Working and unit testing with temporary files in Java
Temporary test files in unit testing
Testing in JUnit 5.x
Temporary files outside of unit test frameworks
Using the jimfs in-memory file system
Conclusion
Dig deeper

TESTING

Working and unit testing with temporary files in Java

Temporary files are frequently used in testing and in production. Here is how to create and manage—and delete—them.

by Andrew Binstock

February 26, 2021

Temporary files are underappreciated resources. They make excellent stand-ins for mocks and stubs because they're the real thing and they can be easily manipulated to produce the error conditions that you might want to simulate in test suites. In addition, testing frameworks such as Junit make it simple to create temporary files and dispose of them automatically. But as any view of the trash that collects on your system's designated temporary directory will confirm, these files have many uses outside of testing. Installation artifacts, logs, saved configurations, and many other one-off situations make short-lived files attractive.

In this article, I examine the use of temporary files in Java. Much of the emphasis is on the unique APIs for their use, especially for disposing of them after their service is complete. Because testing remains the primary use case, I start with unit tests and then move to using temporary files in non-testing contexts. To follow along, you will need a basic familiarity with Java and JUnit. Because JUnit is currently in transition between two major versions—JUnit 4.x and 5.x—I provide code for both versions.

Temporary test files in unit testing

For many years, creating files in unit tests was considered a bad practice because file creation and I/O were slow. In addition, file I/O was considered outside the scope of testing a single unit. To avoid slowing down test runs, organizations that were strongly committed to testing sequestered the file I/O outside of the continuous runs of unit tests.

The book *How Google Tests Software* (by James Whittaker, Jason Arbon, and Jeff Carollo) explains in detail how Google runs its test suites: It makes no distinction between unit tests, integration tests, and user acceptance testing when determining what to run. Rather, the principal criterion is the *speed* of the tests. Fast tests run together and frequently. Slower tests are run less frequently. Very slow tests are run even less frequently.

The idea of excluding file I/O may have been valid in the era of relatively slow hard disk drives that consisted of spinning platters, but everything changed with the advent of solid-state disks. File access is now so fast that you can safely run file tests with unit tests. As you shall shortly see, JUnit has specific provisions for doing just that since version 4.7.

Let's look at some JUnit 4.13 code for creating a temporary directory and populating it with temporary files. (Please read this section even if you're using JUnit 5, since almost everything in it applies to version 5.x.)

```
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TemporaryFolder;

import static org.junit.Assert.*;

import java.io.*;

public class TempFilesJUnit4Test {

    File file1, file2;

    /* This folder and the files created in i
     * tests are run, even in the event of fa
     */
    @Rule
    public TemporaryFolder folder = new Tempo

    /* executed before every test: create tem
    @Before
    public void setUp() {
        try {
            file1 = folder.newFile( "testfile
            file2 = folder.newFile( "testfile
        }
        catch( IOException ioe ) {
            System.err.println(
                "error creating temporary tes
                this.getClass().getSimpleName
        }
    }

    /**
     *  Write to the two temporary files and
     */
    @Test
    public void test2TempFiles() {

        //write out data to the test files
        try {
```

```
FileWriter fw1 = new FileWriter( file1 );
BufferedWriter bw1 = new BufferedWriter( fw1
bw1.write( "content for file1");
bw1.close();

FileWriter fw2 = new FileWriter( file2 );
BufferedWriter bw2 = new BufferedWriter( fw2
            bw2.write( "content for file2" );
            bw2.close();
        }
        catch( IOException ioe ) {
            System.err.println(
                "error creating temporary tes
                this.getClass().getSimpleName
        }

        assertTrue( file1.exists() );
        assertTrue( file2.isFile() );

        assertEquals( file1.length(), 17L );
        assertEquals( file1.length(), file2.l

        assertTrue( file1.getAbsolutePath().e
                        "testfile1.txt" ));
    }
}
```

This code creates a temporary directory, creates two temporary files in that directory, writes to them, and then validates the operations.

The imports are mostly the familiar ones for JUnit, but with the addition of `import org.junit.rules.TemporaryFolder` which brings in the bits you'll need in a moment. In the principal testing class, immediately after the first comment block is the following rule:

```
@Rule
    public TemporaryFolder folder = new Tempo
```

This rule tells JUnit to create a temporary folder. Specifically, because the rule is annotated with `@Rule`, it will be run before each test method. After the test completes, JUnit automatically deletes the temporary folder and all its contents. This is true whether the tests pass or fail or whether they throw an exception. The folder itself is located in the temporary directory designated by the operating system, and its location can be found via

```
System.getProperty("java.io.tmpdir");
```

On my Windows system, the file appears in the directory specified by the `TEMP` variable in my environment. It would be in `/tmp` on UNIX and Linux systems. The temporary directory's name is randomly created as shown in this directory on my system:

```
C:\Users\me\AppData\Local\Temp\junit3689092698555504845\
```

After creating the directory, the code then creates temporary files using the newFile() method. As you see, it then writes to the files. And in a series of assertions, I check that the files are there and that they contain the data I wrote to them. I also check that the files are named as I specified.

In this code, if I had wanted to create a directory under the temporary directory, I would have used

```
folder.newFile( "sub-folderName" );
```

One important guideline in writing unit tests is to avoid depending on resources created by other unit tests. A handy way to think of this is that you should be able to run the tests in a suite in any order and they should pass or fail the same way regardless of when they execute. Because JUnit's rule deletes the temporary directory after each test method is run, you know that you're always starting with a fresh temporary directory. Likewise, in my code, you'll also start with two fresh files, because they're specified in a rule with the annotation `@Before`. Code marked with this annotation in JUnit 4.x runs before every executed test. If you want to specify that code should be run once before the entire run of tests, use the `@BeforeClass` annotation.

Testing in JUnit 5.x

JUnit 5 came out in 2017 and is slowly displacing JUnit 4.x, even though the JUnit team continues to patch and update the JUnit 4 line. JUnit 5.x brings many convenient features to unit testing. It is comparatively easy to run JUnit 4 and 5 tests in the same project, despite the differences in syntax. In the following code, you'll see the differences from the previous code. (For brevity, I've omitted the actual tests, which are the same in both versions.)

To learn more about the JUnit transition, see Brian McGlauflin's "Migrating from JUnit 4 to JUnit 5: Important differences and benefits."

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.io.TempDir;

import java.io.*;
import java.nio.file.InvalidPathException;
import java.nio.file.Path;

import static org.junit.jupiter.api.Assertion

public class TempFilesJUnit5Test {

    Path path1, path2;
    File file1, file2;

    /* This directory and the files created i
     * tests are run, even in the event of fa
     */
    @TempDir
    Path tempDir;

    /* executed before every test: create two
    @BeforeEach
    public void setUp() {
        try {
            path1 = tempDir.resolve( "testfil
            path2 = tempDir.resolve( "testfil
        }
        catch( InvalidPathException ipe ) {
            System.err.println(
                "error creating temporary tes
                this.getClass().getSimpleName
        }

        file1 = path1.toFile();
        file2 = path2.toFile();
    }
```

The first thing to notice is that the import statements are different. Also, the creation of the temporary directory is now done by simply annotating the field containing the directory. Notice that there is no call to a constructor. The annotation takes care of that. Next, the former `@Before` annotation for the creation of the files is replaced by the more explicit `@BeforeEach`.

The file creation is also different: It relies on the NIO framework to `resolve( String filename )` to create an NIO `Path`. To maintain symmetry in the code for the tests, I convert the `Path` objects to `File` objects. A final note on this code: The exception thrown if something goes wrong when creating the files is different.

I should point out that the creation of temporary files was added to JUnit 5 in release 5.4 and is still technically listed as an “experimental” feature. With JUnit now at release 5.7, there is no reason to believe that the experimental label should be a cause for uncertainty. It’s extremely unlikely the feature will be removed, or its syntax changed substantially.

If you’re considering which release to use for a new project, I’d favor JUnit 5. A recent article, “JUnit 5.6 makes testing easy with new features” by Mert Çalişkan, provides a good look at the benefits and points to earlier articles on moving from version 4 to 5.x.

Temporary files outside of unit test frameworks

Sometimes you need to create temporary files for your application or for specialized testing and so desire to use them outside of a unit-testing framework. You have several options. The most common is to use `createTempDirectory()` and `createTempFile()`, which have been part of `java.nio.file.Files` since Java 7. A key benefit of these two methods is that you can create the temporary folder anywhere you want, rather than having it default to a predetermined locale. That is, you can create a temporary file in any directory, not necessarily the system's temporary directory. Let's examine this a little more, because there is an important gotcha to watch out for. Here is sample code for creating a temporary directory and a temporary file within it.

```
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

...

        try {
            path1 = Files.createTempDirectory
                Paths.get( "D:\\Dev\\misc" ),
            path2 = Files.createTempFile( pat
        }
        catch( IOException ioe ) {
            System.err.println( "error creati
        }
    }
```

The first call is to create the temporary directory. I pass in a location for it and the prefix to use in the name of the directory. What the call promises is that it will create a directory with a unique name that starts with the specified prefix. On my Windows system, it created

`D:\Dev\misc\tmpDirPrefix1028709432202080433`

The next call is to create a file. I pass it the `Path` to the directory that will hold the file, the prefix to use in the filename, and the extension of the file. The call promises to create a unique filename in the directory using the prefix and extension. On my system, in the temporary directory, this call created

`testfile14901457104244137806.txt`

As it stands, however, neither of these files is truly temporary. When I quit the JVM, both the file and directory are still present. The two calls promise only that they will create a unique name directory and file, using the criteria I passed into the respective functions. It's up to me to delete them when the JVM shuts down.

Clearly, deletion of the temporary files is a function I want to automate, rather than perform manually. The `deleteOnExit()` function in `java.nio.File` enables this to happen.

In the previous code, I would add the following after creating the file:

```
if( path2 != null )
    path2.toFile().deleteOnExit();
```

And as expected, the file is deleted when the JVM exits. However, the directory remains. To delete the directory, you need to empty the directory and then delete it. The `deleteOnExit()` method works only on a directory that is empty. There is no simple way around this requirement: You must walk the directory tree, delete all the files (using `File.delete()`), and after the directory is empty, delete the directory or let the JVM delete it by use of `deleteOnExit()`. I showed how to walk a directory tree in "The joy of writing command-line utilities, Part 1: Finding duplicate files."

If this process sounds like more bother than it's worth, I suggest skipping the creation of a temporary directory. Just create the temporary files you need in your system's default temporary directory and mark them for deletion on exit. In most cases, you don't care what directory they're in, so this location is as good as any other—actually better because you're implying to users that they can delete the files anytime they wish. That's not so evident for files placed elsewhere.

Using the jimfs in-memory file system

There are occasional situations where you might want to create temporary directory trees: temporary files inside temporary subdirectories under a temporary parent directory. This can become a lot to manage, and you might want to consider jimfs, which is a handy library from Google for creating a Java in-memory file system.

Jimfs is an on-heap, in-memory file system, which means that it is deleted entirely when the JVM shuts down. You don't need to flag individual files or directories for deletion—they are all destroyed upon exit from the JVM.

While in the previous examples you could count on the JVM to clean up the temporary files you created on exit, there was no absolute guarantee they'd be destroyed. For example, if the server the app was running on crashed, the JVM would not have the opportunity to delete the files. In certain cases, having these temporary files now living permanently on the server could represent an important security concern. Jimfs does not suffer from this liability.

Jimfs provides most of the functionality you'd expect of a file system, including file attributes, support for all standard file operations, symbolic links, and even the ability to watch for file system changes via Java's WatchService.

To use jimfs, add the following to your Maven pom.xml file:

```
<dependency>
  <groupId>com.google.jimfs</groupId>
  <artifactId>jimfs</artifactId>
  <version>1.2</version>
</dependency>
```

Jimfs also depends on the Google Guava library, which is an excellent omnibus library filled with a treasure trove of goodies that will make your work considerably simpler.

You add Guava to your POM file with the following:

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>30.1-jre</version>
</dependency>
```

When creating a jimfs file system, you can specify whether it should imitate a UNIX- or Windows-style file system. They differ in the naming conventions and attribute usage, so choose the one you prefer. The Windows file system implements some of Windows' quirks, such as not permitting the creation of a file in the root directory.

Here is the code for creating a file system in jimfs, then creating a file, writing to it, and reading back the contents:

```
Path path1;

// For a simple file system with Windows-styl
FileSystem fs = Jimfs.newFileSystem( Configur

try { // create foo/myfile.txt
    Path filePath = fs.getPath( "foo" );
    Files.createDirectory( filePath );
    path1 = filePath.resolve( "myfile.txt" );

    Files.write( path1, "content for file1".g

    byte[] bytesWritten = readAllBytes( path1
    if( bytesWritten.length != "content for f
        System.err.println( "error in content
    }
    else {
        System.out.println( "Contents OK" );
    }
}
catch( IOException ioe ) {
    System.err.println( "error creating or wr
}
```

In the first executable line, I create the file system—in this case a Windows model. However, I could also have created a UNIX-style file system even though I'm running on Windows. Then within the try-block, I create a directory named `foo` and within it, I create a file imaginatively named `myfile.txt`. Jimfs implements strictly the NIO file operations, so there are none of the usual `File` operations. Rather, as you see in this code, reading and writing to the files is done in arrays of bytes that NIO expects. In addition, an attempt to use the `Paths.toFile()` conversion results in an exception.

Jimfs was originally written for testing purposes inside Google, but according to its author, Colin Decker, it is also used for production purposes. For example, you can use it to write data to a file that will later be converted into a zip file and written to disk. I would expect other tasks, such as holding confidential data that needs to be encrypted before being stored to disk, would find jimfs useful as well.

Conclusion

In this article, I've shown how to use temporary files in their primary role: as stand-ins for testing against real files. I've illustrated their use in two different versions of JUnit, and I've also shown how to use temporary files outside of the testing domain, including standard Java SE options and an in-memory file system. If I have succeeded, I have removed any cloak of uncertainty around using temporary files and they should now be a standard part of your toolbox both in the architecture of applications and in testing.

Dig deeper

- Beyond the simple: An in-depth look at JUnit 5's nested tests, dynamic tests, parameterized tests, and extensions
- How to test Java microservices with Pact
- The Java tutorials: Watching a directory for changes

Andrew Binstock

Andrew Binstock (@platypusguy) was formerly the editor in chief of *Java Magazine*. Previously, he was the editor of *Dr. Dobb's Journal*. He co-founded the company behind the open-source iText PDF library, which was acquired in 2015. His book on algorithm implementation in C went through 16 printings before joining the long tail. Previously, he was the editor in chief of *UNIX Review* and, earlier, the founding editor of the *C Gazette*. He lives in Silicon Valley with his wife. When not coding or editing, he studies piano.

Share this Page