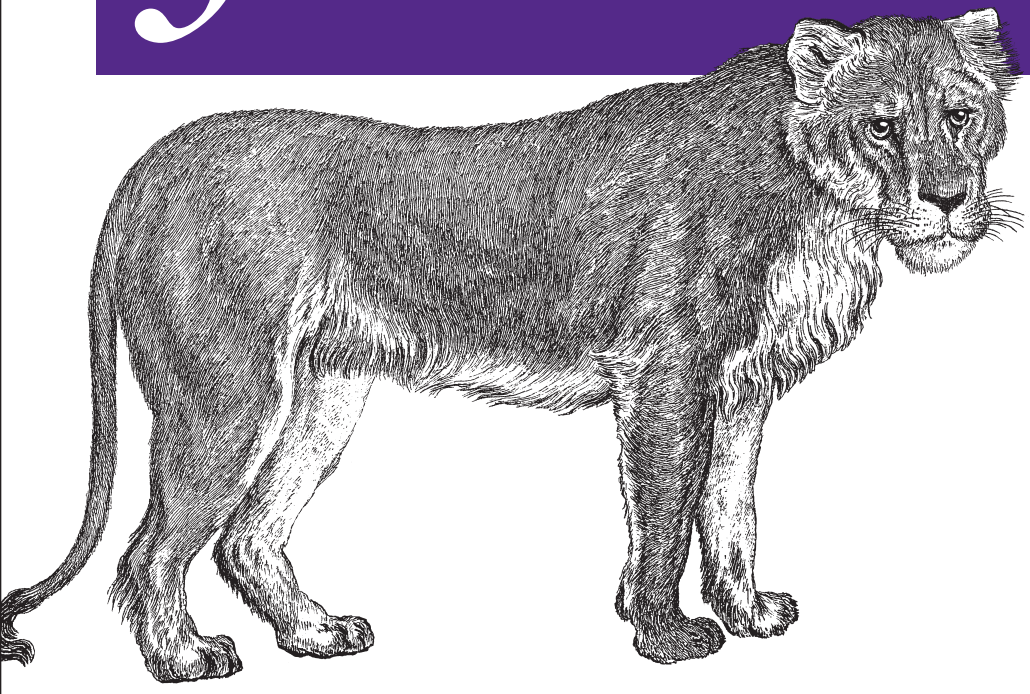
What's New in

Java 7



O'REILLY®

Madhusudhan Konda

What's New in Java 7

Madhusudhan Konda

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

What's New in Java 7
by Madhusudhan Konda

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Revision History for the :

2011-10-21 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449314392* for release details.

ISBN: 978-1-449-31439-2
1319127844

Table of Contents

What's New in Java 7

A survey of important updates and changes in Java 7

By Madhusudhan Konda

There are a number of features in Java 7 that will please developers. Things such as strings in switch statements, multi-catch exception handling, try-with-resource statements, the new File System API, extensions of the JVM, support for dynamically-typed languages, the fork and join framework for task parallelism, and a few others will certainly be embraced by the community.

Below I outline the features and provide examples where appropriate. A zip file containing code snippets used in this post can be downloaded here.

Language enhancements

Java 7 includes a few new language features via Project Coin. These features are quite handy for a developer.

Diamond Operator

You may have noted on many occasions your IDE complaining of types when working with Generics. For example, if we have to declare a map of trades using Generics, we write the code as follows:

```
Map<String, List<Trade>> trades = new TreeMap<String, List<Trade>> ();
```

The not-so-nice thing about this declaration is that we must declare the types on both the sides, although the right-hand side seems a bit redundant. Can the compiler infer the types by looking at the left-hand-side declaration? Not unless you're using Java 7. In 7, it's written like this:

```
Map<String, List<Trade>> trades = new TreeMap <> ();
```

How cool is that? You don't have to type the whole list of types for the instantiation. Instead you use the <> symbol, which is called diamond operator. Note that while not declaring the diamond operator is legal, as `trades = new TreeMap ()`, it will make the compiler generate a couple of type-safety warnings.

Using strings in switch statements

Switch statements work either with primitive types or enumerated types. Java 7 introduced another type that we can use in Switch statements: the `String` type.

Say we have a requirement to process a Trade based on its status. Until now we used to do this by using if-else statements.

```
private void processTrade(Trade t) {
            String status = t.getStatus();
            if (status.equalsIgnoreCase(NEW)) {
                  newTrade(t);
            } else if (status.equalsIgnoreCase(EXECUTE)) {
                  executeTrade(t);
            } else if (status.equalsIgnoreCase(PENDING)) {
                  pendingTrade(t);
            }
}
```

This method of working on strings is crude. In Java 7, we can improve the program by utilizing the enhanced Switch statement, which takes a `String` type as an argument.

```
public void processTrade(Trade t) {
          String status = t.getStatus();

          switch (status) {
          case NEW:
               newTrade(t);
               break;
          case EXECUTE:
               executeTrade(t);
               break;
          case PENDING:
               pendingTrade(t);
               break;

          default:
               break;
          }
     }
```

In the above program, the status field is always compared against the case label by using the `String.equals()` method.

Automatic resource management

Resources such as `Connections`, `Files`, `Input/OutStreams`, etc. should be closed manually by the developer by writing bog-standard code. Usually we use a `try-finally` block to close the respective resources. See the current practice of creating a resource, using it and finally closing it:

```
public void oldTry() {
            try {
                  fos = new FileOutputStream("movies.txt");
                  dos = new DataOutputStream(fos);
                  dos.writeUTF("Java 7 Block Buster");
            } catch (IOException e) {
                  e.printStackTrace();
            } finally {
                  try {
                        fos.close();
                        dos.close();
                  } catch (IOException e) {
                        // log the exception
                  }
            }
      }
```

However, Java 7 has introduced another cool feature to manage the resources automatically. It is simple in operation, too. All we have to do is declare the resources in the `try` as follows:

```
try(resources_to_be_cleant){
   // your code
}
```

The above method with the old try can finally can be re-written using this new feature as shown below:

```
public void newTry() {

      try (FileOutputStream fos = new FileOutputStream("movies.txt");
                  DataOutputStream dos = new DataOutputStream(fos)) {
            dos.writeUTF("Java 7 Block Buster");
      } catch (IOException e) {
            // log the exception
      }
}
```

The above code also represents another aspect of this feature: working with multiple resources. The `FileOutputStream` and `DataOutputStream` resources are enclosed in the try statement one after the other, each one separated by a semicolon (;) separator. We do not have to nullify or close the streams manually, as they are closed automatically once the control exists the try block.

Behind the scenes, the resources that should be auto closed must implement `java.lang.AutoCloseable` interface.

Any resource that implements `AutoCloseble` interface can be a candidate for automatic resource management. The `AutoCloseable` is the parent of `java.io.Closeable` interface and has just one method `close()` that would be called by the JVM when the control comes out of the try block.

Numeric literals with underscores

Numerical literals are definitely eye strainers. I am sure you would start counting the zeroes like me if you've been given a number with, say, ten zeros. It's quite error prone and cumbersome to identify a literal if it's a million or a billion unless you count the places from right to left. Not anymore. Java 7 introduced underscores in identifying the places. For example, you can declare 1000 as shown below:

```
int thousand =  1_000;
```

or 1000000 (one million) as follows

```
int million  =  1_000_000
```

Note that *binary* literals are also introduced in this release too — for example "0b1" — so developers don't have to convert them to hexadecimals any more.

Improved exception handling

There are a couple of improvements in the exception handling area. Java 7 introduced multi-catch functionality to catch multiple exception types using a single catch block.

Let's say you have a method that throws three exceptions. In the current state, you would deal them individually as shown in below:

```
public void oldMultiCatch() {
        try {
             methodThatThrowsThreeExceptions();
        } catch (ExceptionOne e) {
             // log and deal with ExceptionOne
        } catch (ExceptionTwo e) {
             // log and deal with ExceptionTwo
        } catch (ExceptionThree e) {
             // log and deal with ExceptionThree
        }
}
```

Catching an endless number of exceptions one after the other in a catch block looks cluttered. And I have seen code that catches a dozen exceptions, too. This is incredibly inefficient and error prone. Java 7 has brought in a new

language change to address this ugly duckling. See the improved version of the method `oldMultiCatch` method below:

```
public void newMultiCatch() {
      try {
            methodThatThrowsThreeExceptions();
      } catch (ExceptionOne | ExceptionTwo | ExceptionThree e) {
            // log and deal with all Exceptions
      }
}
```

The multiple exceptions are caught in one catch block by using a '|' operator. This way, you do not have to write dozens of exception catches. However, if you have bunch of exceptions that belong to different types, then you could use "*multi multi-catch*" blocks too. The following snippet illustrates this:

```
public void newMultiMultiCatch() {
            try {
                  methodThatThrowsThreeExceptions();
            } catch (ExceptionOne e) {
                  // log and deal with ExceptionOne

            } catch (ExceptionTwo | ExceptionThree e) {
                  // log and deal with ExceptionTwo and ExceptionThree
            }
      }
```

In the above case, the ExceptionTwo and ExceptionThree belong to a different hierarchy, so you would want to handle them differently but with a single catch block.

New file system API (NIO 2.0)

Those who worked with Java IO may still remember the headaches that framework caused. It was never easy to work seamlessly across operating systems or multi-file systems. There were methods such as delete or rename that behaved unexpected in most cases. Working with symbolic links was another issue. In an essence, the API needed an overhaul.

With the intention of solving the above problems with Java IO, Java 7 introduced an overhauled and in many cases new API.

The NIO 2.0 has come forward with many enhancements. It's also introduced new classes to ease the life of a developer when working with multiple file systems.

Working with Path

A new `java.nio.file` package consists of classes and interfaces such as `Path`, `Paths`, `FileSystem`, `FileSystems` and others.

A `Path` is simply a reference to a file path. It is the equivalent (and with more features) to `java.io.File`. The following snippet shows how to obtain a path reference to the "temp" folder:

```
public void pathInfo() {
            Path path = Paths.get("c:\\Temp\\temp");
System.out.println("Number of Nodes:" + path.getNameCount());
            System.out.println("File Name:" + path.getFileName());
            System.out.println("File Root:" + path.getRoot());
            System.out.println("File Parent:" + path.getParent());
      }
```

The console output would be:

```
Number of Nodes:2

File Name:temp.txt

File Root:c:\

File Parent:c:\Temp
```

Deleting a file or directory is as simple as invoking a delete method on Files (note the plural) class. The `Files` class exposes two `delete` methods, one that throws `NoSuchFileException` and the other that does not.

The following delete method invocation throws `NoSuchFileException`, so you have to handle it:

```
Files.delete(path);
```

Where as `Files.deleteIfExists(path)` does not throw exception (as expected) if the file/directory does not exist.

You can use other utility methods such as `Files.copy(..)` and `Files.move(..)` to act on a file system efficiently. Similarly, use the `createSymbolicLink(..)` method to create symbolic links using your code.

File change notifications

One of my favorite improvements in the JDK 7 release is the addition of File Change Notifications. This has been a long-awaited feature that's finally carved into NIO 2.0. The `WatchService` API lets you receive notification events upon changes to the subject (directory or file).

The steps involved in implementing the API are:

- Create a `WatchService`. This service consists of a queue to hold `WatchKeys`
- Register the directory/file you wish to monitor with this `WatchService`
- While registering, specify the types of events you wish to receive (create, modify or delete events)
- You have to start an infinite loop to listen to events
- When an event occurs, a `WatchKey` is placed into the queue
- Consume the `WatchKey` and invoke queries on it

Let's follow this via an example. We create a `DirPolice` Java program whose responsibility is to police a particular directory. The steps are provided below:

1. Creating a `WatchService` object:

```
WatchService  watchService = FileSystems.getDefault().newWatchService();
```

2. Obtain a path reference to your watchable directory. I suggest you parameterize this directory so you don't hard code the file name.

```
path = Paths.get("C:\\Temp\\temp\\");
```

3. The next step is to register the directory with the `WatchService` for all types of events:

```
dirToWatch.register(watchService, ENTRY_CREATE, ENTRY_MODIFY,
ENTRY_DELETE);
```

These are `java.nio.file.StandardWatchEventKinds` event types

4. Initiate the infinite loop and start taking the events:

```
while(true)
{
    WatchKey key = watchService.take(); // this would return you keys
    ...
}
```

5. Run through the events on the key:

```
for (WatchEvent<?> event : key.pollEvents()) {
            Kind<?> kind = event.kind();
System.out.println("Event on " + event.context().toString() + " is " + kind);
}
```

For example, if you modify or delete the temp directory, you would see statement as shown below on the console respectively:

```
Event on temp is ENTRY_MODIFY

Event on temp is ENTRY_DELETE
```

The relevant methods of the `DirPolice` source code are posted below (download the full source code):

```
/**
 * This initiates the police
 */
private void init() {
        path = Paths.get("C:\\Temp\\temp\\");
        try {
              watchService = FileSystems.getDefault().newWatchService();
              path.register(watchService, ENTRY_CREATE, ENTRY_DELETE,
                          ENTRY_MODIFY);
        } catch (IOException e) {
              System.out.println("IOException"+ e.getMessage());
        }
}
/**
 * The police will start making rounds
 */
private void doRounds() {
        WatchKey key = null;
        while(true) {
              try {
                    key = watchService.take();
                    for (WatchEvent<?> event : key.pollEvents()) {
                          Kind<?> kind = event.kind();
System.out.println("Event on " + event.context().toString() + " is " + kind);
                    }
            }  catch (InterruptedException e) {
System.out.println("InterruptedException: "+e.getMessage());
            }
            boolean reset = key.reset();
            if(!reset)
                  break;
      }
}
```

Fork and Join

The effective use of parallel cores in a Java program has always been a challenge. There were few home-grown frameworks that would distribute the work across multiple cores and then join them to return the result set. Java 7 has incorporated this feature as a Fork and Join framework.

Basically the Fork-Join breaks the task at hand into mini-tasks until the mini-task is simple enough that it can be solved without further breakups. It's like a divide-and-conquer algorithm. One important concept to note in this framework is that ideally no worker thread is idle. They implement a work-stealing algorithm in that idle workers "steal" the work from those workers who are busy.

The core classes supporting the Fork-Join mechanism are `ForkJoinPool` and `ForkJoinTask`. The `ForkJoinPool` is basically a specialized implementation of

`ExecutorService` implementing the *work-stealing* algorithm we talked about above.

We create an instance of `ForkJoinPool` by providing the target parallelism level — the number of processors as shown below:

```
ForkJoinPool pool = new ForkJoinPool(numberOfProcessors)
```

Where numberOfProcessors = Runtime.getRunTime().availableProcessors();

However, the default `ForkJoinPool` instantiation would set the parallelism level equal to the same number obtained as above.

The problem that needs to be solved is coded in a `ForkJoinTask`. However, there are two implementations of this class out of the box: the `RecursiveAction` and `RecursiveTask`. The only difference between these two classes is that the former one does not return a value while the latter returns an object of specified type.

Here's how to create a `RecursiveAction` or `RecursiveTask` class that represents your requirement problem (I use the `RecursiveAction` class):

```
public class MyBigProblemTask extends RecursiveAction {

    @Override
    protected void compute() {

        . . . // your problem invocation goes here
    }
}
```

You have to override the compute method where in you need to provide the computing functionality. Now, provide this `ForkJoinTask` to the `Executor` by calling invoke method on the `ForkJoinPool`:

```
pool.invoke(task);
```

Supporting dynamism

Java is a statically typed language — the type checking of the variables, methods and return values is performed at compile time. The JVM executes this strongly-typed bytecode at runtime without having to worry about finding the type information.

There's another breed of typed languages — the dynamically typed languages. Ruby, Python and Clojure are in this category. The type information is unresolved until runtime in these languages. This is not possible in Java as it would not have any necessary type information.

There is an increasing pressure on Java folks improvise running the dynamic languages efficiently. Although it is possible to run these languages on a JVM (using `Reflection`), it's not without constraints and restrictions.

In Java 7, a new feature called invokedynamic was introduced. This makes VM changes to incorporate non-Java language requirements. A new package, `java.lang.invoke`, consisting of classes such as `MethodHandle`, `CallSite` and others, has been created to extend the support of dynamic languages.

Wrapping up

As we've covered, Java 7 has a few bells and whistles that should put smiles on developers' faces, and the open-source collaboration and support for dynamic languages via JVM extensions should also be well received by those outside the Java community.