java.lang.StackTraceElement 1.4 *(Continued)*

- `boolean isNativeMethod()`

  returns `true` if the execution point of this element is inside a native method.

- `String toString()`

  returns a formatted string containing the class and method name and the file name and line number, if available.

7.3 Tips for Using Exceptions

There is a certain amount of controversy about the proper use of exceptions. Some programmers believe that all checked exceptions are a nuisance, others can't seem to throw enough of them. We think that exceptions (even checked exceptions) have their place, and offer you these tips for their proper use.

1. *Exception handling is not supposed to replace a simple test.*

   As an example of this, we wrote some code that tries 10,000,000 times to pop an empty stack. It first does this by finding out whether the stack is empty.

   ```
   if (!s.empty()) s.pop();
   ```

   Next, we force it to pop the stack no matter what and then catch the `EmptyStackException` that tells us we should not have done that.

   ```
   try
   {
      s.pop();
   }
   catch (EmptyStackException e)
   {
   }
   ```

   On our test machine, the version that calls `isEmpty` ran in 646 milliseconds. The version that catches the `EmptyStackException` ran in 21,739 milliseconds.

   As you can see, it took far longer to catch an exception than to perform a simple test. The moral is: Use exceptions for exceptional circumstances only.

2. *Do not micromanage exceptions.*

   Many programmers wrap every statement in a separate `try` block.

   ```
   PrintStream out;
   Stack s;
   ```

```
for (i = 0; i < 100; i++)
{
   try
   {
      n = s.pop();
   }
   catch (EmptyStackException e)
   {
      // stack was empty
   }
   try
   {
      out.writeInt(n);
   }
   catch (IOException e)
   {
      // problem writing to file
   }
}
```

This approach blows up your code dramatically. Think about the task that you want the code to accomplish. Here, we want to pop 100 numbers off a stack and save them to a file. (Never mind why—it is just a toy example.) There is nothing we can do if a problem rears its ugly head. If the stack is empty, it will not become occupied. If the file contains an error, the error will not magically go away. It therefore makes sense to wrap the *entire task* in a `try` block. If any one operation fails, you can then abandon the task.

```
try
{
   for (i = 0; i < 100; i++)
   {
      n = s.pop();
      out.writeInt(n);
   }
}
catch (IOException e)
{
   // problem writing to file
}
catch (EmptyStackException e)
{
   // stack was empty
}
```

This code looks much cleaner. It fulfills one of the promises of exception handling: to *separate* normal processing from error handling.

3. *Make good use of the exception hierarchy.*

   Don't just throw a `RuntimeException`. Find an appropriate subclass or create your own.

   Don't just catch `Throwable`. It makes your code hard to read and maintain.

   Respect the difference between checked and unchecked exceptions. Checked exceptions are inherently burdensome—don't throw them for logic errors. (For example, the reflection library gets this wrong. Callers often need to catch exceptions that they know can never happen.)

   Do not hesitate to turn an exception into another exception that is more appropriate. For example, when you parse an integer in a file, catch the `NumberFormatException` and turn it into a subclass of `IOException` or `MySubsystemException`.

4. *Do not squelch exceptions.*

   In Java, there is a tremendous temptation to shut up exceptions. If you're writing a method that calls a method that might throw an exception once a century, the compiler whines because you have not declared the exception in the `throws` list of your method. You do not want to put it in the `throws` list because then the compiler will whine about all the methods that call your method. So you just shut it up:

   ```
   public Image loadImage(String s)
   {
      try
      {
         // code that threatens to throw checked exceptions
      }
      catch (Exception e)
      {} // so there
   }
   ```

   Now your code will compile without a hitch. It will run fine, except when an exception occurs. Then, the exception will be silently ignored. If you believe that exceptions are at all important, you should make some effort to handle them right.

5. *When you detect an error, "tough love" works better than indulgence.*

   Some programmers worry about throwing exceptions when they detect errors. Maybe it would be better to return a dummy value rather than throw an exception when a method is called with invalid parameters? For example, should `Stack.pop` return `null`, or throw an exception when a stack is empty? We think it is better to throw a `EmptyStackException` at the point of failure than to have a `NullPointerException` occur at later time.

6. *Propagating exceptions is not a sign of shame.*

   Many programmers feel compelled to catch all exceptions that are thrown. If they call a method that throws an exception, such as the `FileInputStream` constructor or the `readLine` method, they instinctively catch the exception that may be generated. Often, it is actually better to *propagate* the exception instead of catching it:

```
public void readStuff(String filename) throws IOException // not a sign of shame!
{
   InputStream in = new FileInputStream(filename);
   . . .
}
```

   Higher-level methods are often better equipped to inform the user of errors or to abandon unsuccessful commands.

NOTE: Rules 5 and 6 can be summarized as "throw early, catch late."

7.4 Using Assertions

Assertions are a commonly used idiom of defensive programming. In the following sections, you will learn how to use them effectively.

7.4.1 The Assertion Concept

Suppose you are convinced that a particular property is fulfilled, and you rely on that property in your code. For example, you may be computing

```
double y = Math.sqrt(x);
```

You are certain that `x` is not negative. Perhaps it is the result of another computation that can't have a negative result, or it is a parameter of a method that requires its callers to supply only positive inputs. Still, you want to double-check rather than allow confusing "not a number" floating-point values creep into your computation. You could, of course, throw an exception:

```
if (x < 0) throw new IllegalArgumentException("x < 0");
```

But this code stays in the program, even after testing is complete. If you have lots of checks of this kind, the program may run quite a bit slower than it should.

The assertion mechanism allows you to put in checks during testing and to have them automatically removed in the production code.