SAMPLE CHAPTER

Vital techniques of Java 7 and polyglot programming

The Well-Grounded Java Developer

Benjamin J. Evans
Martijn Verburg

FOREWORD BY Dr. Heinz Kabutz

MANNING

The Well-Grounded Java Developer

by Benjamin J. Evans
Martijn Verburg

Chapter 1

brief contents

PART 1 DEVELOPING WITH JAVA 7 1

1 ■ Introducing Java 7 3
2 ■ New I/O 20

PART 2 VITAL TECHNIQUES 51

3 ■ Dependency Injection 53
4 ■ Modern concurrency 76
5 ■ Class files and bytecode 119
6 ■ Understanding performance tuning 150

PART 3 POLYGLOT PROGRAMMING ON THE JVM.....191

7 ■ Alternative JVM languages 193
8 ■ Groovy: Java's dynamic friend 213
9 ■ Scala: powerful and concise 241
10 ■ Clojure: safer programming 279

PART 4 CRAFTING THE POLYGLOT PROJECT 311

11 ■ Test-driven development 313
12 ■ Build and continuous integration 342
13 ■ Rapid web development 380
14 ■ Staying well-grounded 410

1 Introducing Java 7

This chapter covers

- Java as a platform and a language
- Small yet powerful syntax changes
- The try-with-resources statement
- Exception-handling enhancements

Welcome to Java 7. Things around here are a little different than you may be used to. This is a really good thing—we have a lot to explore, now that the dust has settled and Java 7 has been unleashed. By the time you finish this book, you'll have taken your first steps into a larger world—a world of new features, of software craftsmanship, and of other languages on the Java Virtual Machine (JVM).

We're going to warm up with a gentle introduction to Java 7, but one that still acquaints you with powerful features. We'll start by explaining a distinction that is sometimes misunderstood—the duality between the *language* and the *platform*.

After that, we'll introduce Project Coin—a collection of small yet effective new features in Java 7. We'll show you what's involved in getting a change to the Java platform accepted, incorporated, and released. With that process covered, we'll move on to the six main new features that were introduced as part of Project Coin.

You'll learn new syntax, such as an improved way of handling exceptions (multi-catch) as well as try-with-resources, which helps you avoid bugs in code that deals

with files or other resources. By the end of this chapter, you'll be writing Java in a new way and you'll be fully primed and ready for the big topics that lie ahead.

Let's get under way by discussing the language versus platform duality that lies at the heart of modern Java. This is a critically important point that we'll come back to again throughout the book, so it's an essential one to grasp.

1.1 The language and the platform

The critical concept we're kicking off with is the distinction between the Java language and the Java platform. Surprisingly, different authors sometimes give slightly different definitions of what constitutes the language and platform. This can lead to a lack of clarity and some confusion about the differences between the two and about which provides the programming features that application code uses.

Let's make that distinction clear right now, as it cuts to the heart of a lot of the topics in this book. Here are our definitions:

- *The Java language*—The Java language is the statically typed, object-oriented language that we lightly lampooned in the "About This Book" section. Hopefully, it's already very familiar to you. One very obvious point about the Java language is that it's human-readable (or it should be!).
- *The Java platform*—The platform is the software that provides a runtime environment. It's the JVM that links and executes your code as provided to it in the form of (not human-readable) class files. It doesn't directly interpret Java language source files, but instead requires them to be converted to class files first.

One of the big reasons for the success of Java as a software system is that it's a standard. This means that it has specifications that describe how it's supposed to work. Standardization allows different vendors and project groups to produce implementations that should all, in theory, work the same way. The specs don't make guarantees about how well different implementations will perform when handling the same task, but they can provide assurances about the correctness of the results.

There are a number of separate specs that govern the Java system—the most important are the Java Language Specification (JLS) and the JVM Specification (VMSpec). In Java 7, this separation is taken very seriously; in fact, the VMSpec no longer makes any reference whatsoever to the JLS. If you're thinking that this might be an indication of how seriously non-Java source languages are taken in Java 7, then well done, and stay tuned. We'll talk a lot more about the differences between these two specs later.

One obvious question, when you're faced with the described duality, is, "What's the link between them?" If they're now so separate in Java 7, how do they come together to make the familiar Java system?

The link between the language and platform is the shared definition of the class file format (the .class files). A serious study of the class file definition will reward you, and it's one of the ways a good Java programmer can start to become a great one. In figure 1.1 you can see the full process by which Java code is produced and used.

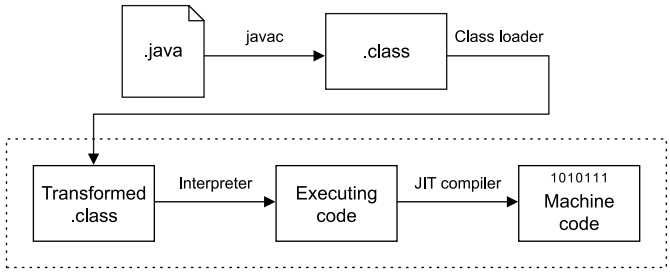


Figure 1.1 Java source code is transformed into .class files, then manipulated at load time before being JIT-compiled.

As you can see in the figure, Java code starts life as human-readable Java source, and it's then compiled by `javac` into a .class file. This is then loaded into a JVM. Note that it's very common for classes to be manipulated and altered during the loading process. Many of the most popular frameworks (especially those with "Enterprise" in their names) will transform classes as they're loaded.

Is Java a compiled or interpreted language?

The standard picture of Java is of a language that's compiled into .class files before being run on a JVM. If pressed, many developers can also explain that bytecode starts off by being interpreted by the JVM but will undergo just-in-time (JIT) compilation at some later point. Here, however, many people's understanding breaks down in a somewhat hazy conception of bytecode as basically being machine code for an imaginary or simplified CPU.

In fact, JVM bytecode is more like a halfway house between human-readable source and machine code. In the technical terms of compiler theory, bytecode is really a form of intermediate language (IL) rather than a true machine code. This means that the process of turning Java source into bytecode isn't really compilation in the sense that a C or C++ programmer would understand it, and javac isn't a compiler in the same sense as gcc is—it's really a class file generator for Java source. The real compiler in the Java ecosystem is the JIT compiler, as you can see in figure 1.1.

Some people describe the Java system as "dynamically compiled." This emphasizes that the compilation that matters is the JIT compilation at runtime, not the creation of the class file during the build process.

So, the real answer to, "Is Java compiled or interpreted?" is "Both."

With the distinction between language and platform hopefully now clearer, let's move on to talk about some of the visible changes in language syntax that have arrived with Java 7, starting with smaller syntax changes brought in with Project Coin.

1.2 Small is beautiful—Project Coin

Project Coin is an open source project that has been running as part of the Java 7 (and 8) effort since January 2009. In this section, we're going to explain how features

get chosen and how the language evolution process works by using the small changes of Project Coin as a case study.

Naming Project Coin

The aim of Project Coin was to come up with small changes to the Java language. The name is a piece of wordplay—small change comes as coins, and "to coin a phrase" means to add a new expression to our language.

These types of word games, whimsy, and the inevitable terrible puns are to be found everywhere in technical culture. You may just as well get used to them.

We think it's important to explain the "why" of language change as well as the "what." During the development of Java 7, there was a lot of interest around new language features, but the community didn't always understand how much work is required to get changes fully engineered and ready for prime time. We hope to shed a bit of light on this area, and hopefully dispel a few myths. But if you're not very interested in how Java evolves, feel free to skip ahead to section 1.3 and jump right into the language changes.

There is an effort curve involved in changing the Java language—some possible implementations require less engineering effort than others. In figure 1.2 we've tried to represent the different routes and show the relative effort required for each, in a complexity scale of increasing effort.

In general, it's better to take the route that requires the least effort. This means that if it's possible to implement a new feature as a library, you generally should. But not all features are easy, or even possible, to implement in a library or an IDE capability. Some features have to be implemented deeper inside the platform.

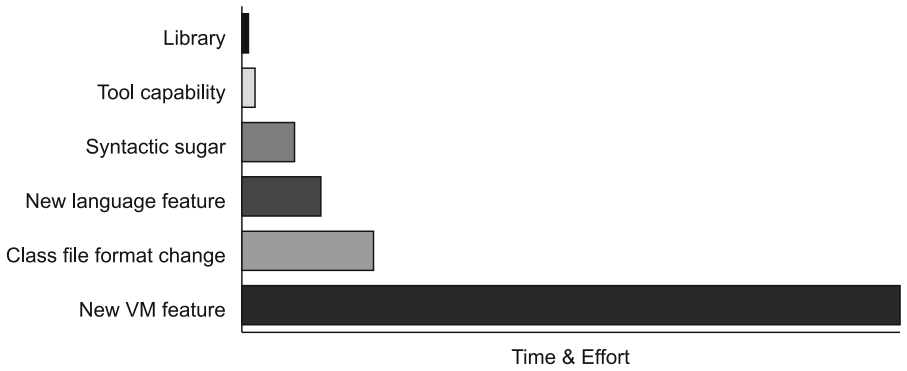


Figure 1.2 The relative effort involved in implementing new functionality in different ways

Here's how some (mostly Java 7) features fit into our complexity scale for new language features:

- *Syntactic sugar*—Underscores in numbers (Java 7)
- *Small new language feature*—try-with-resources (Java 7)
- *Class file format change*—Annotations (Java 5)
- *New JVM feature*—`invokedynamic` (Java 7)

Syntactic sugar

A phrase that's sometimes used to describe a language feature is "syntactic sugar." This means that the syntax form is redundant—it already exists in the language—but the syntactic sugar form is provided because it's easier for humans to work with.

As a rule of thumb, a feature referred to as syntactic sugar is removed from the compiler's representation of the program early on in the compilation process—it's said to have been "desugared" into the basic representation of the same feature.

This makes syntactic sugar changes to a language easier to implement because they usually involve a relatively small amount of work, and only involve changes to the compiler (javac in the case of Java).

Project Coin (and the rest of this chapter) is all about changes that are somewhere in the range from syntactic sugar to small new language features.

The initial period for suggestions for Project Coin changes ran on the coin-dev mailing list from February to March 2009 and saw almost 70 proposals submitted, representing a huge range of possible enhancements. The suggestions even included a joke proposal for adding multiline strings in the style of lolcat captions (superimposed captions on pictures of cats that are either funny or irritating, depending on your viewpoint—http://icanhascheezburger.com/).

The Project Coin proposals were judged under a fairly simple set of rules. Contributors needed to do three things:

- Submit a detailed proposal form describing their change (which should fundamentally be a Java language change, rather than a virtual machine change)
- Discuss their proposal openly on a mailing list and field constructive criticism from the other participants
- Be prepared to produce a prototype set of patches that could implement their change

Project Coin provides a good example of how the language and platform may evolve in the future, with changes discussed openly, early prototyping of features, and calls for public participation.

One question that might well be asked at this point is, "What constitutes a small change to the spec?" One of the changes we'll discuss in a minute adds a single word—`"String"`—to section 14.11 of the JLS. You can't really get much smaller than that as a change, and yet even this change touches several other aspects of the spec.

Java 7 is the first version developed in an open source manner

Java was not always an open source language, but following an announcement at the JavaOne conference in 2006, the source code for Java itself (minus a few bits that Sun didn't own the source for) was released under the GPLv2 license. This was around the time of the release of Java 6, so Java 7 is the first version of Java to be developed under an open source software (OSS) license. The primary focus for open source development of the Java platform is the OpenJDK project.

Mailing lists such as coin-dev, lambda-dev, and mlvm-dev have been major forums for discussing possible future features, allowing developers from the wider community to participate in the process of producing Java 7. In fact, we help lead the "Adopt OpenJDK" program to guide developers new to the OpenJDK, helping improve Java itself! See http://java.net/projects/jugs/pages/AdoptOpenJDK if you'd like to join us.

Any alteration produces consequences, and these have to be chased through the entire design of the language.

The full set of actions that that must be performed (or at least investigated) for *any* change is as follows:

- Update the JLS
- Implement a prototype in the source compiler
- Add library support essential for the change
- Write tests and examples
- Update documentation

In addition, if the change touches the VM or platform aspects:

- Update the VMSpec
- Implement the VM changes
- Add support in the class file and VM tools
- Consider the impact on reflection
- Consider the impact on serialization
- Think about any impacts on native code components, such as Java Native Interface (JNI).

This isn't a small amount of work, and that's after the impact of the change across the whole language spec has been considered!

An area of particular hairiness, when it comes to making changes, is the type system. That isn't because Java's type system is bad. Instead, languages with rich static type systems are likely to have a lot of possible interaction points between different bits of those type systems. Making changes to them is prone to creating unexpected surprises.

Project Coin took the very sensible route of suggesting to contributors that they mostly stay away from the type system when proposing changes. Given the amount of work that has gone into even the smallest of these small changes, this has proved a pragmatic approach.

With that bit of the background on Project Coin covered, it's time to start looking at the features chosen for inclusion.

1.3 The changes in Project Coin

Project Coin brought six main new features to Java 7. These are `Strings` in `switch`, new numeric literal forms, improved exception handling, try-with-resources, diamond syntax, and fixes for varargs warnings.

We're going to talk in some detail about these changes from Project Coin—we'll discuss the syntax and the meaning of the new features, and also try to explain the motivations behind the features whenever possible. We won't resort to the full formal details of the proposals, but all that material is available from the archives of the coin-dev mailing list, so if you're a budding language designer, you can read the full proposals and discussion there.

Without further ado, let's kick off with our very first new Java 7 feature—`String` values in a `switch` statement.

1.3.1 Strings in switch

The Java `switch` statement allows you to write an efficient multiple-branch statement without lots and lots of ugly nested ifs—like this:

```
public void printDay(int dayOfWeek) {
  switch (dayOfWeek) {
    case 0: System.out.println("Sunday"); break;
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
    case 4: System.out.println("Thursday"); break;
    case 5: System.out.println("Friday"); break;
    case 6: System.out.println("Saturday"); break;
    default: System.err.println("Error!"); break;
  }
}
```

In Java 6 and before, the values for the cases could only be constants of type `byte`, `char`, `short`, `int` (or, technically, their reference-type equivalents `Byte`, `Character`, `Short`, `Integer`) or `enum` constants. With Java 7, the spec has been extended to allow for the `String` type to be used as well. They're constants after all.

```
public void printDay(String dayOfWeek) {
  switch (dayOfWeek) {
    case "Sunday": System.out.println("Dimanche"); break;
    case "Monday": System.out.println("Lundi"); break;
    case "Tuesday": System.out.println("Mardi"); break;
    case "Wednesday": System.out.println("Mercredi"); break;
    case "Thursday": System.out.println("Jeudi"); break;
    case "Friday": System.out.println("Vendredi"); break;
    case "Saturday": System.out.println("Samedi"); break;
    default: System.out.println("Error: '"+ dayOfWeek
    ➥ +"' is not a day of the week"); break;
  }
}
```

In all other respects, the `switch` statement remains the same. Like many Project Coin enhancements, this is really a very simple change to make life in Java 7 a little bit easier.

1.3.2 Enhanced syntax for numeric literals

There were several separate proposals around new syntax for the integral types. The following aspects were eventually chosen:

- Numeric constants (that is, one of the integer primitive types) may now be expressed as binary literals.
- Underscores may be used in integer constants to improve readability

Neither of these is, at first sight, particularly earth-shattering, but both have been minor annoyances to Java programmers.

These are both of special interest to the low-level programmer—the sort of person who works with raw network protocols, encryption, or other pursuits, where a certain amount of bit twiddling is involved. Let's begin with a look at binary literals.

BINARY LITERALS

Before Java 7, if you wanted to manipulate a binary value, you'd have had to either engage in awkward (and error-prone) base conversion or utilize `parseX` methods. For example, if you wanted to ensure that an `int x` represented the bit pattern for the decimal value 102 correctly, you'd write an expression like:

```
int x = Integer.parseInt("1100110", 2);
```

This is a lot of code just to ensure that `x` ends up with the correct bit pattern. There's worse to come though. Despite looking fine, there are a number of problems with this approach:

- It's really verbose.
- There is a performance hit for that method call.
- You'd have to know about the two-argument form of `parseInt()`.
- You need to remember the details of how `parseInt()` behaves when it has two arguments.
- It makes life hard for the JIT compiler.
- It represents a compile-time constant as a runtime expression, which means the constant can't be used as a value in a `switch` statement.
- It will give you a `RuntimeException` (but no compile-time exception) if you have a typo in the binary value.

Fortunately, with the advent of Java 7, we can now write this:

```
int x = 0b1100110;
```

No one's saying that this is doing anything that couldn't be done before, but it has none of the problems we listed.

If you've got a reason to work with binary, you'll be glad to have this small feature. For example, when doing low-level handling of bytes, you can now have bit patterns as binary constants in `switch` statements.

Another small, yet useful, new feature for representing groups of bits or other long numeric representations is underscores in numbers.

UNDERSCORES IN NUMBERS

You've probably noticed that the human mind is radically different from a computer's CPU. One specific example of this is in the way that our minds handle numbers. Humans aren't, in general, very comfortable with long strings of numbers. That's one reason we invented hexadecimal—because our minds find it easier to deal with shorter strings that contain more information, rather than long strings containing not much information per character.

That is, we find 1c372ba3 easier to deal with than 00011100001101110010101110100011, even though a CPU would only ever see the second form. One way that we humans deal with long strings of numbers is to break them up. A U.S. phone number is usually represented like this: 404-555-0122.

> **NOTE** If you're like the (European) authors and have ever wondered why US phone numbers in films or books always start with 555, it's because the numbers 555-01xx are reserved for fictional use—precisely to prevent real people getting calls from people who take their Hollywood movies a little too seriously.

Other long strings of numbers have separators too:

- $100,000,000 (large sums of money)
- 08-92-96 (UK banking sort codes)

Unfortunately, both the comma (,) and hyphen (-) have too many possible meanings within the realm of handling numbers in programming, so we can't use either as a separator. Instead, the Project Coin proposal borrowed an idea from Ruby, and introduced the underscore (_) as a separator. Note that this is just a bit of easy-on-the-eyes compile-time syntax. The compiler strips out those underscores and stores the usual digits.

This means that you can write 100_000_000 and hopefully not confuse it with 10_000_000, whereas 100000000 is easily confused with 10000000. Let's look at a couple of examples, at least one of which should be familiar:

```
long anotherLong = 2_147_483_648L;
int bitPattern = 0b0001_1100__0011_0111__0010_1011__1010_0011;
```

Notice how much easier it is to read the value being assigned to `anotherLong`.

> **WARNING** In Java, it's still legal to use the lowercase *l* character to denote a `long`. For example `10101001`. Make sure you always use an uppercase *L* so that maintainers don't get confused between the number *1* and the letter *l*: `1010100L` is much clearer!

By now, you should be convinced of the benefit of these tweaks to the handling of integers, so let's move on to looking at Java 7's improved exception handling.

1.3.3 Improved exception handling

There are two parts to this improvement—multicatch and final rethrow. To see why they're a help, consider the following Java 6 code, which tries to find, open, and parse a config file and handle a number of different possible exceptions.

Listing 1.1 Handling several different exceptions in Java 6

```
public Configuration getConfig(String fileName) {
  Configuration cfg = null;
  try {
    String fileText = getFile(fileName);
    cfg = verifyConfig(parseConfig(fileText));
  } catch (FileNotFoundException fnfx) {
    System.err.println("Config file '" + fileName + "' is missing");
  } catch (IOException e) {
    System.err.println("Error while processing file '" + fileName + "'");
  } catch (ConfigurationException e) {
    System.err.println("Config file '" + fileName + "' is not consistent");
  } catch (ParseException e) {
    System.err.println("Config file '" + fileName + "' is malformed");
  }

  return cfg;
}
```

This method can encounter a number of different exceptional conditions:

- The config file may not exist.
- The config file may disappear while you're trying to read from it.
- The config file may be malformed syntactically.
- The config file may have invalid information in it.

These conditions fit into two distinct functional groups. Either the file is missing or bad in some way, or the file is present and correct but couldn't be retrieved properly (perhaps because of a hardware failure or network outage).

It would be nice to compress this down to just these two cases, and handle all the "file is missing or bad in some way" exceptions in one `catch` clause. Java 7 allows you to do this.

Listing 1.2 Handling several different exceptions in Java 7

```
public Configuration getConfig(String fileName) {
  Configuration cfg = null;
  try {
    String fileText = getFile(fileName);
    cfg = verifyConfig(parseConfig(fileText));
  } catch (FileNotFoundException|ParseException|ConfigurationException e) {
    System.err.println("Config file '" + fileName +
                       "' is missing or malformed");
  } catch (IOException iox) {
    System.err.println("Error while processing file '" + fileName + "'");
  }
```

```
  return cfg;
}
```

The exception `e` has a type that isn't precisely knowable at compile time. This means that it has to be handled in the `catch` block as the common supertype of the exceptions that it *could* be (which will often be `Exception` or `Throwable`, in practice).

An additional bit of new syntax helps with rethrowing exceptions. In many cases, developers may want to manipulate a thrown exception before rethrowing it. The problem is that in previous versions of Java you'll often see code like this:

```
try {
  doSomethingWhichMightThrowIOException();
  doSomethingElseWhichMightThrowSQLException();
} catch (Exception e) {
  ...
  throw e;
}
```

This forces you to declare the exception signature of this code as `Exception`—the real dynamic type of the exception has been swallowed.

Nevertheless, it's relatively easy to see that the exception can only be an `IOException` or a `SQLException`, and if you can see it, so can the compiler. This snippet changes a single word change to use the Java 7 syntax:

```
try {
  doSomethingWhichMightThrowIOException();
  doSomethingElseWhichMightThrowSQLException();
} catch (final Exception e) {
  ...
  throw e;
}
```

The appearance of the `final` keyword indicates that the type that's actually thrown is the runtime type of the exception that was encountered—in this example, that would be either `IOException` or `SQLException`. This is referred to as *final rethrow*, and it can protect against throwing an overly general type, which then has to be caught by a very general `catch` in a higher scope.

The `final` keyword is optional in the previous example, but in practice, we've found that it helps to use it while adjusting to the new semantics of catch and rethrow.

In addition to these general improvements in exception handling, the specific case of resource management has been improved in Java 7, so that's where we'll turn next.

1.3.4 Try-with-resources (TWR)

This change is easy to explain, but it has proved to have hidden subtleties, which made it much less easy to implement than originally hoped. The basic idea is to allow a resource (for example, a file or something a bit like one) to be scoped to a block in such a way that the resource is automatically closed when control exits the block.

This is an important change, for the simple reason that virtually no one gets manual resource closing 100 percent right. Until recently, even the reference how-tos

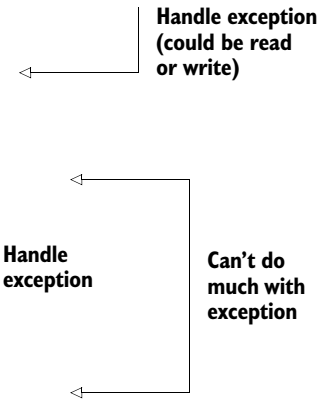
from Sun were wrong. The proposal submitted to Project Coin for this change includes the astounding claim that two-thirds of the uses of `close()` in the JDK had bugs in them!

Fortunately, compilers can be made to produce exactly the sort of pedantic, boilerplate code that humans so often get wrong, and that's the approach taken by this change.

This is a big help in writing error-free code. To see just how helpful, consider how you'd write a block of code that reads from a stream coming from a URL (`url`) and writes to a file (`out`) with Java 6. Here's one possible solution.

Listing 1.3 Java 6 syntax for resource management

```
InputStream is = null;
try {
  is = url.openStream();
  OutputStream out = new FileOutputStream(file);
  try {
    byte[] buf = new byte[4096];
    int len;
    while ((len = is.read(buf)) >= 0)
      out.write(buf, 0, len);
  } catch (IOException iox) {
  } finally {
    try {
      out.close();
    } catch (IOException closeOutx) {
    }
  }
} catch (FileNotFoundException fnfx) {
} catch (IOException openx) {
} finally {
  try {
    if (is != null) is.close();
  } catch (IOException closeInx) {
  }
}
```



How close did you get? The key point here is that when handling external resources, Murphy's Law applies—anything can go wrong at any time:

- The `InputStream` can fail to open from the URL, to read from it, or to close properly.
- The `File` corresponding to the `OutputStream` can fail to open, to write to it, or to close properly.
- A problem can arise from some combination of more than one factor.

This last possibility is where a lot of the headaches come from—a combination of exceptions is very difficult to deal with well.

This is the main reason for preferring the new syntax—it's much less error-prone. The compiler isn't susceptible to the mistakes that every developer will make when trying to write this type of code manually.

Let's look at the Java 7 code for performing the same task as listing 1.3. As before, `url` is a URL object that points at the entity you want to download, and `file` is a `File` object where you want to save what you're downloading. Here's what this looks like in Java 7.

Listing 1.4 Java 7 syntax for resource management

```
try (OutputStream out = new FileOutputStream(file);
     InputStream is = url.openStream() ) {
  byte[] buf = new byte[4096];
  int len;
  while ((len = is.read(buf)) > 0) {
    out.write(buf, 0, len);
  }
}
```

This basic form shows the new syntax for a block with automatic management—the `try` with the resource in round brackets. For C# programmers, this is probably a bit reminiscent of a `using` clause, and that's a good conceptual starting point when working with this new feature. The resources are used by the block, and they're automatically disposed of when you're done with them.

You still have to be careful with try-with-resources, as there are cases where a resource might still not be closed. For example, the following code would not close its `FileInputStream` properly if there was an error creating the `ObjectInputStream` from the file (`someFile.bin`).

```
try ( ObjectInputStream in = new ObjectInputStream(new
      FileInputStream("someFile.bin")) ) {
 ...
}
```

Let's assume that the file (`someFile.bin`) exists, but it might not be an `ObjectInput` file, so the file might not open correctly. Therefore, the `ObjectInputStream` wouldn't be constructed and the `FileInputStream` wouldn't be closed!

The correct way to ensure that try-with-resources always works for you is to split the resources into separate variables.

```
try ( FileInputStream fin = new FileInputStream("someFile.bin");
         ObjectInputStream in = new ObjectInputStream(fin) ) {
 ...
}
```

One other aspect of TWR is the appearance of enhanced stack traces and suppressed exceptions. Prior to Java 7, exception information could be swallowed when handling resources. This possibility also exists with TWR, so the stack traces have been enhanced to allow you to see the type information of exceptions that would otherwise be lost.

For example, consider this snippet, in which a null `InputStream` is returned from a method:

```
try(InputStream i = getNullStream()) {
  i.available();
}
```

This will give rise to an enhanced stack trace, in which the suppressed `NullPointerException` (NPE for short) can be seen:

```
Exception in thread "main" java.lang.NullPointerException
  at wgjd.ch01.ScratchSuprExcep.run(ScratchSuprExcep.java:23)
  at wgjd.ch01.ScratchSuprExcep.main(ScratchSuprExcep.java:39)
  Suppressed: java.lang.NullPointerException
  at wgjd.ch01.ScratchSuprExcep.run(ScratchSuprExcep.java:24)
    1 more
```

TWR and AutoCloseable

Under the hood, the TWR feature is achieved by the introduction of a new interface, called `AutoCloseable`, which a class must implement in order to be able to appear as a resource in the new TWR `try` clause. Many of the Java 7 platform classes have been converted to implement `AutoCloseable` (and it has been made a superinterface of `Closeable`), but you should be aware that not every aspect of the platform has yet adopted this new technology. It's included as part of JDBC 4.1, though.

For your own code, you should definitely use TWR whenever you need to work with resources. It will help you avoid bugs in your exception handling.

We encourage you to use try-with-resources as soon as you're able, to eliminate unnecessary bugs from your codebase.

1.3.5 Diamond syntax

Java 7 also introduces a change that means less typing for you when dealing with generics. One of the problems with generics is that the definitions and setup of instances can be really verbose. Let's suppose that you have some users, whom you identify by `userid` (which is an integer), and each user has one or more lookup tables specific to that user. What would that look like in code?

```
Map<Integer, Map<String, String>> usersLists =
    new HashMap<Integer, Map<String, String>>();
```

That's quite a mouthful, and almost half of it is duplicated characters. Wouldn't it be better if you could write something like this,

```
Map<Integer, Map<String, String>> usersLists = new HashMap<>();
```

and have the compiler work out the type information on the right side? Thanks to the magic of Project Coin, you can. In Java 7, the shortened form for declarations like that is entirely legal. It's backwards compatible as well, so when you find yourself revisiting old code, you can cut the older, more verbose declaration and start using the new type-inferred syntax to save a few pixels.

We should point out that the compiler is using a new form of type inference for this feature. It's working out the correct type for the expression on the right side, and isn't just substituting in the text that defines the full type.

> **The "diamond syntax" name**
> This form is called "diamond syntax" because, well, the shortened type information looks like a diamond. The proper name in the proposal is "Improved Type Inference for Generic Instance Creation," which is a real mouthful and has ITIGIC as an acronym, which sounds stupid, so diamond syntax it is.

The new diamond syntax will certainly save your fingers from some typing. The last Project Coin feature we'll explore is the removal of a warning when you're using varargs.

1.3.6 Simplified varargs method invocation

This is one of the simplest changes of all—it moves a warning about type information for a very specific case where varargs combines with generics in a method signature.

Put another way, unless you're in the habit of writing code that takes as arguments a variable number of references of type `T` and does something to make a collection out of them, you can move on to the next section. On the other hand, if this bit of code looks like something you might write, you should read on:

```
public static <T> Collection<T> doSomething(T... entries) {
  ...
}
```

Still here? Good. So what's this all about?

As you probably know, a varargs method is one that takes a variable number of parameters (all of the same type) at the end of the argument list. What you may not know is how varargs is implemented; basically, all of the variable parameters at the end are put into an array (which the compiler automatically creates for you) and they're passed as a single parameter.

This is all well and good, but here we run into one of the admitted weaknesses of Java's generics—you aren't normally allowed to create an array of a known generic type. For example, this won't compile:

```
HashMap<String, String>[] arrayHm = new HashMap<>[2];
```

You can't make arrays of a specified generic type. Instead, you have to do this:

```
HashMap<String, String>[] warnHm = new HashMap[2];
```

This gives a warning that has to be ignored. Notice that you can define the type of `warnHm` to be an array of `HashMap<String, String>`—you just can't create any instances of that type, and instead have to hold your nose (or at least, suppress the warning) and force an instance of the raw type (which is array of `HashMap`) into `warnHm`.

These two features—varargs methods working on compiler-generated arrays, and arrays of known generic types not being an instantiable type—come together to cause a slight headache. Consider this bit of code:

```
HashMap<String, String> hm1 = new HashMap<>();
HashMap<String, String> hm2 = new HashMap<>();

Collection<HashMap<String, String>> coll = doSomething(hm1, hm2);
```

The compiler will attempt to create an array to contain `hm1` and `hm2`, but the type of the array should strictly be one of the forbidden array types. Faced with this dilemma, the compiler cheats and breaks its own rule about the forbidden array of generic type. It creates the array instance, but grumbles about it, producing a compiler warning that mutters darkly about "unchecked or unsafe operations."

From the point of view of the type system, this is fair enough. But the poor developer just wanted to use what seemed like a perfectly sensible API, and there are scary-sounding warnings for no adequately explained reason.

WHERE DID THE WARNING GO IN JAVA 7?

The new feature in Java 7 changes the emphasis of the warning. After all, there is a potential for violating type safety in these types of constructions, and *somebody* had better be informed about them. There's not much that the users of these types of APIs can really do, though. Either the code inside `doSomething()` is evil and violates type safety, or it doesn't. In any case, it's out of the API user's hands.

The person who should really be warned about this issue is the person who wrote `doSomething()`—the API producer, rather than the consumer. So that's where the warning goes—it's moved from where the API is used to where the API was defined.

The warning once was triggered when code that used the API was compiled. Instead, it's now triggered when an API that has the potential to trigger this kind of type safety violation is written. The compiler warns the coder implementing the API, and it's up to that developer to pay proper attention to the type system.

To make things easier for API developers, Java 7 also provides a new annotation type, `java.lang.SafeVarargs`. This can be applied to an API method (or constructor) that would otherwise produce a warning of the type discussed. By annotating the method with `@SafeVarargs`, the developer essentially asserts that the method doesn't perform any unsafe operations. In this case, the compiler will suppress the warning.

CHANGES TO THE TYPE SYSTEM

That's an awful lot of words to describe a very small change—moving a warning from one place to another is hardly a game-changing language feature, but it does serve to illustrate one very important point. Earlier in this chapter we mentioned that Project Coin encouraged contributors to mostly stay away from the type system when proposing changes. This example shows how much care is needed when figuring out how different features of the type system interact, and how that interaction will alter when a change to the language is implemented. This isn't even a particularly complex change—larger changes would be far, far more involved, with potentially dozens of subtle ramifications.

This final example illustrates how intricate the effect of small changes can be. Although they represent mostly small syntactic changes, they can have a positive impact on your code that is out of proportion with the size of the changes. Once you've started using them, you'll likely find that they offer real benefit to your programs.

1.4 Summary

Making changes to the language itself is hard. It's always easier to implement new features in a library (if you can—not everything can be implemented without a language change). The challenges involved can cause language designers to make smaller, and more conservative, changes than they might otherwise wish.

Now, it's time to move on to some of the bigger pieces that make up the release, starting with a look at how some of the core libraries have changed in Java 7. Our next stop is the I/O libraries, which have been considerably revamped. It will be helpful to have a grasp of how previous Java versions coped with I/O, because the Java 7 classes (sometimes called NIO.2) build upon the existing framework.

If you want to see some more examples of the TWR syntax in action, or want to learn about the new, high-performance asynchronous I/O classes, then the next chapter has all the answers.

JAVA

The Well-Grounded Java Developer

B. J. Evans • M. Verburg

This book takes a fresh and practical look at new Java 7 features, new JVM languages, and the array of supporting technologies you need for the next generation of Java-based software.

You'll start with thorough coverage of Java 7 features like try-with-resources and NIO.2. You'll then explore a cross-section of emerging JVM-based languages, including Groovy, Scala, and Clojure. You will find clear examples that are practical and that help you dig into dozens of valuable development techniques showcasing modern approaches to the dev process, concurrency, performance, and much more.

What's Inside

- New Java 7 features
- Tutorials on Groovy, Scala, and Clojure
- Discovering multicore processing and concurrency
- Functional programming with new JVM languages
- Modern approaches to testing, build, and CI

Written for readers familiar with Java. No experience with Java 7 or new JVM languages required.

Ben Evans is the CEO of a Java performance firm and a member of the Java Community Process Executive Committee. **Martijn Verburg** is the CTO of a Java performance firm, coleader of the London JUG, and a popular conference speaker.

"How to become a well-grounded Java developer—and how to *stay* that way."
—From the Foreword by Dr. Heinz Kabutz
The Java Specialists' Newsletter

"At the cutting edge of Java development ... learn to speak Java 7 and next-gen languages."
—Paul Benedict
Corporate Personnel & Associates

"Buy this book for what's new in Java 7. Keep it open for lessons in expert Java."
—Stephen Harrison, PhD
FirstFuel Software

"A great collection of knowledge on the JVM platform."
—Rick Wagner, Red Hat

To download their free eBook in PDF, ePub and Kindle formats, owners of this book should visit manning.com/TheWell-GroundedJavaDeveloper

ISBN 13: 978-1-617290-06-0
ISBN 10: 1-617290-06-8
54999
9 781617 290060

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]