Issue June 2011 | presented by jaxenter www.jaxenter.com

Java™ TECH JOURNAL

With no major release of the Java platform for nearly five years, the upcoming release of Java 7 is big news. But, with Lambda, Jigsaw, and part of Coin postponed until Java 8 and late 2012, what can we expect from the long-awaited Java 7? In this issue of Java Tech Journal, we examine the new features coming up in Java 7, and what impact these might have on the fast-changing programming language market.

#9

Java 7

Java 7 – Project Coin
A Gentle Introduction to Java 7

Bringing Java 7 Support to IntelliJ IDEA 10.5
Interview with IntelliJ IDEA senior software developer

Java the Language vs. Java the Platform
New and Noteworthy in JDK 7

Java 7: The Top 8 Features
The Must-Have Features in Java 7

Using Try-With-Resources in Java SE 7
Cleaning up your code and your resources

"Java 7 is here"

Not only is Java the world's number one programming language, but it also provides the backbone to a rich and complex ecosystem of tools and projects. With such a diverse community relying on Java, a major update is always going to be a tricky business – stability and backwards compatibility are big issues, and there's no shortage of contrasting ideas about what should be included in a new release of Java. Sun once estimated that it takes around 18 months to evaluate and approve proposals for changes to Java (and that's before you factor in the time it takes to build, test and release the change.) There's no such thing as a quick Java release, but Java 7 has taken longer than most: in 2007, the Java in Production blog gave 2009 as an approximate launch date.

This delay is in no small part due to the acquisition of Sun by Oracle. The original Java 7 roadmap was masterminded by Sun Microsystems, before the legal wranglings (largely centred around MySQL) turned the acquisition of Sun by Oracle into a near-twelve-month battle. It's hardly surprising Java 7 has been a long time coming.

Things finally came to a head with the proposal of Plan B. This put forward the idea of releasing JDK 7 in 2011, minus Lambda, Jigsaw, and part of Coin, as oppose to waiting until 2012 and delivering JDK7 with all of the initially planned features. Ultimately, the community and Oracle opted for a quicker update, with less features, which means we'll have to wait until 2012 and Java 8 to get our hands on lambda expressions, literal expressions for immutable lists, sets, and maps, and other deferred features. Although the open debate regarding Java 7 language features is a prime example of an open community in action, the Plan A vs. Plan B debate (and the eventual outcome) highlighted the average Java users' frustration at the language's pace of change.

It's been a long wait, but Java 7 is almost here! NetBeans 7 is already available with support for the Java SE Java Development Kit 7, and the latest 10.5 release of IntelliJ IDEA offers full Java 7 support. With Oracle recently announcing that the Java Development Kit 7 will be generally available on July 28, 2011, and work already underway for Java EE 7 and Java SE 8, this is an exciting time for Java! Not to mention JCP.next JSR 1 and JCP.next JSR 2, which Oracle have already submitted to "update and revitalise" the JCP: often criticised as one of the major bottlenecks when it comes to moving Java forward.

In this issue, we scrutinise the new features of Java 7, ask how relevant they will be to your typical Java developer, and look at Java 7 support in IntelliJ IDEA. Has the long wait been worth it? Read on to find out!

Jessica Thornsby

Index

A lot to explore

Java 7 – Project Coin

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 is on its way. We're going to warm you up with a gentle introduction to Java 7, but one that still acquaints you with its 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). You'll also learn about try-with-resources and how it helps you to avoid bugs in the code that deals with files or other resources such as JDBC. By the end of this article, you'll be writing Java in a new way and you'll be fully primed and ready for larger changes in Java 7 such as NIO.2. So, let's get going, shall we!

By Benjamin J. Evans and Martijn Verburg

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 "whys" – that is, we'll 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 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.

Strings in switch

Java's *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.out.println("Error!"); break;
  }
}
```

In Java 6 and earlier versions, the values for the cases can only be constants of type *byte*, *char*, *short*, and *int* (or, technically, their reference-type equivalents, *Byte*, *Character*, *Short*, and *Integer*) or *enum* constants. With Java 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;
    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 changes, this really is a very simple change for making life in Java 7 a little easier.

Enhanced syntax for numeric literals

Several proposals offered new syntax for integers. The aspects that were ultimately chosen were:

- Numeric constants (one of the integer primitive types) expressed as binary values.
- 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 particularly earth-shattering at first sight but all have in their own 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 they may have to indulge in a certain amount of bit twiddling. So let's take a 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 this:

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

This is a lot 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 two-argument form of *parseInt()*.
- Requires you to remember the detail of how *parseInt()* behaves when it has two args.
- Makes life hard for the JIT compiler.
- Is representing a compile-time constant as a runtime expression (so it can't be used as a value in a *switch* statement).
- Will give you a runtime exception (but no compile-time exception) if you get 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 values – for example, low-level handling of bytes, where you can now have bit-patterns as binary constants in switch statements – this is one small feature that you might find helpful.

Listing 1: Handling several different exceptions in Java 6

```
public Configuration getConfig(String fileName_) {

Configuration cfg = null;
try {
   String fileText = getFileText(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;
}
```

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 the hexadecimal system – because our minds find it easier to deal with shorter strings that contain more information rather than long strings containing little information per character.

That is, we find 1c372ba3 easier to deal with than 00011100000110111001010111010011, 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
```

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 (unlike 100000000, which is easily confused with 10000000). Or, to apply this to our own examples:

```
long l2 = 2_147_483_648L;
int bitPattern = 0b0001_1100__0011_0111__0010_1011__1010_0011;
```

Listing 2: Handling several different exceptions in Java 7

```
public Configuration getConfig(String fileName_) {

Configuration cfg = null;
try {
   String fileText = getFileText(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;
}
```

Notice how much easier it is to read the value assigned to *l2*. (Yes, it's 2G, and it's 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.

Improved exception handling

There are two parts to this improvement – multi-catch and, effectively, final rethrow. To see why they're helpful, consider the following Java 6 code, which tries to find, open, and parse a configuration file and handles a number of different possible exceptions, as shown in listing 1.

getConfig() is a method that can encounter a number of different exceptional conditions:

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

The exceptions really fit into two distinct functional groups. Either the file is missing or bad in some way or the file is present and correct in theory but was not retrievable (perhaps because of 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, as shown in listing 2.

Note that the exception *e* has to be handled in the *catch* block as the common supertype of the possible exceptions (which will usually be *Exception* or *Throwable* in practice) because 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 is that, in previous versions of Java, code like this:

```
try {
   doSomethingWhichMightThrowIOException();
   doSomethingElseWhichMightThrowSQLException();
} catch (Throwable t) {
   ...
   throw t;
}
```

Will force the programmer to declare the exception signature of this code as *Throwable* – 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 Throwable t) {
   ...
   throw t;
}
```

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. Enhancements in the compiler mean that the *final* keyword is actually optional, but we've found that, while starting out with this feature, it's actually easier to include it.

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

Try-with-resources

This change is easy to explain but has proven 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.

Listing 3: Java 6 syntax for resource management

```
InputStream is = null;
try {
    File file = new File("output.txt");
    URL url = new URL("http://www.java7developer.com/blog/?page_id=97");
    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) {
        // Handles exception (could be read or write)
    } finally {
        try {
            out.close();
        } catch (IOException closeOutx) {
            // Can't do much with exception
        }
    }
} catch (FileNotFoundException fnfx) {
    // Handles exception
} catch (IOException openx) {
    // Handles exception
} finally {
    try {
        if (is != null) is.close();
    } catch (IOException closeInx) {
        // Can't do much with exception
    }
}
```

This is an important change for the simple reason that virtually no one gets the manual handling of resource closing 100 percent right. Until recently, even the reference how-to from Sun was 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 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, which is usually referred to as *try-with-resources*.

This is a big help in writing error-free code. To see just how much, consider how you would write a block of code in order to read from a URL-based stream URL and write to a file with Java 6. It would look something like it's shown in listing 3.

The key point here is that, when handling external resources, Murphy's Law applies – anything can go wrong at any time:

1. The *InputStream* can fail:
- To open from the URL.
- To read from it.
- To close properly.

2. The file corresponding to the *OutputStream* can fail:
- To open.
- To write to it.
- To close properly.

3. Or have 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.

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

TIP

Java 7's NIO.2 API makes the above logic even simpler, with less exception handling. Check out the *Files* class for more details.

Try-with-resources and AutoCloseable

Under the hood, the try-with-resources feature is achieved by introducing a new interface called *AutoCloseable*, which a class must implement in order to appear as a resource in the new ARM *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 adopted this new technology – specifically, the AWT classes have not rolled it out yet.

For your own code, however, you should definitely use try-with-resources whenever you need to work with resources – it will help you avoid bugs in your exception handling.

where we want to save what we're downloading. Let's look at Listing 4.

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 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. You still need to worry about handling exceptions with regards to finding the valid resource in the first place, but, once you're using it, the resource gets automatically closed.

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 that basically every developer will make when trying to write this type of code manually.

Diamond syntax

One of the problems with generics is that the definitions and the setup of instances can be really verbose. Let's suppose that you have some users, whom you identify by a user id (which is an integer), and each user has some lookup tables, 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 the code below, and have the compiler just infer the type information on the right hand side?

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

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.

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, such as code that looks like this:

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

Then you can move on to the next section. Still here? Good. So what's this issue all about?

Well, as you probably already 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. 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];
```

Which gives a warning that 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 the synthetic arrays that the compiler conjures up and arrays of known generic types not being valid instantiable types – 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);
```

Listing 4: 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);
  }
} catch (IOException | FileNotFoundException e) { // If file is not found }
```

Why diamond syntax?

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* (ITIGIC), which is a headache to remember – so, diamond syntax it is.

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 that 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

Java 7 brought a change in 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 developer'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 the API use (the warning used to be triggered when the code that used the API was compiled) to the site where the API was defined (so the warning is now triggered when an API is written, which has the possibility to trigger this kind of potential type safety violation). The compiler warns the coder implementing the API and it's up to them to pay proper attention to the type system.

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 paper, we mentioned that Project Coin encouraged contributors to mostly try and stay away from the type system when proposing changes.

This example shows how involved you need to get 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, once you've started using them in practice, you will probably find that they have a positive impact on your code that is out of proportion with the size of the change.

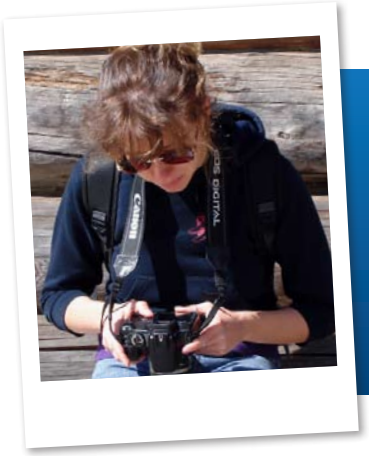
Summary

This article has been all about introducing some of the smaller changes in the syntax for Java 7. You saw that, although the changes are not earth-shattering, Java 7 will be a little bit easier to write in a more concise and error-free manner. You also learned that there can be challenges that cause language designers to make smaller and more conservative changes than they might otherwise wish.

We hope you enjoyed this article and look forward to discussing more about Java 7 and polyglot programming on the JVM with you in a pub near you soon!

Ben Evans is a member of the Java SE/EE Executive Committee, helping define standards for the Java ecosystem. He works as a technical architect and development lead in the financial industry. He is an organizer for the UK Graduate Developer Community, a co-leader of the London Java Community and a regular speakeron Java, concurrency, new programming languages and related topics.

Martijn Verburg is a Dutch Born Kiwi who co-leads the London JUG (aka the LJC) and also is heavily involved in the London graduate/undergraduate developer, CTO and software craftsmanship communities. The Javaranch kindly invited him to be a bartender in 2008 and he's been humbled by the awesomeness of that community ever since. He's currently working on somewhat complex JCA Connectors and an associated open source middleware platform (Ikasan) and also spends a good deal of time herding monkeys on another open source project that deals with creating characters for d20 based role playing games (PCGen). More recently he's joined Ben Evans in writing "The Well-Grounded Java Developer (Covers Java 7 and polyglot programming on the JVM) for Manning publications and can be found speaking at conferences (such as TSSJS and DevNexus) on a wide range of topics including open sourcing software, software craftsmanship and the latest advancements in the OpenJDK.



Bringing Java 7 Support to IntelliJ IDEA 10.5

"Java 7 in Action"

Back in February, JetBrains announced that the major focus of their 10.5 release, would be Java 7 support. In this interview, we speak to IntelliJ IDEA senior software developer, Anna Kozlova, to find out more about implementing the new Java 7 language features in their latest IDE release.

JTJ: IntelliJ IDEA 10.5 will come with full Java 7 support. What has been the most challenging Java 7 language feature to implement in IntelliJ IDEA?

Anna Kozlova: Definitely, the ability to support String types in switch statements. We even implemented it as early as IntelliJ IDEA 10, to have more time to polish its usability. Indeed, the real challenge is to integrate new language features with the huge number of IntelliJ IDEA's intentions, inspections, refactorings and so on. It is not rocket science to write a lexer or a parser. It is much more important and time-consuming to get all the existing features updated to match the change, so that everything still works consistently after the new features have been added.

JTJ: For you, what key benefit will Java 7 support bring to the IntelliJ IDEA user?

Anna: IntelliJ IDEA has always been good at generating and completing complex code fragments for you. Now, with the new 'language syntactical sugar,' even this is not needed anymore. So, that is the benefit - even less typing, and thus faster coding.

JTJ: How have you tried to manage the learning curve, for developers moving to Java 7 with IntelliJ IDEA 10.5?

Anna: As always, IntelliJ IDEA provides batch inspections for an easier transition. For example, it is possible to run "Explicit type can be replaced with <>" inspection over the whole project and to apply a simplification fix which will convert new expressions to use diamonds. Of course, it will convert only the usages where diamonds are applicable. We've also added other inspections to compactify 'try/finally' to 'try' with resources, collapse identical branches into multi-catch try, etc.

JTJ: Java 7 support has been available in early access builds of IntelliJ IDEA since March. What are the challenges of supporting a technology that's still under development?

Anna: Apart from the fact that we had to rework some parts... One other situation that we've faced recently was a difference between the specification and the way the Java compiler works. We decided to follow the compiler because this is what people will use.

JTJ: And, looking forward, what are the plans for Java 8 support in future releases of IntelliJ IDEA?

Anna: It doesn't take much planning to decide that all of the features will be fully supported and that we will provide inspections and refactorings to help introduce the new features into the codebase, just the same as we did with Java 7 in IntelliJ IDEA 10.5. As for the specific features and timeframes, it all depends on how the development of Java 8 itself proceeds.

Portrait

Passionate about software development and with over 10 years of professional experience, Anna has been one of the core IntelliJ IDEA developers since joining JetBrains in 2004. Her major areas of expertise include code analysis, code refactorings, and test tools integration.

JAX – The Premier Java, Architecture & Agile Experience

JSF • summit

JAX 2011 It's about you, it's about Java, it's about Web, Architecture, Cloud & Agile

JAX is one of the world's most comprehensive conferences on web and enterprise development. It provides the ideal forum for software developers, project managers and architects to learn about all the latest Technology, Architecture and Agile Methodologies. JAX has become internationally renowned for its unique blend of topics, since its conception in 2001. This year JAX is pleased to be working in collaboration with the popular JSF Summit; a conference for application developers, solution architects, and project managers who develop applications with JavaServer Faces (JSF), Seam, Java EE, and related technologies. We'd like to provide you a snapshot of our programme.

All updates, keynotes and talks can be found on www.jaxconf.com

KEYNOTE

The Future of Java

Rod Johnson (VMware, Creator of Spring Framework)

The availability of cloud computing resources fundamentally changes the way that enterprises use technology and introduces new programming paradigms. Rod Johnson, Senior Vice President of Application Platform Division at VMware and founder of the Spring Framework, will discuss how these changes have produced a demand for an enterprise Java cloud platform and how existing enterprise standards are not sufficient to meet these new challenges. Developers need to build applications that leverage a dynamic and changing infrastructure, access data in non-traditional storage formats, perform complex computations against large data sets, support access from a plethora of client platforms and do so more quickly than ever before without sacrificing scalability, reliability and performance. Meeting these demands is necessary to maintain Java as the most useful technology to the enterprise and requires the introduction of an open, productive, Java Platform-as-a-Service.

SESSIONS

Toward Java SE 8: Project Lambda

Daniel Smith (Oracle)

This talk will cover the primary new language features for Java SE 8 – lambda expressions, method references, and extension methods – and explore how both existing and future libraries will be able to take advantage of them to make client code both more performant and less error-prone.

JavaSE7 - Overview

N.N.

With the release of the long anticipated Java SE 7 this summer, developers will gain an in depth look, with the help of code examples and scenarios at the new features of the platform. Shaped by the big trends in the computing industry of multiple languages and concurrent programming, attendees will understand the advantages of the new enhancements to the Java language, will gain insight into the new Filesystem API and learn how best to apply it to their existing programs, and will learn the advantages Java SE 7 brings to running other languages such as Groovy, Scala, Python, Ruby and more. They will understand how they can apply the latest techniques in concurrent programming, with the new Fork/Join framework, in order to maximize the performance of their applications on multi-core multi-processor architectures.

The Ceylon Language Module

Gavin King (Red Hat)

Java - the language and the platform - is one of the great success stories of the computing industry. Java code is robust and easy to understand, making it appropriate for large-scale deployments and large-team development. And Java was the first major language amenable to automated refactoring and other sophisticated tooling.

As Java continues to age, many developers ask what a language for general purpose and business computing would look like if it were designed today, with a close eye on the successes and failures of Java. For the past two or three years, our team at Red Hat have also been asking ourselves that question. The result is the Ceylon Project - a prototype language for the Java Virtual Machine which attempts to combine the strengths of Java with the power of higher order functions, a declarative syntax for defining user interfaces and specifying structured data, and a completely redesigned SDK.

This talk demonstrates some interesting features of the language by exploring the design of the basic types built into the Ceylon language module.

New and Noteworthy in JDK 7

Java the language vs. Java the platform

The recent preview release of JDK7 is the first public message Oracle has sent amidst the uncertainty about the future of Java. JDK7 was originally targeted for 2008-2009 [1] and promised some great new language features, most notably lambda support, new collections support and unsigned literals. Some twenty-four months late, the preview release includes only a handful of new language features but given the rocky road so far, that's probably to be expected. In this article, we'll take a closer look at some of the new features and discuss how useful they're actually likely to be and by extension, Java's place in the overcrowded language market.

By Toby Weston

Java has started to show its age, it's over sixteen years old now and hasn't kept up with the modern developer. The JVM has proven itself as a serious platform for execution but the language itself has started to feel dated. With the current trend towards functional-style programming and the rise of JVM targeted languages such as Scala, *Java the language* has found itself in the position where it has to compete with *Java the platform*. Can the new language features of JDK7 bolster the language's position or is it just too late? A summary of the new and noteworthy language features include:

- Type inference on generic object creation
- Try-with-resources statements
- Catching multiple exceptions in a single catch block

Type Inference on Generic Object Creation and Constructor Arguments

This new feature allows a little brevity to the garrulity of the language, at least when instantiating new generic objects when the type can be inferred. For example,

```
private Map<Size, List<Shoe>> stock = new HashMap<Size, List<Shoe>>();
```

can be reduced to

```
private Map<Size, List<Shoe>> stock = new HashMap<>();
```

where the *diamond operator* can be inferred from the declaration. It's subtly different than leaving out the generic completely, which would reduce your type to being of *Object*. General inference rules apply. So for example, return types of methods can be used to infer the type as in the example below.

```
private Callable<Long> calculateExecutionTime() {
    return new Callable<>() {
        @Override
        public Long call() throws RuntimeException {
            return ...
        }
    };
}
```

Things don't get much better than this. Actually, they do. Just a little. Constructor generics always used to be fun and that hasn't really changed, although with JDK7 you can do a little more. For example,

```
public class Bob<X>{
  public <T> Bob(T t) {
  }

  public static void example() {
  Bob<Integer> bob = new Bob<>("yum");
  }

  public static void anotherExample() {
  Bob<Integer> bob = new <String> Bob<Integer>("yum");
  }
}
```

These examples are the same as the ones Oracle give (more or less) [2], they both work with JDK7 only and show the *Integer* type inferred as the class generic (*X*) in combination with the diamond operator. The second example shows new syntax to explicitly set type of the method generic to give some additional compile time checks. Specifically, if you attempt something crazy such as

```
private void yetAnotherExampleDoesNotCompile() {
  Bob<Integer> bob = new <String> Bob<Integer>(30.5); // wont compile!
}
```

you'll see the friendly compilation error like this

```
constructor Bob in class Bob<X> cannot be applied to given types;
required: T
found: double
reason: actual argument double cannot be converted to String by method
                                                        invocation conversion
where T,X are type-variables:
T extends Object declared in constructor <T>Bob (T)
X extends Object declared in class Bob
```

Interestingly, Oracle's own examples from [2] don't actually compile against the preview JDK7 release. In the official documentation, they show the following

```
private void anotherExampleDoesNotCompile() {
  MyClass<Integer> aClass = new <String> MyClass<>(""); // wont compile!
}
```

which wouldn't compile for me, citing a *cannot infer type arguments for MyClass<>* error. This would imply that you can't use the diamond operator with explicit type specification against generic constructor arguments. This just sounds too flakey, I'm sure it's an oversight and subsequent updates will move inline with Oracle's documentation. They have also seemingly slipped a typo into the official documentation with a roguè in their example;

```
MyClass<Integer> myObject = new <String`> MyClass<>("")
```

Remove it and things will compile. Leave it and languish. The trouble is, pretty much all the examples on the web have been based on their documentation so cut-and-pasters beware. As an attempt to reduce the visual clutter we're exposed to, this feature isn't very impressive at all. In fact, IDEs such as IntelliJ IDEA have been doing it for some time. If you look at the first example above in IDEA, it will automatically use code folding to hide the repetition and display something like the following.

```
private Map<Size, List<Shoe>> stock = new HashMap<~>();
```

combined with the hit and miss documentation, this new language feature from Oracle is decidedly underwhelming.

try-with-resources Statements and AutoCloseable

Another bugbear with the verbosity of Java has always been the try-catch-finally syntax. The new language feature try-with-resources statement allow you to compact this in combination with auto-closable resources. Here, rather than the familiar, try-finally to close a resource, you can "open" the resource within the parenthesis of the try statement (as long as the object implements the *AutoCloseable* interface) and Java will take care of the close call. For example, Oracle's documentation [3] shows how

```
private String example() throws IOException {
  BufferedReader reader = new BufferedReader(...);
  try {
    return reader.readLine();
  } finally {
    reader.close();
  }
}
```

becomes,

```
private String example() throws IOException {
  try(BufferedReader reader = new BufferedReader(...) {
    return reader.readLine();
  }
}
```

This reduced syntax is interesting as it goes a long way to reducing the noise typical to try blocks. An expanded and all too familiar example might be the common try-try-catch-do-nothing block such as the following.

```
public void ridiculous() {
  FileInputStream stream = null;
  try {
    stream = new FileInputStream(...);
  } catch (FileNotFoundException e) {
    // ...
  } finally {
    if (stream != null) {
      try {
        stream.close();
      } catch (IOException e) {
        // ... seriously?
      }
    }
  }
}
```

Which can be reduced to

```
public void lessRidiculous() {
  try (FileInputStream stream = new FileInputStream(new File(""))) {
    // do your thing
  } catch (FileNotFoundException e) {
    // ...
  } catch (IOException e) {
    // ...
  }
}
```

Here, the auto-closable resource has taken care of the call to close the stream and in its implementation has kindly taken care of the null check for us too. On the down side, the implementation of *close* in the *FileInputStream* has added an *IOException* to the catch list (more accurately, the *Closeable* interface which extends *AutoCloseable* and is implemented by *FileInputStream* has added the exception). Despite the exception, all in all, this

should go some way towards tidying up this kind of resource usage so it gets the thumbs up. Its not clear to me however, why Oracle have chosen to miss-spell the interface names though.

```
interface Closeable extends AutoCloseable {
   public void close() throws IOException;
}
```

There is a little extra detail which could be troublesome when using try-with-resources and that's suppressed exceptions. Exceptions thrown from within the *close* method can be suppressed in favour of exceptions thrown from within the try statement's block. Lets look a little closer at this.

```
public class Fudge {

   public void suppressionOfException() {
      try (Foo foo = new Foo()) {
         throw new RuntimeException();
      } catch (CloseException e) {
         // aint gonna happen
      }
   }

   private class Foo implements AutoCloseable {
      @Override
      public void close() throws CloseException {
         throw new CloseException("exception closing resource");
      }
   }
}
```

The above example demonstrates an exception being thrown in the *close* method but it being suppressed, the actual exception caught by the default exception handler in this case will be *RuntimeException*. For example,

```
Exception in thread "main" java.lang.RuntimeException
 at Fudge.suppressionOfException (Fudge.java:36)
 at Fudge.main(Fudge.java:30)
 ...
 Suppressed: Fudge$CloseException: exception closing resource
  at Fudge$Foo.close(Fudge.java:45)
    ... 6 more
```

If we put something together based on a decompiled version of the above, you can see what happens behind the scenes.

```
public void simulatingSuppressionOfException () {
      AutoCloseable closeable = new Foo();
      Throwable throwable = null;
      try {
         // this is the statement block and in our case with throw an exception
         throw new RuntimeException();
      } catch (Throwable e) {
         throwable = e;
         throw e;
      } finally {
         try {
            closeable.close();
         } catch (Exception e) {
            throwable.addSuppressed(e);
         }
      }
}
```

The JavaDoc tells us that in these situations two exceptions were logically thrown but because the flow of control can only continue with one exception, the other is suppressed. Supressed exceptions are new in JDK7 and can be retrieved in a similar way a *cause* can via a "getter" method. I can see this occasionally causing the odd problem as I imagine it will become another less well-known caveat that you're not going to need to be aware of until it's too late. To be fair though, it's likely to be something that will be more of an issue when writing your own *AutoCloseable* implementations than using Oracle's retrofitted classes.

What's perhaps a little more concerning is putting together tests for things that use *AutoCloseable* as collaborators. Previously, if something works with an *InputStream*, we would typically inject that (interface) directly into the class under test and have at it. We're unable to do that when we "new up" the collaborator within a try-with-resources statement so we're forced to pass in a factory. Not really a huge issue but it can lead to another indirect collaborator that you could argue obfuscates things. For example, the following won't compile.

```
public class Example {

   private final AutoCloseable closeable;

   public Example(AutoCloseable closeable) {
      this.closeable = closeable;
   }

   public void methodThatThrowsMultipleExceptions() {
      try (closeable) {   // compilation problem!
         // ...
      } catch (Exception e) {
         // ...
      }
   }
}
```

so, we're forced to use a factory.

```
public class Example {

   private final AutoCloseableFactory factory;

   public Example(AutoCloseableFactory factory) {
      this.factory = factory;
   }

   public void methodThatThrowsMultipleExceptions() throws Exception {
      try (AutoCloseable closeable = factory.create()) {
         throw new RuntimeException();
```

```
    }
  }
}
```

Which in turn means a typical test (in our case using *jmock*) is a little more verbose. I'll leave it to you to decide if this could become a problem.

```
@RunWith(JMock.class)
public class ExampleTest {

  private final Mockery context = new Mockery();

  private final Factory factory = context.mock(AutoCloseableFactory.class);
  private final AutoCloseable closeable = context.mock(AutoCloseable.class);
  private final Example example = new Example(factory);

  @Test
  public void shouldThrowExceptionFromStatementBlock() throws Exception {
    context.checking(new Expectations(){{
      one(factory).create(); will(returnValue(closeable));
      one(closeable).close(); will(throwException(new CloseException()));
    }});
    try {
      example.methodThatThrowsMultipleExceptions();
      fail();
    } catch (RuntimeException e) {
      assertThat(e.getSuppressed(), hasItemInArray(instanceOf(
                                                    CloseException.class)));
    }
  }
}
```

Dr Kabutz combined this new feature with a way to automatically unlock locked resources in a recent news letter [5]. Here, the Java champion implements a basic unlock of a *java.util.concurrent.Lock*.

```
public class AutoLockSimple implements AutoCloseable {
  private final Lock lock;

  public AutoLockSimple(Lock lock) {
    this.lock = lock;
    lock.lock();
  }

  public void close() {
    lock.unlock();
  }
}
```

with the client calling something like

```
private void doSomething() {
  try { new AutoLockSimple(new ReentrantLock()) } {
    // do some stuff under the lock's protection
  }
}
```

Although this is an interesting use of the new feature, developers have been getting around this kind of verbosity for a while by wrapping some anonymous instance of an interface or decorating classes with this kind of boiler plate repetition. An example I wrote for the tempus-fugit micro-library looks like this:

```
public class ExecuteUsingLock<T> {

  private final Callable<T> callable;

  private ExecuteUsingLock(Callable<T> callable) {
    this.callable = callable;
  }

  public static <T> ExecuteUsingLock<T> execute(Callable<T> callable) {
    return new ExecuteUsingLock<T>(callable);
  }

  public T using(Lock lock) throws E {
    try {
      lock.lock();
      return callable.call();
    } finally {
      lock.unlock();
    }
  }
}
```

The "close" call is found in the familiar *finally* block. This is a good example of moving towards a lambda-like approach where clients would call some anonymous implementation like the following (making use of static imports for more syntactic sugar).

```
private void doSomethingDifferent() {
  execute(something()).using(lock);
}

private Callable<Void> something() {
  return new Callable<Void>() {
    public Void call() throws RuntimeException {
      // do some stuff under the lock's protection
      return null;
    }
  };
}
```

The reason I mention this alternative is to reflect on the more significant move to support lambdas that Oracle has put off. The tempus-fugit example is verbose because Java is verbose but with language support for lambdas, developers would be free to solve their own problems in a concise way. The tempus-fugit example is working within Java's constraints, attempting to push aside the noise but with the introduction of lambdas proper, we wouldn't need to. Introducing try-with-resources is a response to the noise we usually put up with but it's focused on a very specific case. If instead, we saw lambda

support, there just wouldn't be such demand for things like this; we'd have all coded our way out of it already.

Catching Multiple Exceptions

This new feature allows you to catch multiple exceptions using a pipe to separate exception types. It removes the duplicated code you often get catching several exceptions and treating them in the same way. For example,

```
catch (IOException | SQLException e) {
  logger.log(e);
  throw ex;
}
```

It looks like another workaround for the general gripes with Java; if you've got pages and pages of catch statements around a piece of code, it's probably trying to tell you something. Exception handling is often contentious in Java. Forcing checked exceptions can often lead to the over use of the catch-and-re-throw anti-pattern and it takes a carefully considered approach to avoid the mess.

Some alternatives to leaning on the new syntax which may well lead to a better system, include decomposing the problem, identifying and separating roles and responsibilities and as a by-product isolating exception generating code. You could also try using lambda-like anonymous interface implementations or vanilla decoration to push off to the side the exception handling code (typically logging or wrapping works best here).

What's probably more important than the mechanics though is identifying the real boundaries of your system; those places where you actually interact with system actors like the UI, frameworks or just the architectural "layers" of your system. Once you've spotted these, you can take steps to deal with exceptions at the appropriate place and answer the question of when to re-throw. The logical extension to this is to treat exceptions as sub-classes of *RuntimeException* and only catch and process them at your boundaries. Exclusively avoiding checked exceptions can reduce the clutter enormously but throwing runtime exceptions forces a high degree of responsibility onto the developer, something that is at odds with the typical "code defensively" development culture.

Given the example from Oracle above, I suspect this new feature will just facilitate ugly, jammed in code. It seems to say "it's ok to deal with a bunch of exceptions in the same way. In fact, we'll make it easier for you". Typical to the Java world, there's never a caveat around if you actually *should* be doing something, just an outline of *how* you could. The fact the example above (Oracle's example, by the way) logs then re-throws is a smell in itself, something that developers around the world are likely to copy (it has the official Oracle stamp of approval after all). Perhaps I'm being too harsh, but I'm not a fan of this one.

Miscellaneous

There has also been a bunch of API additions in the JDK7 release, too many additions to mention here. A few notables however, include a new class *Objects* which offers helper methods to help with null safety and a trivial deep equals method. From the ever-popular concurrency package, there's a new double-ended queue (Deque) implementation and a linked list based transfer queue. All in all though, I don't imagine the average developer has been crying out for, or will relish, these minor additions.

I've certainly focused a lot on the language features and the JDK7 release is going to be more than just language features including updates to JDBC, NIO, and the new Sockets Direct Protocol for streaming over InfiniBand fabric on Solaris. I've also left out the details of some of the other new language features. We can look forward to more specificity on throwing exceptions, better support for dynamically typed languages running on the JVM (via the new *invokedynamically* byte code instruction), binary literals (0B10101010), underscores in numeric literals and the ability to use strings in switch statements.

Conclusion

Recent trends and the emergence of new JVM targeted languages has meant that Java has started to feel a little dated, a little long in the tooth. Amongst others, Scala is offering a less verbose, more elegant way to exercise our craft. The new language features of JDK7 are clearly aimed at addressing some of the communities' frustrations with Java but just don't go far enough to repair Java's fading reputation. We're left on tenterhooks for JDK8 just like we've been on tenterhooks for JDK7 for this past two-dozen months.

With no major release of the Java platform for nearly five years [4], Oracle has a lot of lost time to make up for. Newer, more elegant and less verbose languages have emerged to meet the developer communities' needs and most of them run on the JVM. As a platform, Java feels safe and secure, a warm place to curl up in. It's a proven, hardened platform and new players are happy to build their futures on that premise. However, to compete as a *language*, Oracle would have to lurch Java's language features forward like they did in 2004 with the release of Java 5, and the JDK7 release just isn't it. Too much ground has been lost and Java's fallen out of touch with the modern developer. Lambda support or reifiable generic types may have helped keep things fresh but I wonder if they'll ever arrive and if so, whether we'll have all moved on by then. The platform itself should also look forward. The lack of JVM support for infinite stacks or atomic multi-address updates (supporting STM) mean the platform can't afford to sit on its laurels either.

Toby is an independent consultant specialising in agile software development and helping teams deliver. He's passionate about testing, concurrency and open source software. He has contributed to many open source projects and created the tempus-fugit Java micro-library. If you enjoyed this article, head over to Toby's blog for more of the same at http://pequenoperro.blogspot.com/.

Links & Literature

[1] http://today.java.net/pub/a/today/2007/08/09/looking-ahead-to-java-7.html

[2] Type Inference and Generic Constructors of Generic and Non-Generic Classes

[3] The try-with-resources Statement

[4] http://en.wikipedia.org/wiki/Java_version_history

[5] Automatically Unlocking with Java 7 Dr Heinz Kabutz

Language Enhancements and Features

Java 7: The Top 8 Features

It's been a while since the last major Java release and expectations were naturally high for the upcoming release. The Java 7 release initially included many JSRs with exciting features, like support for closures, which were later deferred to Java 8 in order to release JSRs that are already done. This effectively diluted what is now offered in Java 7 and has left some disappointed.

by Vineet Manohar

The Java language has undergone major changes since I started using it in 1998. Most of the changes were driven by the **Java Community Process** (**JCP**) which was established in 1998 as a formal and transparent process to let interested individuals and entities participate and influence how the language should evolve. This is done through the submission of a change request, known as **Java Specification Request** (**JSR**), followed by a review and a voting process. Changes or enhancements made to the language can be usually tracked back to a JSR where they were originally put forward for review. For example, the addition of Generics in Java 5 was done via **JSR 14**.

Version	Release Date
Java 1.0	1996
Java 1.1	1997
Java 1.2	1998
Java 1.3	2000
Java 1.4	2002
Java 5	2004
Java 6	2006
Java 7	*Expected mid 2011*
Java 8	*Expected mid-late 2012*

Table 1: Java timeline

Java releases

Here's a quick snapshot of the past Java release dates (table 1).

There are several small new features and enhancements in Java 7. Out of the 28 features that I looked at, here are the ones that I found useful.

New features and enhancements

#1 Strings in switch

In programming, we often encounter situations where we need to do different things based on different values of a variable. For *Boolean* variables, an *if-then-else* statement is the perfect way of branching code. For primitive variable types we use the *switch* statement. However, for *String* variables, we tend to resort to using multiple *if-then-else* branches as follows.

Java 6 and Before

```
if (language.equals("java") || language.equals("scala")) {
  // static
}
else if (language.equals("groovy")) {
  // dynamic
}
else if (language.equals("clojure")) {
  // functional
}
else {
  // new language!
}
```

One workaround for this is to convert the String into an enum and then switch on the enum.

Java 7

Java 7, however, has added a language level support for *String in switch*. Now you can rewrite the same code more elegantly:

```
switch (language) {
case "java":
case "scala":
  // static
  break;
case "groovy":
  // dynamic
  break;
case "clojure":
  // functional
  break;
default:
  // new language!
}
```

Not only does this help us write more readable code, but it also helps the compiler generate more efficient byte code as compared to the *if-then-else* by actually switching on the *hashcode()* and then doing an *equals()* comparison.

Please note that you will get a *NullPointerException* if the variable *language* in the above example resolves to *null*.

I like this feature, but unlike some of the other enhancements in the past (like Generic in Java 5), I don't anticipate using this feature a lot.

Practically, I find myself using *if-then-else* for one or two values and resort to an Enum when the number of values are higher.

#2 try-with-resources statement

One of the most useful additions in Java 7 is the auto closing of resources like *InputStream* which helps us reduce boiler plate code from our programs. Suppose we were writing a program which reads a file and closes the *FileInputStream* when it's done, here is how you would write the program:

With Java 6 and Before

```
InputStream is = null;

try {
  is = new FileInputStream(new File("foobar.txt"));

  // read file
  // ...
}
catch (IOException ex) {
  // handle this error
}
finally {
  if (is != null) {
    is.close();
  }
}
```

I want to point out a couple of things in this code. Firstly, notice that we declare the *FileInputStream* outside the *try* block just so that it can be accessed in the *finally* block. The second observation is that we need to initialize the *InputStream* to *null*, so that it is guaranteed to be initialized when we access it in the *finally* block. Last but not the least, the *is.close()* in the *finally* block may throw an *Exception* as well, thereby hiding the original *Exception* thrown in the *try* block *Exception* from the caller. What we probably want is to handle the *Exception* thrown from *is.close()* and throw the original *IOException*.

```
InputStream is = null;

try {
  is = new FileInputStream(new File("foobar.txt"));

  // read file
  // ...
}
catch (IOException ex) {
  // handle this error
}
finally {
  if (is != null) {
    try {
      is.close();
    }
    catch (IOException ex) {
      // supress this exception
    }
  }
}
```

The above code still has a shortcoming that the *Exception* thrown from *finally* is supressed and not accessible to the calling code. I'm not sure how often we want to get both the original *Exception* and also the *Exception* thrown from the *finally* block, but if we did want it, we could do always do something like this:

```
InputStream is = null;

try {
  is = new FileInputStream(new File("foobar.txt"));

  // read file
  // ...
}
catch (IOException ex) {
  // handle this error
}
finally {
  if (is != null) {
    try {
      is.close();
    }
    catch (IOException ex) {
      // supress this exception
      // ... but save it in a thread local object
      SuppressedException.getThreadLocal().setException(ex);
    }
  }
}
```

SuppressedException above is a user written Java bean with a field named *suppressed* of type *Exception*. The calling code can then call *SupressedException.getThreadLocal().getException()* to get the *Exception* that was supressed in the *finally* clause. Great, we solved all the problems associated with the *try-catch-finally!* Now we must remember to repeat this exact sequence with each use of *try-catch-finally* when handling files or other resources which need to be closed. Enter Java 7, and we can do the above without the boiler plate code.

With Java 7

```
try (InputStream is = new FileInputStream(new File("foobar.txt"))) {
  // read file
  // ...
}
catch (IOException ex) {
  // handle this error
}
```

try can now have multiple statements in the parenthesis and each statement should create an object which implements the new *java.lang.AutoClosable* interface. The *AutoClosable* interface consists of just one method.

```
void close() throws Exception {
}
```

Each *AutoClosable* resource created in the *try* statement will be automatically closed! If an exception is thrown in the *try* block and another *Exception* is thrown while closing the resource, the first *Exception* is the one eventually thrown to the caller. The second *Exception* is available to the caller via the *ex.getSupressed()* method. *Throwable.getSupressed()* is a new method added on *Throwable* in Java 7 just for this purpose.

Mandatory catch block when using the try statement

Note that if you are creating *AutoClosable* resources in the *try* block, you are required to declare a *catch* block to catch the concrete exception thrown from the actual *AutoClosable.close()* method. Alternatively you need to throw the *Exception*. Think of the *close()* method as implicitly being called as the last line in the *try* block. So, if an application has its own *AutoClosable* class as follows:

```
public class MyAutoClosable implements AutoClosable {
  public void close() throws MyException {
    // do something...
    // if error, then
    throw new MyException();
  }
}
```

Then, the following will cause a compile error:

```
try (MyAutoClosable my = new MyAutoClosable();) {
  // do something...
}
finally {
  // done
}

ERROR: unreported exception MyException; must be caught or declared to be thrown
```

To fix the above, you need to catch or throw the *Exception* from the calling method.

```
try (MyAutoClosable my = new MyAutoClosable();) {
  // do something...
}
catch (MyException ex) {
  // catch exception arising from the close()
}
finally {
  // done
}
```

Syntax for declaring multiple resources

The try statement can contain multiple statements separated by semicolon. Each statement must result in an *AutoClosable* object.

```
try (
InputStream is = new FileInputStream(new File("foo.txt"));
InputStream is2 = new FileInputStream(new File("bar.txt"))
) {
  // read file
  // ...
}
catch (IOException ex) {
  // handle this error
}
```

It is illegal to declare any variable which isn't an AutoClosable.

```
try (
InputStream is = new FileInputStream(new File("foo.txt"));

// illegal
long startTime = new java.util.Date().getTime();
) {
  // read file
  // ...
}
catch (IOException ex) {
  // handle this error
}
```

Output

```
ERROR: try-with-resources not applicable to variable type
  required: AutoCloseable
     found: long
```

AutoClosable vs Closable

The old *Closable* interface introduced in Java 5, which also has the same method that now extends from *AutoClosable*, implying that all *Closable* classes are automatically *AutoClosable*. Those classes automatically become resources that can be created in the *try* statement. The slight difference in *AutoClosable* and *Closable* is that unlike *Closable.close(), AutoClosable.close()* is not supposed to be idempotent, which means that calling it twice can have side effects. The second difference is that since exceptions thrown from *AutoClosable.close()* are suppressed, *AutoClosable.close()* should not throw exceptions which can cause problems when suppressed, like the *InterruptedException*.

#3 More precise rethrow

There are often situations when we want to catch multiple exceptions of different types, do "something" with them, and rethrow them. Let us consider this example, where we have some code which throws *IOException* and *SQLException*.

```
public static void main(String[] args) throws IOException, SQLException {
        try {
          loadFileToDb();
        } catch (SQLException ex) {
          System.err.print(ex.getMessage());
          throw ex;
        } catch (IOException ex) {
          System.err.print(ex.getMessage());
          throw ex;
        }
  }
```

In the above example, we are logging each type of exception being thrown by the *try* bock before rethrowing them. This results in a duplication of code in the *catch* blocks. Before Java 7, to get around the code duplication issue we would catch the base exception as follows:

```
public static void main(String[] args) throws IOException, SQLException {
        try {
                loadFileToDb();
        } catch (Exception ex) {
                System.err.print(ex.getMessage());
                throw ex;
        }
}
```

However, this requires us to rethrow the base Exception type java.lang.Exception from the calling method.

```
public static void main(String[] args) throws Exception {
```

With Java 7

Java 7 has added the "precice rethrow" feature which lets you catch and throw the base exception while still throwing the precise exception from the calling method.

```
public static void main(String[] args) throws IOException, SQLException {
        try {
                loadFileToDb();
        } catch (final Exception ex) {
                System.err.print(ex.getMessage());
                throw ex;
        }
}
```

Note the keyword *final* in the above *catch* clause. When a parameter is declared *final*, the compiler statically knows to only throw those checked exceptions that were thrown by the *try* block and were not caught in any preceding *catch* blocks.

#4 Multi-catch

There is no elegant way with Java 6 to catch multiple exceptions of different types and handle them in the same way.

```
try {
  doSomething();
} catch (Ex1 ex) {
  handleException();
} catch (Ex2 ex) {
  handleException();
}
```

You can always catch the parent *Exception* in order to avoid duplication of code, but it is not always suitable especially if the parent is *java.lang.Exception*.

Java 7 lets you catch multiple *Exceptions* in a single catch block by defining a "union" of *Exceptions* to be caught in the catch clause.

```
public class MultiCatchDemo {
    public static class Ex1 extends Exception {}
    public static class Ex2 extends Exception {}
    public static class Ex3 extends Exception {}
    public static class Ex4 extends Exception {}
```

```
    public static void main(String[] args) throws Ex1, Ex2, Ex3, Ex4 {
            try {
                    doSomething();
            } catch (Ex1 | Ex2 ex) {
                    handleException();
            } catch (Ex3 | Ex4 ex) {
                    handleException2();
            }
    }

    private static void doSomething() throws Ex1, Ex2, Ex3, Ex4 {
            // ...
    }
}
```

Note that the pipe '|' character is used as the delimiter. The variable '*ex*' in the above example is statically typed to the base class of *Ex1* and *Ex2*, which is *java.lang.Exception* in this case.

#5 Binary integral literals
With Java 7, you can now create numerical literals using binary notation using the prefix "`0b`"

```
int n = 0b100000;
System.out.println("n = " + n);
```

Output

```
n = 32
```

#6 Underscores in numeric literals
With Java 7, you can include underscores in numeric literals to make them more readable. The underscore is only present in the representation of the literal in Java code, and will not show up when you print the value.

Without underscore

```
int tenMillion = 10000000;
System.out.println("Amount is " + tenMillion);
```

Output

```
10000000
```

With underscore

```
int tenMillionButMoreReadable = 10_000_000;
System.out.println("Amount is " + tenMillionButMoreReadable);
```

Output

```
10000000
```

More rules and examples
1. Consecutive underscores is legal.

```
int n = 1000_____000;
```

2. Underscores can be included in other numeric types as well.

```
double d =  1000_000.0d;
long l = 1000_000l;
int hex = 0xdead_c0de;
int bytes = 0x1000_0000;
```

3. Underscore can be included after the decimal point.

```
double example8 = 1000_000.000_000d;
```

4. It is illegal to start a literal with an underscore

```
// illegal
int i = _1000;

ERROR: cannot find symbol: variable _1000
```

5. It is illegal to end a literal with an underscore.

```
// illegal
int i = 1000_;

ERROR: illegal underscore
```

6. It is also illegal to have underscore be just before or after a decimal point.

```
// illegal
double d1 = 1000_000_.0d;
double d2 = 1000_000._0d;

ERROR: illegal underscore
```

#7 Improved type inference for generic instance creation
Java 5 introduced generics which enabled developers to write type safe collections. However, generics can sometimes be too verbose. Consider the following example where we are creating a *Map* of *List* of *String*.

With Java 5 and 6

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

Note that the full type is specified twice and is therefore redundant. Unfortunately, this was a limitation of Java 5 and 6.

With Java 7
Java 7 tries to get rid of this redundancy by introducing a left to right type inference. You can now rewrite the same statement by using the <> construct.

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

This does make the code a little less verbose. You can also use <> construct when returning a newly created object.

```
public static Map<String, List<String>> parseQueryString(String queryString) {
  if (queryString == null) {
    // type is interred from the method return type
    return new HashMap<>();
  }

  // ...
}
```

This, in my opinion, goes only half way. The full solution would have been a right to left full type inference.

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

The above would have made the code even less verbose. Though this enhancement can still be done in a later version.

#8 More new I/O APIs for the Java platform (NIO.2)
A new set of I/O APIs and features were introduced in Java 1.4 under the *java.nio* package. This addition was called the **New I/O APIs**, also known as **NIO**. Naming something **New** is always short-sighted because it will not remain new forever. When the next version comes along, what should the new version be called, the **NEW NEW I/O?** Java 1.7 offers a rich set of new features and I/O capabilities, called NIO.2 (New I/O version 2?). Here are the key highlights of NIO.2.

a) Package
The most important package to look for is *java.nio.file*. This package contains many practical file utilities, new file I/O related classes and interfaces.

b) The java.nio.file.Path interface
Path is probably the new class that developers will use most often. The file referred by the path does not need to exist. The file referred to does not need to exist. For all practical purposes, you can think of replacing *java.io.File* with *java.io.Path*.

Old way

```
File file = new File("hello.txt");
System.out.println("File exists() == " + file.exists());
```

New way

```
Path path = FileSystems.getDefault().getPath("hello.txt");
System.out.println("Path exists() == " + Files.exists(path));
```

c) The java.nio.file.Files class
The *Files* class offers over 50 utility methods for File related operations which many developers would have wanted to be a part of earlier Java releases. Here are some methods to give you a sense of the range of methods offered.

- **copy()** – copy a file, with options like *REPLACE_EXISTING, NOFOLLOW_LINKS public static Path copy(Path source, Path target, CopyOption... options);*
- **move()** – move or rename a file *public static Path move(Path source, Path target, CopyOption... options);*
- **newInputStream()** – create input stream *public static InputStream newInputStream(Path path, OpenOption... options);*
- **readAllBytes()** – similar to the Apache *IOUtils.readFileToByteArray public static byte[] readAllBytes(Path path) throws IOException;*
- **createSymbolicLink()** – creates a symbolic link, if supported by the file system *public static Path createSymbolicLink(Path link, Path target, FileAttribute<?>... attrs) throws IOException*

The full list of methods can be found at: http://download.java.net/jdk7/docs/api/java/nio/file/Files.html

d) WatchService API
WatchService API is a new feature introduced in Java 1.7. It provides an API that lets you "listen" to a certain type of file system events. Your code gets called automatically when those events occur. Examples of event types are captured by *StandardWatchEventKinds* class.

- *ENTRY_CREATE*:an entry is created or renamed in the directory
- *ENTRY_DELETE*:an entry is created or renamed out of the directory
- *ENTRY_MODIFY*:a directory entry is modified

Example
Here's a full example of how to watch a directory and print any newly created files.

```
// the 'logs' directory
Path path = FileSystems.getDefault().getPath("logs");

// Create a watch service from the file system
WatchService watcher = FileSystems.getDefault().newWatchService();

// Tell me when a new entry is created in the 'logs' directory
WatchKey watchKey = path.register(watcher,
                                  StandardWatchEventKind.ENTRY_CREATE);

// now we wait to be called back
for (;;) {
  // take() blocks until something happens
  WatchKey key = watcher.take();

  // see if this is what you were looking for
  if (key.equals(watchKey)) {
    // process each event
    for (WatchEvent event : key.pollEvents()) {
      // simply print the path created
      System.out.println(path + ": new file created " + event.context());
        }
  }

  // reset the key to continue receiving events
```

```
        key.reset();
    }
```

Run the above program. Then create a file '*new.txt*' in the directory '*logs*'. The program will print:

```
logs: new file created new.txt
```

Note about WatchService implementation

The implementation will take advantage of native support for file change notification when supported by the native file system, but will resort to polling otherwise.

IDE Support

Java 7 support is now available in NetBeans and IntelliJ. Eclipse does not support Java 7 yet. The upcoming release of Eclipse 3.7 will not have support for Java 7 either, but support will be added in 3.7 SR1, expected September 2011 (table 2).

	Eclipse	NetBeans	IntelliJ
Java 7 Support	*Expected Sept 2011*	Yes	Yes
IDE Version	Indigo (3.7) SR1	NetBeans IDE 7.0	IntelliJ IDEA 1.5

Table 2: Java 7 support in IDEs

Conclusion

Java 7 offers many small language enhancements and features. However, I did not find any feature as compelling as Regex enhancement in Java 1.4 or Generics, Auto-boxing or Enum enhancement in Java 1.5. I find the *try-with-resources* enhancement particularly useful and am looking forward to using it. I also look forward to using new features from the NIO.2 library. Overall, I am glad that something was released this year as opposed to releasing a monolith of changes next year.

Vineet Manohar is a programmer, entrepreneur and technology enthusiast with a total of 14 years of industry experience with large companies and small startups, including his own garage startup. Vineet graduated from Indian Institute of Technology (India) in 1997 where he won two national level programming contests. He has used Java for over 12 years and his key interests lie in large scale systems and web frameworks which has led him to TripAdvisor LLC, where he currently works as a Senior Software Engineer. You can reach him via twitter @vineetmanohar or via his blog at http://www.vineetmanohar.com.

Imprint

Publisher
Software & Support Media GmbH

Editorial Office Address
Geleitsstraße 14
60599 Frankfurt am Main
Germany
www.jaxenter.com

Editor in Chief: Sebastian Meyen
Editors: Jessica Thornsby, Claudia Fröhling
Authors: Benjamin J. Evans, Anna Kozlova, Vineet Manohar, Stuart W. Marks, Martijn Verburg, Toby Weston
Copy Editor: Claudia Fröhling, Lisa Pychlau
Creative Director: Jens Mainz
Layout: Dominique Kalbassi

Sales Clerk:
Mark Hazell
+44 (0)20 7401 4845
markh@jaxlondon.com

Exception Handling

Using Try-With-Resources in Java SE 7

Java SE 7 includes "Project Coin," a set of small enhancements to the Java programming language. One of the most significant of these enhancements is the new *try-with-resources* statement. This is a variation of the original try statement that adds the concept of a resource. In a try-with-resources statement, resources are reliably closed at the end of the statement, even under error conditions. The *try-with-resources* statement allows programs to handle errors, prevent resource leaks, and to simplify exception processing code. This article begins by examining exception handling as it stood in Java SE 6 and earlier, and how preventing resource leaks and handling exceptions made code complex and error-prone. We will next cover the basic definition of the *try-with-resources* feature and show how this simplifies the code while avoiding resource leaks. We will conclude with examples of handling multiple resources and wrapped resources using a more advanced form of the *try-with-resources* statement.

By Stuart W. Marks, Oracle Corporation

Consider the code in Listing 1, which loads properties from a properties file. It looks simple enough, but it has a bug. What's wrong with it?

Normally, this code works fine. However, if an exception is thrown from the *props.load()* call, it will terminate the execution of this method, causing the *in.close()* statement to be skipped. The *FileInputStream* will be left open. This code hasn't kept any references to the *FileInputStream*, so it is subject to garbage collection. The garbage collector will reclaim the heap memory consumed by the *FileInputStream* object, but it also contains an open file descriptor or file handle allocated from the underlying operating system. Unless special action is taken before the *FileInputStream* object is collected, the open file handle will remain open, without any possibility of ever being closed. If this happens repeatedly, the system might eventually run out of open file handles, leading to intermittent and hard-to-diagnose failures.

We will refer to objects that require a special cleanup action as *resources*, and we will refer to the failure to perform this special action as a *resource leak*. Most often, the special action is a requirement on clients of any resource objects to call a cleanup method such as *close()* when they have finished using the resource. The Java Programming Language includes a feature called *finalization*. [1] Isn't this sufficient? Finalization was intended for preventing resource leaks, and indeed *FileInputStream* has a *finalize()* method [2] that makes sure the open file handle is closed before the object is collected. This would seem to solve the problem. However, for a variety of reasons, finalization is unreliable and its use is not recommended. [3] The best way to prevent resources from leaking is for client code to perform an explicit cleanup action. In this example, the code must always make sure to call the *FileInputStream.close()* method.

The proper way to do this in Java SE 6 and earlier is to use the *finally* clause of the *try-catch-finally* statement. The code in a *finally* block is always executed, even if an exception was thrown from the *try* block. Calling *close()* from the *finally* block will ensure that the resource does not leak. This

Listing 1

```
Properties loadProperties(String filename) throws IOException {
    Properties props = new Properties();
    FileInputStream in = new FileInputStream(filename);
    props.load(in);
    in.close();
    return props;
}
```

Listing 2

```
Properties loadProperties(String filename) throws IOException {
    Properties props = new Properties();
    FileInputStream in = new FileInputStream(filename);
    try {
        props.load(in);
    } finally {
        in.close();
    }
    return props;
}
```

technique is shown in Listing 2. The cost is a few extra lines of code and a level of nesting, but this is necessary in order to avoid resource leaks.

Handling Exceptions

Each of the *FileInputStream* constructor, the *Properties.load()* method, and the *FileInputStream.close()* method can throw *IOException*. Since our code doesn't handle this exception, we have included a *throws IOException* clause in our method declaration. This requires callers to handle these exceptions, which may be inconvenient. Instead of propagating *IOException* to callers, suppose that we want our method to handle these exceptions and do something reasonable such as returning *null*. We can do this by adding a *try-catch* statement with a *catch* block that catches *IOException* and returns *null*. This code is shown in Listing 3.

This code uses two separate *try* statements: an outer *try-catch* statement to handle exceptions, and an inner *try-finally* statement to ensure that the *FileInputStream* is closed. Note that it isn't possible to merge these into a single *try* statement. The *finally* block contains a call to *close()*, which can throw *IOException*. A *catch* block attached to a *try* statement cannot catch exceptions from a *finally* block attached to the same *try* statement. It's tempting to try to write this code using a single *try* statement, but doing so will inevitably lead to bugs or errors.

This code isn't terribly bad, but the nested *try* statements tend to obscure the actual work being performed by the method: half of the lines in the method are dedicated to error checking. This is the best we can do, until we get to Java SE 7.

The New Try-With-Resources Statement

Java SE 7 introduces an enhanced version of the *try* statement called *try-with-resources*. The purpose of the *try-with-resources* statement is to ensure that resources are always closed properly at the end of the statement. There is a new syntax that allows the programmer to declare one or more *resource variables*. Each resource variable must be of a reference type that implements the new *java.lang.AutoCloseable* interface. This interface consists of a single *close()* method. It is similar to the existing *java.io.Closeable* interface, which also simply defines a *close()* method. The main difference is that their *close()* methods have slightly different exception signatures.

AutoCloseable has been retrofitted to be a superinterface of *Closeable*. The *InputStream* and *OutputStream* class families, the *Reader* and *Writer* families, Jar- and Zip-related classes, NIO channels, and JDBC *Statement* and *RowSet* interfaces already implement *Closeable* and thus now also implement *AutoCloseable*, enabling one to use objects of any of these types as resources in a *try-with-resources* statement.

The *try-with-resources* statement is defined in terms of expansion into ordinary *try-catch-finally* statements. *Try-with-resources* achieves its purpose by expanding into a form that includes a *finally* block that automatically calls *close()* on each resource variable that has been successfully initialized to a non-null value. Although one might be tempted to dismiss *try-with-resources* as merely "syntactic sugar," it is of great value because it relieves programmers of the burden of writing tedious and error-prone exception handling code, as shown in the previous section.

The simple form of *try-with-resources* contains only a *try* block. Unlike the existing *try* statement, it is legal for a *try-with-resources* statement to omit both the *catch* and *finally* blocks. The expansion of the simple form of *try-with-resources* is shown in Listing 4.

The more complex form of the *try-with-resources* statement includes one or more *catch* blocks and optionally a *finally* block. When this form of *try-with-resources* is expanded, the statements from the *try* block are first expanded into a *try-finally* statement as shown in Listing 4. Then, the *catch* and *finally* blocks are attached to a new *try-catch-finally* statement that is wrapped around the earlier *try-finally* statement. This expansion is shown in Listing 5. This looks quite complicated, but it turns out to be exactly right for many common cases, as we will see shortly.

Listing 3

```
Properties loadProperties(String filename) {
    Properties props = new Properties();
    try {
        FileInputStream in = new FileInputStream(filename);
        try {
            props.load(in);
        } finally {
            in.close();
        }
    } catch (IOException ioe) {
        return null;
    }
    return props;
}
```

Listing 4

```
try (Resource r = new Resource()) {
    process(r);
}
// EXPANDS TO
Resource r = new Resource();
try {
    process(r);
} finally {
    r.close();
}
```

Listing 5

```
try (Resource r = new Resource()) {
    normalProcessing(r);
} catch (ExceptionType e) {
    exceptionProcessing();
} finally {
    finallyProcessing();
}
// EXPANDS TO
try {
    Resource r = new Resource();
    try {
        normalProcessing(r);
    } finally {
        r.close();
    }
} catch (ExceptionType e) {
    exceptionProcessing();
} finally {
    finallyProcessing();
}
```

(Note that this expansion is simplified from the actual definition. In the actual expansion's *finally* block, there is logic to close the resource only if it is not *null*. This is useful for resources that are initialized by methods such as *Class.getResourceAsStream()* that return *null* to indicate an error. In addition, there is code to handle *suppressed* exceptions. Ordinarily, an exception thrown from a *finally* block would cause a pending exception from the *try* block to be discarded. The expansion of *try-with-resources* includes extra logic so that an exception from the generated *finally* block is added to the *suppressed exception list* of the pending exception, allowing it to be propagated instead. See the proposed specification [4] for details.)

Observe that the structure of the expansion in Listing 5 is exactly the same as the structure of the code that we ended up with in Listing 3. We can therefore simplify the code from Listing 3 using the *try-with-resources* statement. This simplification is shown in Listing 6. This code is clearer and more concise than the earlier code, but its semantics are essentially unchanged.

Note that the *catch* block covers any exceptions that might be thrown by the resource variable initializer, by the main code in the *try* block, and by the *close()* call in the *finally* block added by the expansion. However, the resource variable itself is not in scope within the *catch* or *finally* blocks.

Multiple Resources

It's quite common for a section of code to deal with multiple resources simultaneously, for example, to copy data from one stream to another. Using only Java SE 6 constructs, this would require as many *try* statements as there are resources, in order to ensure that each resource can be closed properly without affecting the others. The *try-with-resources* statement handles multiple resources gracefully, by allowing declaration and initialization of multiple resources in a single statement. This form is expanded automatically to a series of nested *try-finally* statements. An example of *try-with-resources* using multiple resource variables is shown in Listing 7, and its expanded form is shown in Listing 8. This expansion provides an enormous convenience to code that handles multiple resources. Resources are closed in the opposite order from which they are initialized, which is usually the desired behavior. In addition, if an exception occurs after initialization of some, but not all of the resources, only the resources that have been initialized are closed.

Listing 6

```
Properties loadProperties(String filename) {
    Properties props = new Properties();
    try (FileInputStream in = new FileInputStream(filename)) {
        props.load(in);
    } catch (IOException ioe) {
        return null;
    }
    return props;
}
```

Wrapped Resources

The semantics that *try-with-resources* provides for multiple resource variables is especially significant when they are applied to wrapped resources. The design of the *java.io* libraries encourages the use of wrapped resources. For example, converting bytes to characters is accomplished by wrapping a *Reader* around an *InputStream*, and buffering is added by wrapping a *BufferedReader* around the *Reader*. The common idiom for creating these wrapped objects is to use nested constructor calls. Let's consider an example method that takes a URL, opens an *InputStream* from this URL, and reads characters using a named character set, with buffering. A straightforward way to do this using *try-with-resources* is shown in Listing 9.

In this example, the only resource variable is *br*. A call to *br.close()* will automatically be inserted within a *finally* block after the processing step. This works because the semantics of *close()* on a wrapper resource are to close any wrapped resources as well. When processing completes normally, or if an exception occurs during the processing step, *br.close()* will be called. This will close the underlying *InputStreamReader* as well as the *InputStream* that it contains.

Unfortunately, there is still a bug in this code. It is also possible for exceptions to be thrown during the initialization step. If the requested character set name *csName* is invalid, the *InputStreamReader* constructor will throw *UnsupportedEncodingException*. This is a subclass of *IOException* and so it will be caught by the *catch* block. However, since *br* has not been successfully initialized, *br.close()* will not be called. As a result, the *InputStream* returned from *url.openStream()* will be left open with no references to it, and we will have a resource leak.

The problem here is that we are creating a resource by calling *url.openStream()*, but we are not storing a reference to it in a resource variable. If a reference to the resource has not

Listing 7

```
try (Resource1 r1 = new Resource1();
     Resource2 r2 = new Resource2();
     Resource3 r3 = new Resource3())
{
    normalProcessing(r1, r2, r3);
} catch (ExceptionType e) {
    exceptionProcessing();
} finally {
    finallyProcessing();
}
```

Listing 8

```
try {
    Resource1 r1 = new Resource1();
    try {
        Resource2 r2 = new Resource2();
        try {
            Resource3 r3 = new Resource3();
            try {
                normalProcessing(r1, r2, r3);
            } finally {
                r3.close();
            }
        } finally {
            r2.close();
        }
    } finally {
        r1.close();
    }
} catch (ExceptionType e) {
    exceptionProcessing();
} finally {
    finallyProcessing();
}
```

been stored in a resource variable, the *try-with-resources* statement cannot ensure that it is closed. The same thing can happen with the *InputStreamReader* and the *BufferedReader*. If the *BufferedReader* constructor were to throw an exception (admittedly, this seems unlikely) it will leave unreferenced, open instances of *InputStreamReader* and *InputStream*, again resulting in a resource leak.

The solution to this problem is to unfold the initialization so that each resource is stored in its own resource variable. Earlier resource variables are in scope and can be used in the initialization of later resource variables, enabling us to declare and initialize the resources from the inside out. By assigning each resource to its own resource variable, we can ensure that if an exception were to occur during the initialization of one of the resources, the previously initialized resources will be closed. The resulting code is shown in Listing 10. This code properly handles exceptions that occur any time during initialization, processing, and the subsequent *close()* calls, and it avoids resource leaks in all cases.

> This code is clearer than the earlier code, but its semantics are essentially unchanged.

Observant readers will notice that *close()* is called more than once for the wrapped resources. In the expansion of the above code, the *try-with-resources* statement automatically generates calls to *close()* on each of *br*, *isr*, and *is* because they are declared as individual resource variables. As noted above, closing a wrapper object is defined to close wrapped objects as well. Thus, *br.close()* will call *isr.close()*, which in turn will call *is.close()*. This results in multiple, redundant calls to *isr.close()* and *is.close()*. This is not a problem. These classes all implement the *Closeable* interface, whose definition requires that the second and subsequent calls to *close()* have no effect. Note that this applies to *Closeable* resources but not necessarily all *AutoCloseable* resources. Fortunately, the *java.io* classes that are likely to be wrapped all implement the *Closeable* interface.

Conclusion

The new *try-with-resources* statement added in Java SE 7 addresses several common problems with Java exception processing and with resource leaks. Exception processing has historically been very inconvenient in Java. Simple code often mishandled exceptions or leaked resources. On the other hand, avoiding resource leaks and properly handling all exceptions resulted in tedious, complex code. The new *try-with-resources* statement enables one to write code that handles all exceptions and that prevents resource leaks, while keeping the code simple and straightforward.

If you would like to try out *try-with-resources* or any other new features of Java SE 7, you can download Oracle's implementation, documentation, and source code from [5].

Listing 9

```
void readFromUrl(URL url, String csName) {
    try (BufferedReader br = new BufferedReader(
            new InputStreamReader(url.openStream(), csName)))
    {
        normalProcessing(br);
    } catch (IOException ioe) {
        exceptionProcessing();
    }
}
```

Listing 10

```
void readFromUrl(URL url, String csName) {
    try (InputStream is = url.openStream();
         InputStreamReader isr = new InputStreamReader(is, csName);
         BufferedReader br = new BufferedReader(isr))
    {
        normalProcessing(br);
    } catch (IOException ioe) {
        exceptionProcessing();
    }
}
```

Stuart Marks is a Principal Member of Technical Staff in the Java Platform Group at Oracle Corporation. He is currently working on improving the core libraries of the JDK. He has previously worked on JavaFX and Java ME at Sun Microsystems. He has over twenty years of software platform product development experience in the areas of window systems, interactive graphics, and mobile and embedded systems. Stuart holds a Masters degree in Computer Science from Stanford University.

References

[1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. *The Java Language Specification, Third Edition*. Section 12.6, "Finalization of Class Instances," p. 325., http://java.sun.com/docs/books/jls/index.html

[2] Implementation of the FileInputStream class in jdk/src/share/classes/java/io/FileInputStream.java. Available online at http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/jdk7-b144/src/share/classes/java/io/FileInputStream.java among other locations.

[3] Joshua Bloch. *Effective Java, Second Edition*. Item 7: "Avoid Finalizers," pp. 27ff.

[4] *JSR 334: Small Enhancements to the Java Programming Language*. Public review draft specification, v0.875, 23 March 2011, the "try-with-resources statement" section.

[5] JDK 7 Project website. JDK 7 is not complete at the time of this writing. Preview builds, documentation, and sources can be downloaded from here: http://jdk7.java.net/