modern scripting languages are ready for building large and stable applications.

Maintenance

A few aspects of scripting make programs written in scripting languages easier to maintain.

The first important aspect is the fact that programs written in scripting languages are shorter than their system-programming equivalents, due to the natural integration of complex data types, more powerful statements, and dynamic typing. Simple logic dictates it is easier to debug and add additional features to a shorter program than to a longer one, regardless of what programming language it was written in. Here's a more descriptive discussion on this topic, taken from the aforementioned Guido van Rossum interview (www.artima.com/intv/speed.html):

This is all very informal, but I heard someone say a good programmer can reasonably maintain about 20,000 lines of code.

Whether that is 20,000 lines of assembler, C, or some high-level language doesn't matter. It's still 20,000 lines. If your language requires fewer lines to express the same ideas, you can spend more time on stuff that otherwise would go beyond those 20,000 lines.

A 20,000-line Python program would probably be a 100,000-line Java or C++ program. It might be a 200,000-line C program, because C offers you even less structure. Looking for a bug or making a systematic change is much more work in a 100,000-line program than in a 20,000-line program. For smaller scales, it works in the same way. A 500-line program feels much different than a 10,000-line program.

The counterargument to this is the claim that static typing also represents a kind of code documentation. Having every variable, method argument, and return result in a defined type makes code more readable. Although this is a valid claim when it comes to method and property declarations, it certainly is not important to document every temporary variable. Also, in almost every programming language you can find a mechanism and tools used to document your code. For example, Java developers usually use the Javadoc tool (http://java.sun.com/j2se/javadoc/) to generate HTML documentation from specially formatted comments in source code. This kind of documentation is more comprehensive and could be used both in scripting and in system-programming languages.

Also, almost every dynamically typed language *permits* explicit type declaration but does not *force* it. Every scripting developer is free to choose where explicit type declarations should be used and where they are sufficient. This could result in both a rapid development environment and readable, documented code.

Extreme Programming

In the past few years, many organizations adopted extreme programming as their software development methodology. The two basic principles of extreme programming are *test-driven development* (TDD) and *refactoring*.

You can view the TDD technique as a kind of revolution in the way people create programs. Instead of performing the following:

1. Write the code.
2. Test it if appropriate.

The TDD cycle incorporates these steps:

1. Write the test for certain program functionality.
2. Write enough code to get it to fail (API).
3. Run the test and watch it fail.
4. Write the whole functionality.
5. Run the code and watch all tests pass.

On top of this development cycle, the extreme programming methodology introduces refactoring as a technique for code improvement and maintenance. Refactoring is the technique of restructuring the existing code body without changing its external behavior. The idea of refactoring is to keep the code design clean, avoid code duplication, and improve bad design. These changes should be small because that way, it is likely we will not break the existing functionality.

After code refactoring, we have to run all the tests again to make sure the program is still behaving according to its design.

I already stated tests are one way to improve our programs' robustness and to prevent type errors in dynamically typed languages. From the refactoring point of view, interpreted languages offer benefits because they skip the compilation process during development. For applications developed using the system-programming language, after every small change (refactoring), you have to do compilation and run tests. Both of these operations could be time consuming on a large code base, so the fact that compilation could be omitted means we can save some time.

Dynamic typing is a real advance in terms of refactoring. Usually, because of laziness or a lack of the big picture, a developer defines a method with as narrow an argument type as he needs at that moment. To reuse that method later, we have to

change the argument type to some more general or complex structure. If this type is a concrete type or does not share the same interface as the one we used previously, we are in trouble. Not only do we have to change that method definition, but also the types of all variables passed to that method as the particular argument. In dynamically typed languages, this problem does not exist. All you need to do is change the method to handle this more general type.

We could amortize these problems in system programming environments with good refactoring tools, which exist for most IDEs today. Again, the real benefit is speed of development. Because scripting languages enable developers to write code faster, they have more time to do appropriate unit testing and to write stub classes. A higher level of abstraction and a dynamic nature make scripted programs more convenient to change, so we can say they naturally fit the extreme programming methodology.

The Hybrid Approach

As we learned earlier in this chapter, neither system-programming nor scripting languages are ideal tools for all development tasks. System-programming languages have good runtime performance, but developing certain functionality and being able to modify that functionality later takes time. Scripting languages, on the other hand, are the opposite. Their flexible and dynamic nature makes them an excellent development environment, but at the cost of runtime performance.

So the real question is not whether you should use a certain system-programming or scripting language for all your development tasks, but where and how each approach fits into your project. Considering today's diverse array of programming platforms and the many ways in which you can integrate them, there is no excuse for a programmer to be stuck with only one programming language. Knowing at least two languages could help you have a better perspective of the task at hand, and the appropriate tool for that task.

You can find a more illustrative description of this principle in Bill Venners's article, "The Best Tool for the Job" (www.artima.com/commentary/langtool.html):

To me, attempting to use one language for every programming task is like attempting to use one tool for every carpentry task. You may really like screwdrivers, and your screwdriver may work great for a job like inserting screws into wood. But what if you're handed a nail? You could conceivably use the butt of the screwdriver's handle and pound that nail into the wood. The trouble is, a) you are likely to put an eye out, and b) you won't be as productive pounding in that nail with a screwdriver as you would with a hammer.

Because learning a new programming language requires so much time and effort, most programmers find it impractical to learn many languages well. But I think most programmers could learn two *languages well. If you program primarily in a systems language, find a scripting language that suits you and learn it well enough to use it regularly. I have found that having both a systems and a scripting language in the toolbox is a powerful combination. You can apply the most appropriate tool to the programming job at hand.*

So if we agree system-programming and scripting languages should be used together for different tasks in project development, two more questions arise. The first, and the most important one, is what tasks are suitable for a certain tool.

The second question concerns what additional characteristics scripting languages should have to fit these development roles.

Let's try to answer these two questions by elaborating on the most common roles (and characteristics) scripting languages had in the past. This gives us a clear vision of how we can apply them to the development challenges in Java projects today, which is the topic of later chapters.

A Case for Scripting

To end our discussion of this topic, I quote John K. Ousterhout, the creator of the Tcl scripting language. In one of his articles (www.tcl.tk/doc/scripting.html), he wrote the following words:

In deciding whether to use a scripting language or a system programming language for a particular task, consider the following questions:

Is the application's main task to connect together pre-existing components?

Will the application manipulate a variety of different kinds of things?

Does the application include a graphical user interface?

Does the application do a lot of string manipulation?

Will the application's functions evolve rapidly over time?

Does the application need to be extensible?

"Yes" answers to these questions suggest that a scripting language will work well for the application. On the other hand, "yes" answers to the following questions suggest that an application is better suited to a system programming language:

Does the application implement complex algorithms or data structures?

Does the application manipulate large datasets (e.g., all the pixels in an image) so that execution speed is critical?

Are the application's functions well-defined and changing slowly?

You could translate Ousterhout's comments as follows: Dynamic languages are well suited for implementing application parts not defined clearly at the time of development, for wiring (gluing) existing components in a loosely coupled manner, and for implementing all those parts that have to be flexible and changeable over time. System languages, on the other hand, are

a good fit for implementing complex algorithms and data structures, and for all those components that are well defined and probably won't be modified extensively in the future.

Conclusion

In this chapter, I explained what scripting languages are and discussed some basic features found in such environments. After that, I compared those features to system-programming languages in some key development areas. Next, I expressed the need for software developers to master at least one representative of both system-programming and scripting languages. And finally, I briefly described suitable tasks for both of these approaches.

Before we proceed to particular technologies that enable usage of scripting languages in Java applications, we focus in more detail on the traditional roles of scripting languages. This is the topic of Chapter 2, and it helps us to better understand scripting and how it can be useful in the overall system infrastructure.