JAVA 101: LEARN JAVA
By Jeff Friesen, JavaWorld
SEP 5, 2019 10:39 AM PDT

About
A beginner's library for learning about essential Java programming concepts, syntax, APIs, and packages.

ADVANCED JAVA LANGUAGE FEATURES

Exceptions in Java, Part 2: Advanced features and types

Java exception handling with stack traces, exception chaining, try-with-resources, final re-throw, and more

JDK 1.0 introduced a framework of language features and library types for dealing with *exceptions*, which are divergences from expected program behavior. The first half of this tutorial covered Java's basic exception handling capabilities. This second half introduces more advanced capabilities provided by JDK 1.0 and its successors: JDK 1.4, JDK 7, and JDK 9. Learn how to anticipate and manage exceptions in your Java programs using advanced features such as stack traces, causes and exception chaining, try-with-resources, multi-catch, final re-throw, and stack walking.

Note that code examples in this tutorial are compatible with JDK 12.

Get the code

Download the source code for example applications in this tutorial. *Created by Jeff Friesen for JavaWorld.*

Exception handling in JDK 1.0 and 1.4: Stack traces

Each JVM *thread* (a path of execution) is associated with a *stack* that's created when the thread is created. This data structure is divided into *frames*, which are data structures associated with method calls. For this reason, each thread's stack is often referred to as a *method-call stack*.

A new frame is created each time a method is called. Each frame stores local variables, parameter variables (which hold arguments passed to the method), information for returning to the calling method, space for storing a return value, information that's useful in dispatching an exception, and so on.

A *stack trace* (also known as a *stack backtrace*) is a report of the active stack frames at a certain point in time during a thread's execution. Java's `Throwable` class (in the `java.lang` package) provides methods to print a stack trace, fill in a stack trace, and access a stack trace's elements.

Printing a stack trace

When the `throw` statement throws a throwable, it first looks for a suitable `catch` block in the executing method. If not found, it unwinds the method-call stack looking for the closest `catch` block that can handle the exception. If not found, the JVM terminates with a suitable message. Consider Listing 1.

Listing 1. `PrintStackTraceDemo.java` (version 1)

```
import java.io.IOException;

public class PrintStackTraceDemo
{
   public static void main(String[] args) throws IOException
   {
      throw new IOException();
   }
}
```

Listing 1's contrived example creates a `java.io.IOException` object and throws this object out of the `main()` method. Because `main()` doesn't handle this throwable, and because `main()` is the top-level method, the JVM terminates with a suitable message. For this application, you would see the following message:

[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!]

```
Exception in thread "main" java.io.IOException
        at PrintStackTraceDemo.main(PrintStackTraceDemo.java:7)
```

The JVM outputs this message by calling `Throwable`'s `void printStackTrace()` method, which prints a stack trace for the invoking `Throwable` object on the standard error stream. The first line shows the result of invoking the throwable's `toString()` method. The next line shows data previously recorded by `fillInStackTrace()` (discussed shortly).

Additional print stack trace methods

`Throwable`'s overloaded `void printStackTrace(PrintStream ps)` and `void printStackTrace(PrintWriter pw)` methods output the stack trace to the specified stream or writer.

The stack trace reveals the source file and line number where the throwable was created. In this case, it was created on Line 7 of the `PrintStackTrace.java` source file.

You can invoke `printStackTrace()` directly, typically from a `catch` block. For example, consider a second version of the `PrintStackTraceDemo` application.

Listing 2. `PrintStackTraceDemo.java` (version 2)

```
import java.io.IOException;

public class PrintStackTraceDemo
{
   public static void main(String[] args) throws IOException
   {
      try
      {
         a();
      }
      catch (IOException ioe)
      {
         ioe.printStackTrace();
      }
   }

   static void a() throws IOException
   {
      b();
   }

   static void b() throws IOException
   {
      throw new IOException();
   }
}
```

Listing 2 reveals a `main()` method that calls method `a()`, which calls method `b()`. Method `b()` throws an `IOException` object to the JVM, which unwinds the method-call stack until it finds `main()`'s `catch` block, which can handle the exception. The exception is handled by invoking `printStackTrace()` on the throwable. This method generates the following output:

```
java.io.IOException
        at PrintStackTraceDemo.b(PrintStackTraceDemo.java:24)
        at PrintStackTraceDemo.a(PrintStackTraceDemo.java:19)
        at PrintStackTraceDemo.main(PrintStackTraceDemo.java:9)
```

`printStackTrace()` doesn't output the thread's name. Instead, it invokes `toString()` on the throwable to return the throwable's fully-qualified class name (`java.io.IOException`), which is output on the first line. It then outputs the method-call hierarchy: the most-recently called method (`b()`) is at the top and `main()` is at the bottom.

What line does the stack trace identify?

The stack trace identifies the line where a throwable is created. It doesn't identify the line where the throwable is thrown (via `throw`), unless the throwable is thrown on the same line where it's created.

Filling in a stack trace

`Throwable` declares a `Throwable fillInStackTrace()` method that fills in the execution stack trace. In the invoking `Throwable` object, it records information about the current state of the current thread's stack frames. Consider Listing 3.

Listing 3. `FillInStackTraceDemo.java` (version 1)

```
import java.io.IOException;

public class FillInStackTraceDemo
{
   public static void main(String[] args) throws IOException
   {
      try
      {
         a();
      }
      catch (IOException ioe)
      {
         ioe.printStackTrace();
         System.out.println();
         throw (IOException) ioe.fillInStackTrace();
      }
   }

   static void a() throws IOException
   {
      b();
   }

   static void b() throws IOException
   {
      throw new IOException();
   }
}
```

The main difference between Listing 3 and Listing 2 is the `catch` block's `throw (IOException) ioe.fillInStackTrace();` statement. This statement replaces `ioe`'s stack trace, after which the throwable is re-thrown. You should observe this output:

```
java.io.IOException
        at FillInStackTraceDemo.b(FillInStackTraceDemo.java:26)
        at FillInStackTraceDemo.a(FillInStackTraceDemo.java:21)
        at FillInStackTraceDemo.main(FillInStackTraceDemo.java:9)

Exception in thread "main" java.io.IOException
        at FillInStackTraceDemo.main(FillInStackTraceDemo.java:15)
```

Instead of repeating the initial stack trace, which identifies the location where the `IOException` object was created, the second stack trace reveals the location of `ioe.fillInStackTrace()`.

Throwable constructors and `fillInStackTrace()`

Each of `Throwable`'s constructors invokes `fillInStackTrace()`. However, the following constructor (introduced in JDK 7) won't invoke this method when you pass `false` to `writableStackTrace`:

```
Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableSt
```

`fillInStackTrace()` invokes a native method that walks down the current thread's method-call stack to build the stack trace. This walk is expensive and can impact performance if it occurs too often.

If you run into a situation (perhaps involving an embedded device) where performance is critical, you can prevent the stack trace from being built by overriding `fillInStackTrace()`. Check out Listing 4.

Listing 4. `FillInStackTraceDemo.java` (version 2)

```
{
   public static void main(String[] args) throws NoStackTraceException
   {
      try
      {
         a();
      }
      catch (NoStackTraceException nste)
      {
         nste.printStackTrace();
      }
   }

   static void a() throws NoStackTraceException
   {
      b();
   }

   static void b() throws NoStackTraceException
   {
      throw new NoStackTraceException();
   }
}

class NoStackTraceException extends Exception
{
   @Override
   public synchronized Throwable fillInStackTrace()
   {
      return this;
   }
}
```

Listing 4 introduces `NoStackTraceException`. This custom checked exception class overrides `fillInStackTrace()` to return `this` -- a reference to the invoking `Throwable`. This program generates the following output:

```
NoStackTraceException
```

Comment out the overriding `fillInStackTrace()` method and you'll observe the following output:

```
NoStackTraceException
        at FillInStackTraceDemo.b(FillInStackTraceDemo.java:22)
        at FillInStackTraceDemo.a(FillInStackTraceDemo.java:17)
        at FillInStackTraceDemo.main(FillInStackTraceDemo.java:7)
```

Accessing a stack trace's elements

At times you'll need to access a stack trace's elements in order to extract details required for logging, identifying the source of a resource leak, and other purposes. The `printStackTrace()` and `fillInStackTrace()` methods don't support this task, but JDK 1.4 introduced `java.lang.StackTraceElement` and its methods for this purpose.

The `java.lang.StackTraceElement` class describes an element representing a stack frame in a stack trace. Its methods can be used to return the fully-qualified name of the class containing the execution point represented by this stack trace element along with other useful information. Here are the main methods:

- `String getClassName()` returns the fully-qualified name of the class containing the execution point represented by this stack trace element.
- `String getFileName()` returns the name of the source file containing the execution point represented by this stack trace element.
- `int getLineNumber()` returns the line number of the source line containing the execution point represented by this stack trace element.
- `String getMethodName()` returns the name of the method containing the execution point represented by this stack trace element.
- `boolean isNativeMethod()` returns `true` when the method containing the execution point represented by this stack trace element is a native method.

JDK 1.4 also introduced the `StackTraceElement[] getStackTrace()` method to the `java.lang.Thread` and `Throwable` classes. This method respectively returns an array of stack trace elements representing the invoking thread's stack dump and provides programmatic access to the stack trace information printed by `printStackTrace()`.

Listing 5 demonstrates `StackTraceElement` and `getStackTrace()`.

Listing 5. `StackTraceElementDemo.java` (version 1)

```
import java.io.IOException;

public class StackTraceElementDemo
{
   public static void main(String[] args) throws IOException
   {
      try
      {
         a();
      }
      catch (IOException ioe)
      {
         StackTraceElement[] stackTrace = ioe.getStackTrace();
         for (int i = 0; i < stackTrace.length; i++)
         {
            System.err.println("Exception thrown from " +
                               stackTrace[i].getMethodName() + " in class " +
                               stackTrace[i].getClassName() + " on line " +
                               stackTrace[i].getLineNumber() + " of file " +
                               stackTrace[i].getFileName());
            System.err.println();
         }
      }
   }

   static void a() throws IOException
   {
      b();
   }

   static void b() throws IOException
   {
      throw new IOException();
   }
}
```

When you run this application, you'll observe the following output:

```
Exception thrown from b in class StackTraceElementDemo on line 33 of file StackTraceElementDe

Exception thrown from a in class StackTraceElementDemo on line 28 of file StackTraceElementDe

Exception thrown from main in class StackTraceElementDemo on line 9 of file StackTraceElement
```

Finally, JDK 1.4 introduced the `setStackTrace()` method to `Throwable`. This method is designed for use by remote procedure call (RPC) frameworks and other advanced systems, allowing the client to override the default stack trace that's generated by `fillInStackTrace()` when a throwable is constructed.

I previously showed how to override `fillInStackTrace()` to prevent a stack trace from being built. Instead, you could install a new stack trace by using `StackTraceElement` and `setStackTrace()`. Create an array of `StackTraceElement` objects initialized via the following constructor, and pass this array to `setStackTrace()`:

```
StackTraceElement(String declaringClass, String methodName, String fileName, int lineNumber)
```

Listing 6 demonstrates `StackTraceElement` and `setStackTrace()`.

IDG COMMUNICATIONS Copyright © 2020 IDG Communications, Inc.