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