





**MOVING JAVA
FORWARD**

ORACLE®

Coin In Action

Using Java 7 Features in Real Code

Stuart W. Marks — Oracle JDK Core Libraries Group

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.



Latin America 2011

December 6–8, 2011

Tokyo 2012

April 4–6, 2012

Oracle OpenWorld Bookstore

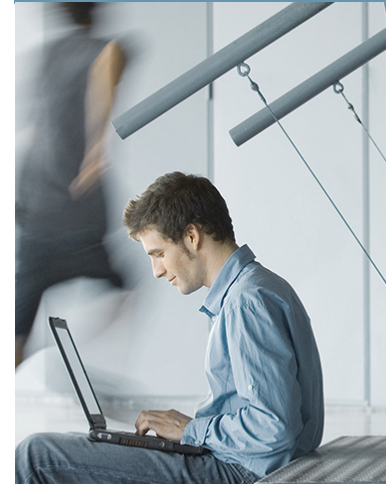
- Visit the Oracle OpenWorld Bookstore for a fabulous selection of books on many of the conference topics and more!
- Bookstore located at Moscone West, Level 2
- All Books at 20% Discount

DigitalGuru
Technical Bookshop



Program Agenda

- Project Coin
 - Six small language changes for Java 7
 - Two Coin (plus one NIO feature) covered today
 - New feature details
- URLJarFile example
 - Review of pre-existing code
 - Applying changes
 - Before-and-after comparison



Project Coin Features

1. Diamond
2. Try-with-resources
3. Multi-catch with more precise rethrow
4. Enhanced integer literals
5. Strings in switch
6. Safe varargs

Project Coin Features Demonstrated Today

1. Diamond
2. Try-with-resources
3. Multi-catch with more precise rethrow
4. Enhanced integer literals
5. Strings in switch
6. Safe varargs
7. NIO.2 File Utilities

Bonus! This isn't actually a Project Coin feature, but it was too good to pass up.

Try-With-Resources

- A variation of the try-catch-finally statement
- Allows initialization of a **resource variable**
 - Must be of type `AutoCloseable`
 - Its `close()` method is called from a generated finally-block
 - Special handling for exceptions thrown by `close()`
- Useful for avoiding leaks of external objects
 - Files, channels, sockets, SQL statements, ...
 - Many JDK classes retrofitted to be `AutoCloseable`

Try-With-Resources

You type this:

```
try (Resource r = ...) {  
    ...  
} catch (Exception e) {  
    ...  
} finally {  
    ...  
}
```

Compiler generates this:

```
try {  
    Resource r = null;  
    try {  
        r = ...;  
        ...  
    } finally {  
        if (r != null)  
            r.close();  
    }  
} catch (Exception e) {  
    ...  
} finally {  
    ...  
}
```

Actually, it's more complicated because of the way exceptions from close() are handled.

Multi-Catch and Precise Rethrow

- Java's checked exceptions must either:
 - Be handled by a ***catch*** clause; or
 - Be declared in the ***throws*** clause of the containing method.
- Where do checked exceptions come from?
 - The ***throw*** statement
 - The ***throws*** clause of called methods

Multi-Catch and Precise Rethrow

- When a caught exception is rethrown, what must appear in the ***throws*** clause of the containing method?
- Java 6 and earlier:
 - the declared type of the exception variable
- Java 7 and later:
 - ***If*** the exception variable is effectively final (not assigned),
 - Only the checked exceptions that can be thrown by the try-block need to appear in the ***throws*** clause

Multi-Catch and Precise Rethrow

```
void exampleMethod(Future future) throws  
    InterruptedException, ExecutionException, TimeoutException  
{  
    Object result = future.get(5, SECONDS);  
}
```

```
// Future.get(long, TimeUnit) is declared with:  
//     throws InterruptedException, ExecutionException,  
//         TimeoutException  
// this stuff is from java.util.concurrent
```

How would we catch, clean up, and rethrow?

Multi-Catch and Precise Rethrow

```
void exampleMethod(Future future) throws
    InterruptedException, ExecutionException, TimeoutException
{
    try {
        Object result = future.get(5, SECONDS);
    } catch (InterruptedException ex) {
        cleanup();
        throw ex;
    } catch (ExecutionException ex) {
        cleanup();
        throw ex;
    } catch (TimeoutException ex) {
        cleanup();
        throw ex;
    }
}
```

Java 6: multiple catch clauses

Multi-Catch and Precise Rethrow

```
void exampleMethod(Future future) throws
    Exception
{
    try {
        Object result = future.get(5, SECONDS);
    } catch (Exception ex) {
        cleanup();
        throw ex;
    }
}
```

Java 6: catch “wider” exception type (considered poor style)

Multi-Catch and Precise Rethrow

```
void exampleMethod(Future future) throws
    InterruptedException, ExecutionException, TimeoutException
{
    try {
        Object result = future.get(5, SECONDS);
    } catch (InterruptedException|ExecutionException|
        TimeoutException ex) {
        cleanup();
        throw ex;
    }
}
```

Java 7: multi-catch



Multi-Catch and Precise Rethrow

```
void exampleMethod(Future future) throws
    InterruptedException, ExecutionException, TimeoutException
{
    try {
        Object result = future.get(5, SECONDS);
    } catch (Exception ex) {
        cleanup();
        throw ex;
    }
}
```

Java 7: precise rethrow (is this good style now?)

NIO.2

- NIO.2 Big Features
 - Asynchronous I/O
 - Filesystem API
- NIO.2 Conveniences
 - Path interface, Paths and Files utility classes
 - Access to file permissions, attributes, symbolic links
 - Files.walkFileTree(), Files.readAllLines()
 - Files.copy() – various flavors of copying all bytes

Scenario – URLJarFile.java

- “Based on a true story”
- My JDK 7 task: apply Project Coin features to the JDK
 - Concentrated mostly on core libraries
- Bug 7018392
 - “update URLJarFile.java to use try-with-resources”
http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7018392
 - Change has been integrated into JDK 7
<http://hg.openjdk.java.net/jdk7/jdk7/jdk/rev/6e33b377aa6e>
 - Simplified here for clarity of presentation

Requirements for `URLJarFile.retrieve()`

- Given a URL ...
 - Open it
 - Download contents into a temporary file
 - Create and return a `JarFile` instance backed by that temp file
 - Remove temp file if there was an error
 - Don't leak anything
 - Handle all errors without loss of information

Original Code

```
JarFile retrieve(URL url) throws IOException {
    InputStream in = url.openStream();
    OutputStream out = null;
    File tmpFile = null;
    try {
        tmpFile = File.createTempFile("jar_cache", null);
        out = new FileOutputStream(tmpFile);
        int read = 0;
        byte[] buf = new byte[BUF_SIZE];
        while ((read = in.read(buf)) != -1) {
            out.write(buf, 0, read);
        }
        out.close();
        out = null;
        return new JarFile(tmpFile);
    } catch (IOException e) {
        if (tmpFile != null) {
            tmpFile.delete();
        }
        throw e;
    } finally {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    }
}
```

DEMO

Before vs After

```
JarFile retrieve(URL url) throws IOException {
    InputStream in = url.openStream();
    OutputStream out = null;
    File tmpFile = null;
    try {
        tmpFile = File.createTempFile("jar_cache", null);
        out = new FileOutputStream(tmpFile);
        int read = 0;
        byte[] buf = new byte[BUF_SIZE];
        while ((read = in.read(buf)) != -1) {
            out.write(buf, 0, read);
        }
        out.close();
        out = null;
        return new JarFile(tmpFile);
    } catch (IOException e) {
        if (tmpFile != null) {
            tmpFile.delete();
        }
        throw e;
    } finally {
        if (in != null) {
            in.close();
        }
        if (out != null) {
            out.close();
        }
    }
}
```

```
JarFile retrieve(URL url) throws IOException {
    Path tmpFile = Files.createTempFile("jar_cache", null);
    try (InputStream in = url.openStream()) {
        Files.copy(in, tmpFile, REPLACE_EXISTING);
        return new JarFile(tmpFile.toFile());
    } catch (Throwable t) {
        try {
            Files.delete(tmpFile);
        } catch (Throwable t2) {
            t.addSuppressed(t2);
        }
        throw t;
    }
}
```

Summary

- Java 7 Features Demonstrated
 - NIO.2 Files utilities
 - Try-with-resources
 - Multi-catch and precise rethrow
- Benefits
 - Code gets ***more concise, more correct, more robust***

What You Should Do Next

- Learn more about Java 7
 - There's lots of stuff I haven't mentioned today
- Documentation
 - <http://download.oracle.com/javase/7/docs/index.html>
- Download JDK 7
 - <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- View, download, build OpenJDK source code
 - <http://openjdk.java.net/>

Q&A



MOVING JAVA FORWARD

ORACLE®





BACKUP SLIDES

Original Code (Part 1 of 3)

```
JarFile retrieve(URL url) throws IOException {
    InputStream in = url.openStream();
    OutputStream out = null;
    File tmpFile = null;
    try {
        tmpFile = File.createTempFile("jar_cache", null);
        out = new FileOutputStream(tmpFile);
        ...
    }
}
```

Original Code (Part 2 of 3)

```
...
int read = 0;
byte[] buf = new byte[BUF_SIZE];
while ((read = in.read(buf)) != -1) {
    out.write(buf, 0, read);
}
out.close();
out = null;
return new JarFile(tmpFile);
...
```

Original Code (Part 3 of 3)

```
    ...  
} catch (IOException e) {  
    if (tmpFile != null) {  
        tmpFile.delete();  
    }  
    throw e;  
} finally {  
    if (in != null) {  
        in.close();  
    }  
    if (out != null) {  
        out.close();  
    }  
}  
}
```

Code Review

- Has bugs!
 - If `in.close()` fails, ***out*** will remain open
 - If non-IOException is thrown, temp file will not be deleted
 - Suppressed exceptions are mishandled
- Other Issues
 - Uses null references to keep track of what needs cleanup
 - Messy, but alternative is to use nested try-statements
... which is arguably worse
 - Pathology: trying to do too much in a single try/catch/finally block

Improvement #1 – Use NIO

- Replace copy loop with `Files.copy(InputStream, Path)`
- Add various `java.nio.file.*` imports
- Change `java.io.File` to `java.nio.file.Path`
- Call `PathToFile()` where necessary to convert back to `java.io.File`
- Get rid of `BUF_SIZE` and `OutputStream` variables
- Use `Files.copy(..., REPLACE_EXISTING)`

Improvement #1 – Use NIO

Allows us to replace this...

```
out = new FileOutputStream(tmpFile);
int read = 0;
byte[] buf = new byte[BUF_SIZE];
while ((read = in.read(buf)) != -1) {
    out.write(buf, 0, read);
}
out.close();
```

With this...

```
Files.copy(in, temp, REPLACE_EXISTING);
```

Improvement #1 – Use NIO

```
JarFile retrieve(URL url) throws IOException {
    InputStream in = url.openStream();
    Path tmpFile = null;
    try {
        tmpFile = Files.createTempFile("jar_cache", null);
        Files.copy(in, tmpFile, REPLACE_EXISTING);
        return new JarFile(tmpFile.toFile());
    } catch (IOException e) {
        if (tmpFile != null) {
            Files.delete(tmpFile);
        }
        throw e;
    } finally {
        if (in != null) {
            in.close();
        }
    }
}
```

Improvement #2 – Use Try-With-Resources

- Declares a resource variable
 - Automatically closed within a finally block
 - Ignored if null
- Suppressed exceptions from close() are added to a suppressed exception list
- Lets us drop our own finally block

Improvement #2 – Use Try-With-Resources

```
JarFile retrieve(URL url) throws IOException {
    Path tmpFile = null;
    try (InputStream in = url.openStream()) {
        tmpFile = Files.createTempFile("jar_cache", null);
        Files.copy(in, tmpFile, REPLACE_EXISTING);
        return new JarFile(tmpFile.toFile());
    } catch (IOException e) {
        if (tmpFile != null) {
            Files.delete(tmpFile);
        }
        throw e;
    }
}
```

Improvement #3 – Get Rid of Null Sentinel

- The *in* and *out* resources are handled for us now
 - *in* is a resource variable
 - *out* is buried inside of Files.copy()
- We can create the temp file first and get rid of special case null handling

Improvement #3 – Get Rid of Null Sentinel

```
JarFile retrieve(URL url) throws IOException {  
    Path tmpFile = Files.createTempFile("jar_cache", null);  
    try (InputStream in = url.openStream()) {  
        Files.copy(in, tmpFile, REPLACE_EXISTING);  
        return new JarFile(tmpFile.toFile());  
    } catch (IOException e) {  
        Files.delete(tmpFile);  
        throw e;  
    }  
}
```

Improvement #4 – Catch/Rethrow Throwable

- We want to delete the temp file on any error
 - Catch and rethrow *Throwable*
 - The method still declares *throws IOException*
 - How is this possible?
- This is the “more precise rethrow” feature of Java 7
 - If the catch block simply rethrows a caught exception,
 - The checked exceptions that *can be thrown* from the catch block are inferred from what *can be thrown* by the try block.
- ***A subtle but significant change in Java 7!***

Improvement #4 – Catch/Rethrow Throwable

```
JarFile retrieve(URL url) throws IOException {
    Path tmpFile = Files.createTempFile("jar_cache", null);
    try (InputStream in = url.openStream()) {
        Files.copy(in, tmpFile, REPLACE_EXISTING);
        return new JarFile(tmpFile.toFile());
    } catch (Throwable t) {
        Files.delete(tmpFile);
        throw t;
    }
}
```

Improvement #5 – Suppressed Exceptions

- An exception from `Files.delete()` could still suppress an earlier exception
- Add explicit code to catch them and add them to the suppressed exception list
- Code gets a bit longer but closes a big hole in exception handling

Improvement #5 – Suppressed Exceptions

```
JarFile retrieve(URL url) throws IOException {
    Path tmpFile = Files.createTempFile("jar_cache", null);
    try (InputStream in = url.openStream()) {
        Files.copy(in, tmpFile, REPLACE_EXISTING);
        return new JarFile(tmpFile.toFile());
    } catch (Throwable t) {
        try {
            Files.delete(tmpFile);
        } catch (Throwable t2) {
            t.addSuppressed(t2);
        }
        throw t;
    }
}
```