

Vital techniques of Java 7 and polyglot programming

The Well-Grounded
Java
Developer

Benjamin J. Evans
Martijn Verburg



 MANNING



**MEAP Edition
Manning Early Access Program
The Well-Grounded Java Developer version 7**

Copyright 2012 Manning Publications
For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part 1: Developing with Java 7

- 1 Introducing Java 7
- 2 New I/O

Part 2: Vital techniques

- 3 Dependency injection
- 4 Modern concurrency
- 5 Classfiles and bytecode
- 6 Understanding performance tuning

Part 3: Polyglot programming on the JVM

- 7 Alternative JVM languages
- 8 Groovy: Java's dynamic friend
- 9 Scala: power and conciseness
- 10 Clojure: safer programming

Part 4: Crafting the polyglot project

- 11 Test-driven development
- 12 Build and continuous integration
- 13 Rapid web development
- 14 Beyond ground level

Appendixes

- A For more information
- B Spring configs
- C Glob syntax
- D Installing alternative JVM languages

1

Introducing Java 7

This chapter covers:

- Java as a Platform and a Language
- Small yet Powerful Syntax Changes
- Try-with-resources
- 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 we finish our journey, you'll have taken your first steps into a larger world – a world of new features, of software craftsmanship and other languages on the JVM.

We're going to warm you up with a gentle introduction to Java 7 - but one which still acquaints you with powerful features. We'll showcase Project Coin – a collection of small yet effective new features. 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 your code which 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 underway by discussing a critically important duality which lies at the heart of modern Java. This is a point that we'll come back to again throughout the book, so it's an absolutely essential one to grasp.

1.1 The Language and the Platform

The critical concept that we're kicking off with is the distinction between the Java language and the Java platform. It's perhaps a bit surprising that there quite a few Java programmers who aren't that clear on the differences between the two, or which of them provide the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=725>

programming features that their code uses. Let's make that distinction clear right now, as it is at the heart of a lot of the topics in this book.

The Java Language

The Java language is the statically typed, object-oriented language that we lightly lampooned in "About This Book" - hopefully, it's already very familiar to you. One very obvious point about the Java language is that it's human-readable (or at least, should be).

The Java Platform

The platform is the software that provides a runtime environment - i.e. the Java virtual machine (JVM) which links and executes your code as provided to it in the form of (not human-readable) class files. It does not 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 is a standard. This means that it has specifications which describe how it is supposed to work. Standardization allows different vendors and project groups to produce implementations which should all, in theory, work the same way. The specs do not, of course, make guarantees about how well different implementations will perform when handling the same task - but they can provide assurances about the correctness of results.

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

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

The link is the shared definition of the classfile format (i.e. the .class files). A serious study of it will reward you, and it's one of the ways a good Java programmer can start to turn herself into 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 and then manipulated at load time before being JIT-compiled

As you can see in Figure 1.1, Java code starts life as human-readable Java source, and is then compiled by `javac` into a .class file – this is then loaded into a JVM. Note that it is 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 are loaded.

Is Java a compiled or interpreted language?

The standard picture of Java is of a language that is compiled into .class files before being run on a JVM. If pressed, many developers can also relate that bytecode starts off by being interpreted by the JVM but will undergo Just-In-Time compilation at some later point. Here, however, many people’s understanding breaks down, with 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 being 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 is not really compilation in the sense that a C or C++ programmer would understand it - and `javac` is not 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 “dynamic compiled”. This emphasizes that the compilation which matters is the JIT compilation at runtime, not the creation of the class file during the build process. So perhaps the real answer to “Is Java compiled or interpreted?” is actually “Yes.”

With the distinction between language and platform hopefully now clearer, we’re going to move on to talk about some of the visible changes in language syntax that have arrived with 7 - starting with some of the smaller syntax changes brought in with Project Coin.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=725>

1.2 *Small Is Beautiful - Project Coin*

Project Coin is an open-source project which has been running as part of the Java 7 effort since January 2009 – the aim of which was to come up with small changes to the Java language. The name is a piece of word-play - small change comes as coins and “to coin a phrase” means to add a new expression to our language¹.

There is an effort curve involved in changing the language. As you can see in Figure 1.2, if it's possible to implement in a library, then generally you should. This section is all about the changes which are somewhere in the range of being syntactic sugar to simple new language feature.

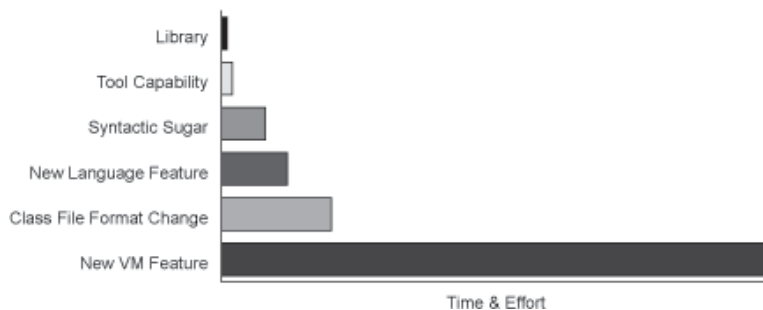


Figure 1.2 The relative effort involved in implementing new functionality in various different ways

The initial period for suggestions for Project Coin changes ran from February to March 2009, on the coin-dev mailing list 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 (the pictures of cats with captions which are either funny or irritating, depending on your point of view - <http://icanhascheezburger.com/> if you've never come across them).

The proposals were judged under a fairly simple set of rules. Contributors needed to:

- 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 which could implement their change.

¹ These types of word games, whimsy and the inevitable terrible puns are to be found everywhere in technical culture. It's probably as well to get used to them.

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.

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

One question which might well be asked at this point is - "What constitutes a small change to the spec?"

Well, one of the changes we'll discuss in a minute adds just 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 tiny change touches several other aspects of the spec - any alteration produces consequences and these have to be chased through the entire design of the language. The full set of actions that would be required (or at least investigated) for *any* change is:

- 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 this change touched the VM or platform aspects:

- Update the VMSpec
- Implement the VM changes
- Add support in the classfile and VM tools
- Consider the impact on reflection
- and on serialization
- and think about JNI

This is not 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=725>

likely to have a lot of possible interaction points between different bits of those type systems. Thus 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 which has gone into even the smallest of these small changes, this seems to have proved a pragmatic approach.

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

1.3 The Changes in Project Coin

We're going to talk in some detail about some of the proposals in Project Coin – we'll discuss the syntax and the meaning of the new features, and also some of the “why” - that is try to explain the motivations behind the feature whenever possible, without resorting to the full formal details of the proposals – all that material is available from the archives of the coin-dev mailing list though, 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 can only be constants of type byte, char, short, int (or, technically, their reference-type equivalents Byte, Character, Short, Integer) or enum constants. With 7, the spec has been extended to allow for Strings 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;
    }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=725>

```

        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 aspects eventually chosen were:

- Numeric constants (i.e. one of the integer primitive types) expressed as binary
- A specific suffix to denote that an integer constant has type short or byte
- Use of underscores in integer constants for readability

None of these is at first sight particularly earth-shattering, but all have in their own individual way been a minor annoyance to the Java programmer.

The first two are of special interest to the low-level programmer – the sort of person who works with raw network protocols, encryption or other pursuits where he or she may have to indulge in a certain amount of bit twiddling. So let's look at those first:

BINARY LITERALS

Before Java 7, if you'd wanted to manipulate a binary value, you'd either have had to engage in awkward (and error-prone) base-conversion, or write an expression like:

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

This is a lot of characters of typing just to ensure that x ends up with that bit pattern (which is 102 in decimal, by the way). There's worse to come though – despite looking fine at first glance, there are a number of problems - it:

- is really verbose
- has a performance hit for that method call
- means you'd have to know about the 2-argument form of parseInt()
- requires you to remember the detail of how parseInt() behaves when it has 2 args
- makes life hard for the JIT compiler
- is representing a compile-time constant as a runtime expression (so can't be used as a value in a switch statement)
- 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:

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

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

So, 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.

SHORT AND BYTE LITERALS

Java has several integer-like types, including short and byte, as well as the perhaps more familiar int and long. All of Java's integral types are signed – meaning that they can represent both positive and negative numbers (there's also the char data type which if squinted at in the right light can look like an unsigned 16-bit type, but which is also quite different from an integer-like type in quite a few ways).

For people coming from other languages, notably C/C++, this lack of unsigned types is a bit weird, to say the least. Having a signed byte type was seen as a major peculiarity (or evidence of lack of maturity at the very least) during the early days of Java. James Gosling, the father of Java, had this to say on the subject:

Quiz any C developer about unsigned, and pretty soon you discover that almost no C developers actually understand what goes on with unsigned, what unsigned arithmetic is.

This seems a little harsh. C has been around since the 70s – so you'd hope that something as simple as an unsigned integer type was pretty well, understood, right? So let's take a quick look at the part of the C spec (K&R, section A6) about how we turn an integer into an unsigned integer:

Integral conversions: any integer is converted to a given unsigned type by finding the smallest non negative value that is congruent to that integer, modulo one more than the largest value that can be represented in the unsigned type.

If you think about what code you'd need to write to implement this piece of spec then you'll immediately see that the whole concept of unsigned types is not as straightforward as it might have first appeared.

All of a sudden Java's slightly funny-looking signed bytes don't seem so bad – and certainly look better than dealing with all the corner cases that unsigned arithmetic has - although there's still a nagging sense that writing either:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=725>

```
byte allOnes = -1;
```

or:

```
byte allOnes = (byte)0xFF;
```

to get a byte composed of all 1s is somehow wrong. As a partial solution, if we can't have unsigned types, we could have unsigned constant values. These could be automatically converted to the correct (signed) values when they're loaded into variables.

This is, of course, exactly the sort of small change that Project Coin was invented for, and so in Java 7 you can write:

```
byte allOnes = 255y;
short myShort = 64000s;
System.out.println("allOnes = "+ allOnes);
```

This prints out -1 (as this is still the signed value that's loaded into allOnes). Hopefully this looks like the corresponding form for long, which should be familiar:

```
long myLong = 2147483648L;
```

This assigns the value 2 Gigabytes to myLong – a value which is of course too big to fit into an int.

Please note that this new syntax doesn't make byte or short into unsigned types – it just provides a convenient syntax to assign to them as though they were. Underneath, they're still signed types, which is why the print statement outputs -1.

Signed Arithmetic in Java

If you're wondering why a byte with all eight of its bits set to 1 represents -1, then this box is for you. A byte can represent 256 distinct numbers. If we want to represent both positive and negative numbers then it makes sense to have roughly equal numbers of each being representable. A byte with all bits zero should obviously represent 0, and a byte 00000001 should represent 1. Continuing that reasoning, 01111111 is 127. The question is what should 10000000 represent? It would be 128 in an unsigned representation, but what should it be for signed? The answer is that we want arithmetic in this representation to be as cheap as possible. In particular, addition should still "work". Adding 00000001 to 11111111 will give a byte of all zeros, plus an overflow. So that tells us that in this representation 11111111 is -1 (Java ignores the carry overflow for byte addition). So 10000000 is -128 (because it gives -1 when added to 127) and 10000001 is -127. This representation is called Two's Complement.

UNDERScores IN NUMBERS

You've probably noticed that the human mind is really quite 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 than 00011100001101110010101110100011 to deal with, even though a CPU would really only ever see the second form. One way that we humans deal with long strings of numbers is to break them up. A US phone number is usually represented like this:

```
404-555-0122
```

(By the way, if you've ever wondered why US phone numbers in films or books always start 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 ',' and '-' have too many possible meanings within the realm of handling numbers while programming, so we can't use either of those 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 just strips out those underscores and stores the usual digits.

So, you can write 100_000_000 and you should hopefully not confuse that with 10_000_000, whereas 100000000 is easily confused with 10000000. Returning to some of our previous examples:

```
long anotherLong = 2_147_483_648L;
int bitPattern = 0b0001_1100_0011_0111_0010_1011_1010_0011;
```

Notice how much easier to read the value being assigned to anotherLong is (Yes, it's 2GB again, and it's still too big to fit into an int).

By now, you should be convinced of the benefit of these tweaks to the handling of integers, so let's move on.

1.3.3 Improved Exception Handling

There are two parts to this improvement - multi-catch 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 handles 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 is a method which can encounter a number of different exceptional conditions:

- The config file may not exist
- It may disappear whilst we're trying to read from it
- It may be malformed syntactically
- It may have invalid information in it

They really fit into 2 distinct functional groups, though. Either the file is missing or bad in some way, or the file is in theory present and correct, but was unable to be retrieved properly (perhaps caused by hardware failure or network outage). It would be nice to compress the cases down into just these two cases. Java 7 allows us 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;
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=725>

```
}

```

Note that the exception `e` has to be handled in the catch block as the common supertype of the exceptions that it could be (which will usually be `Exception` or `Throwable` in practice) – as the exact type is not knowable at compile-time.

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

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

This forces the programmer to declare the exception signature of this code as `Exception` – the real dynamic type of the exception has been swallowed. However, it's relatively easy to see that the exception can only be an `IOException` or a `SQLException`, and if we can see it, then so can the compiler. In this snippet, we've made a single word change to use the next Java 7 syntax:

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

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

The `final` keyword is actually optional in the above example, but in practice, we've found that it helps to write it in while adjusting to the new semantics of `catch` and `throw`. In addition to these general improvements in exception handling, the specific case of resource management has been improved in 7 – so that's where we'll turn next.

1.3.4 Try-with-resources (TWR)

This change is easy to explain, but 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 (e.g. 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 handling of resource closing 100% right. Until recently, even the reference howtos from Sun were wrong. The proposal submitted to Project Coin for this change includes the astounding claim that 2/3 of the uses of `close()` in the JDK had bugs in them!

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

As an example, let's consider some Java 7 code for saving code from the web. As the name suggests, `url` is a URL object that points at the entity we want to download, and `file` is a File object where we want to save what we're downloading.

Listing 1.3 Java 7 Syntax for Resource Management

```
try ( FileOutputStream fos = new FileOutputStream(file);
      InputStream is = url.openStream() ) {
    byte[] buf = new byte[4096];
    int len;
    while ((len = is.read(buf)) > 0) {
        fos.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 then automatically disposed of when you're done with them.

This is a big help in writing error-free code. To see just how much, consider how you would write a similar block of code to read from a stream coming from a URL and write to a file with Java 6.

Listing 1.4 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 n;
        while ((n = is.read(buf)) >= 0)
            out.write(buf, 0, n);
    } catch (IOException iox) {
        #A
    } finally {
        try {
            out.close();
        } catch (IOException closeOutx) {
            #B
        }
    }
}
```

```

    }
  }
} catch (FileNotFoundException fnfx) {
#C
} catch (IOException openx) {
#C
} finally {
  try {
    if (is != null) is.close();
  } catch (IOException closeInx) {
    #B
  }
}
}
#A Handle exception (could be read or write)
#B Can't do much with exception
#C Handle exception

```

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 or
- to read from it or
- to close properly
- The `File` corresponding to the `OutputStream` can fail to open or
- to write to it or
- to close properly or
- some combination of more than one of the above

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

This is the main reason for preferring the new syntax – it's just much less error prone – the compiler is not susceptible to the mistakes which basically every developer will make when trying to write this type of code manually.

One other aspect of TWR is the appearance of enhanced stack traces and suppressed exceptions. Prior to 7 exception information could be swallowed when handling resources. This possibility also exists with TWR, so the stack traces have been enhanced to allow the developer 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 NPE is seen:

```

Exception in thread "main" java.lang.NullPointerException
  at wjgd.ch01.ScratchSuprExcep.run(ScratchSuprExcep.java:23)
  at wjgd.ch01.ScratchSuprExcep.main(ScratchSuprExcep.java:39)
  Suppressed: java.lang.NullPointerException

```

```
at wjgd.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 is included as part of JDBC 4.1, though.

For your own code, however, you should definitely use TWR whenever you need to work with resources – it will help you avoid bugs in your exception handling. As a helping hand, Java 7 ships with a tool which you can run over user code to detect places where `AutoCloseable` can help.

1.3.5 Diamond Syntax

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 a `userid` (which is an integer) and each user has some lookup tables (possibly more than one), and the tables are specific to each user. What would that look like in code?

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

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

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

and have the compiler just work out the type information on the right hand 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 just cut the older, more verbose declaration and start using the new type-inferred syntax to save a few pixels.

For the curious, we should point out that the compiler is using a new form of type inference for this feature. It is actually working out the correct type for the expression on the right hand side, and is not just substituting in the text which defines the full type.

The 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=725>

Instance Creation”, which is a real mouthful, and has ITIGIC as an acronym, which just sounds stupid – so Diamond Syntax it is.

1.3.6 Simplified Varargs Method Invocation

This is one of the simplest changes of all – it just moves a warning about type information for quite a 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 then 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 issue all about?

Well, as you probably know, a varargs method is one which 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 are 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 are not normally allowed to create an array of a known generic type. So, this:

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

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

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

That gives a warning which has to be ignored. Notice that you can *define* warnHm to be of the type 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 really working on synthetic arrays that the compiler conjures up; and arrays of known generic types not being an instantiable type – come together to cause us 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 basically 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 which mutters darkly about “uses unchecked or unsafe operations”.

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

WHAT'S CHANGED IN JAVA 7

The new feature which comes in 7 is to change 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 the site of use of the API to the site where the API was defined.

The warning used to be triggered when code which used the API was compiled. Instead, it's now triggered when an API is written which has the potential to trigger this kind of type safety violation. The compiler warns the coder implementing the API – and it's up to him or her 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) which would otherwise produce a warning of the type discussed above. By annotating the method with `@SafeVarargs`, the developer is essentially asserting that the method does not 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 – and completes our discussion of the changes brought in by Project Coin. Although they represent mostly small syntactic changes, they can have a positive impact on your code which is out of proportion with the size of the change. Once you've started using them in practice you will hopefully find that they offer real benefit to your programs.

1.4 Summary

This chapter has been all about introducing some of the smaller changes in the syntax for Java 7. We've discussed Project Coin, and seen how it is a possible model for how Java may develop in the future.

One other important concept we've seen is how 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. You should already have a grasp of how previous Java versions coped with I/O, as 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 asynch I/O classes, then the next chapter has all the answers.