Exception assertion logging

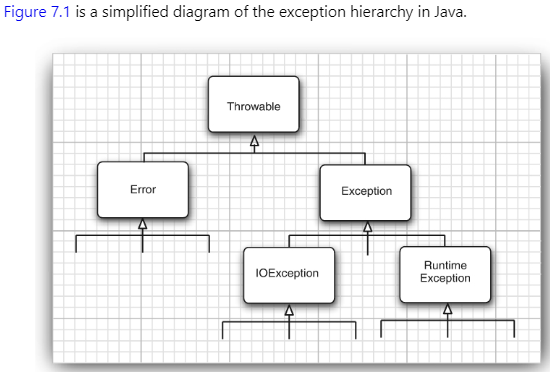
------------------------------

Dealing with errors

* User input errors. In addition to the inevitable typos, some users like to blaze their own trail instead of following directions. Suppose, for example, that a user asks to connect to a URL that is syntactically wrong. Your code should check the syntax, but suppose it does not. Then the network layer will complain.
* Device errors. Hardware does not always do what you want it to. The printer may be turned off. A web page may be temporarily unavailable. Devices will often fail in the middle of a task. For example, a printer may run out of paper during printing.
* Physical limitations. Disks can fill up; you can run out of available memory.
* Code errors. A method may not perform correctly. For example, it could deliver wrong answers or use other methods incorrectly. Computing an invalid array index, trying to find a nonexistent entry in a hash table, or trying to pop an empty stack are all examples of a code error.

NOTE

* Checked expressions 🡪 checked exceptions represent errors outside the control of the program. For example, the constructor of [FileInputStream](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/FileInputStream.html" \l "%3Cinit%3E(java.io.File)) throws *FileNotFoundException*if the input file does not exist.
* Unchecked expressions 🡪 it reflects some error inside the program logic. For example, if we divide a number by 0, Java will throw ArithmeticException



* The Error hierarchy describes internal errors and resource exhaustion situations inside the Java runtime system. You should not throw an object of this type. There is little you can do if such an internal error occurs, beyond notifying the user and trying to terminate the program gracefully. These situations are quite rare.
* When doing Java programming, focus on the Exception hierarchy.
* Exceptions that inherit from RuntimeException include such problems as

A bad cast

An out-of-bounds array access

A null pointer access

* To understand when (and what) you have to advertise in the throws clause of the methods you write, keep in mind that an exception is thrown in any of the following four situations:

1. You call a method that throws a checked exception—for example, the FileInputStream constructor.
2. You detect an error and throw a checked exception with the throw statement (the throw statement is covered in the next section).
3. You make a programming error, such as a[-1] = 0 that gives rise to an unchecked exception (in this case, an ArrayIndexOutOfBoundsException).
4. An internal error occurs in the virtual machine or runtime library.

NOTE

* In summary, a method must declare all the checked exceptions that it might throw. **Unchecked exceptions** are either beyond your control (Error) or result from conditions that you should not have allowed in the first place (RuntimeException).

CAUTION

* If you override a method from a superclass, the checked exceptions that the subclass method declares cannot be more general than those of the superclass method.

NOTE

1. Find an appropriate exception class.

2. Make an object of that class.

3. Throw it.

Catching an Exception

public void read(String filename)**throws IOException**

{

   try

   {

      var in = new FileInputStream(filename);

      int b;

      while ((b = in.read()) != -1)

      {

         process input

      }

   }

**catch (IOException exception)**

   {

      exception.printStackTrace();

   }

}

Creating exception class (Custom class)

* Class FileFormatException extends IOException

Catching Multiple Exceptions

try

{

   code that might throw exceptions

}

catch (**FileNotFoundException e**)

{

   emergency action for missing files

}

catch (**UnknownHostException e**)

{

   emergency action for unknown hosts

}

catch (**IOException e**)

{

   emergency action for all other I/O problems

}

The exception object may contain information about the nature of the exception. To find out more about the object, try

e.getMessage()

to get the detailed error message (if there is one), or

e.getClass().getName()

As of Java 7, you can catch multiple exception types in the same catch clause. For example, suppose that the action for missing files and unknown hosts is the same. Then you can combine the catch clauses:

try

{

*code that might throw exceptions*

}

catch (FileNotFoundException | UnknownHostException e)

{

*emergency action for missing files and unknown hosts*

}

catch (IOException e)

{

*emergency action for all other I/O problems*

}

This feature is only needed when catching exception types that are not subclasses of one another.

Retrhowing and Chaining Exceptions

* However, it is a better idea to set the original exception as the “cause” of the new exception:

try

{

   access the database

}

catch (SQLException original)

{

   var e = new ServletException('database error');

   e.initCause(original);

   throw e;

}

* When the exception is caught, the original exception can be retrieved:

Throwable original = caughtException.getCause();

This wrapping technique is highly recommended. It allows you to throw high-level exceptions in subsystems without losing the details of the original failure

Finally clause

* Cleanup
* When your code throws an exception, it stops processing the remaining code in your method and exits the method. This is a problem if the method has acquired some local resource, which only this method knows about, and that resource must be cleaned up.
* The code in the finally clause executes whether or not an exception was caught. In the following example, the program will close the input stream under all circumstances:

var in = new FileInputStream(. . .);

try

{

   // 1

*code that might throw exceptions*

   // 2

}

catch (IOException e){

   // 3

*show error message*

   // 4

}

finally

{

   // 5

   in.close();

}

// 6

* A finally clause can yield unexpected results when it contains return statements. Suppose you exit the middle of a try block with a return statement. Before the method returns, the finally block is executed. If the finally block also contains a return statement, then it masks the original return value. Consider this example:

public static int parseInt(String s)

{

   try

   {

      return Integer.parseInt(s);

   }

   finally

   {

      return 0; // ERROR

   }

}

Try with resource statement

Analyzing Stack Trace Elements

* A stack trace is a listing of all pending method calls at a particular point in the execution of a program.
* A more flexible approach is the StackWalker

**Tips for using exceptions**

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

As an example of this, here’s 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 catch the EmptyStackException that tells us we should not have done that.

try

{

   s.pop();

}catch (EmptyStackException e)

{

}

On my 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.

Some 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? I 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.**

Some 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. In many situations, it is actually better to propagate the exception instead of catching it:

public void readStuff(String filename) throws IOException // not a sign of shame!

{

   var in = new FileInputStream(filename, StandardCharsets.UTF\_8);

   . . .

}

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

**7. Use standard methods for reporting null-pointer and out-of-bounds exceptions.**

The Objects class has methods

requireNonNull

checkIndex

checkFromToIndex

checkFromIndexSize

for these common checks. Use them for parameter validation:

public void putData(int position, Object newValue)

{

   Objects.checkIndex(position, data.length);

   Objects.requireNonNull(newValue);

   . . .

}

If the method is called with an invalid index or a null argument, an exception is thrown, using the familiar message that the Java library uses.

**8. Don’t show stack traces to end users.**

If your program encounters an unexpected exception, it may seem a good idea to display the stack trace so the users can report it, making it easier for you to pinpoint the issue. However, stack traces can contain implementation details that you do not want to reveal to potential attackers, such as the versions of libraries that you are using. Log the stack trace so that you can retrieve it, but only display a summary message to your users.

USING ASSERTIONS

-----------------------

* The assertion mechanism allows you to put in checks during testing and to have them automatically removed in the production code.
* assert *condition*; 🡪 evaluate the condition and throw an AssertionError if it is false
* assert *condition* : *expression*; 🡪 evaluate the condition and throw an AssertionError if it is false, and the expression is passed to the constructor of the AssertionError object and turned into a message string.

Using assertions for parameter checking

* The Java language gives you three mechanisms to deal with system failures:

• Throwing an exception

• Logging

• Using assertions

* When should you choose assertions? Keep these points in mind:

• Assertion failures are intended to be fatal, unrecoverable errors.

• Assertion checks are turned on only during development and testing. (This is sometimes jokingly described as “wearing a life jacket when you are close to shore, and throwing it overboard once you are in the middle of the ocean.”)

* Therefore, you would not use assertions for signaling recoverable conditions to another part of the program or for communicating problems to the program user. Assertions should only be used to locate internal program errors during testing.

Using assertions for Documenting Assumptions

Logging

* Advantages

• It is easy to suppress all log records or just those below a certain level, and just as easy to turn them back on.

• Suppressed logs are very cheap, so there is only a minimal penalty for leaving the logging code in your application.

• Log records can be directed to different handlers—for displaying in the console, writing to a file, and so on.

• Both loggers and handlers can filter records. Filters can discard boring log entries, using any criteria supplied by the filter implementor.

• Log records can be formatted in different ways—for example, in plain text or XML.

• Applications can use multiple loggers, with hierarchical names such as com.mycompany.myapp, similar to package names.

• The logging configuration is controlled by a configuration file.

* Logging framework

Log4j

Logback

SLF4J

Commons logging

Localization

You may want to localize logging messages so that they are readable for international users.

Handlers

* By default, loggers send records to a ConsoleHandler that prints them to the System.err stream. Specifically, the logger sends the record to the parent handler, and the ultimate ancestor (with name '') has a ConsoleHandler.
* Like loggers, handlers have a logging level. For a record to be logged, its logging level must be above the threshold of both the logger and the handler.

Filters

* By default, records are filtered according to their logging levels. Each logger and handler can have an optional filter to perform additional filtering.

Formatters

* The ConsoleHandler and FileHandler classes emit the log records in text and XML formats. However, you can define your own formats as well. You need to extend the Formatter class and override the method

With so many options for logging, it is easy to lose track of the fundamentals. The following recipe summarizes the most common operations.

1. For a simple application, choose a single logger. It is a good idea to give the logger the same name as your main application package, such as com.mycompany.myprog. You can always get the logger by calling

Logger logger = Logger.getLogger('com.mycompany.myprog');

For convenience, you may want to add static fields to classes with a lot of logging activity.

private static final Logger logger = Logger.getLogger('com.mycompany.myprog');

2. The default logging configuration logs all messages of level INFO or higher to the console. Users can override the default configuration, but as you have seen, the process is a bit involved. Therefore, it is a good idea to install a more reasonable default in your application.

The following code ensures that all messages are logged to an application-specific file. Place the code into the main method of your application.

if (System.getProperty('java.util.logging.config.class') == null

      && System.getProperty('java.util.logging.config.file') == null)

{

   try

   {

      Logger.getLogger('').setLevel(Level.ALL);

      final int LOG\_ROTATION\_COUNT = 10;

      var handler = new FileHandler('%h/myapp.log', 0, LOG\_ROTATION\_COUNT);

      Logger.getLogger('').addHandler(handler);

   }

   catch (IOException e)

   {

      logger.log(Level.SEVERE, 'Can't create log file handler', e);

   }

}

3. Now you are ready to log to your heart’s content. Keep in mind that all messages with level INFO, WARNING, and SEVERE show up on the console. Therefore, reserve these levels for messages that are meaningful to the users of your program. The level FINE is a good choice for logging messages that are intended for programmers.

Whenever you are tempted to call System.out.println, emit a log message instead:

logger.fine('File open dialog canceled');

It is also a good idea to log unexpected exceptions. For example:

try

{

   . . .

}

catch (SomeException e)

{

   logger.log(Level.FINE, 'explanation', e);

}

**DEBUGGING TIPS**

I offer a number of tips that may be worth trying before you launch the debugger.

1. You can print or log the value of any variable with code like this:

System.out.println('x=' + x);

or

Logger.getGlobal().info('x=' + x);

f x is a number, it is converted to its string equivalent. If x is an object, Java calls its toString method. To get the state of the implicit parameter object, print the state of the this object.

Logger.getGlobal().info('this=' + this);

Most of the classes in the Java library are very conscientious about overriding the toString method to give you useful information about the class. This is a real boon for debugging. You should make the same effort in your classes.

2. One seemingly little-known but very useful trick is putting a separate main method in each class. Inside it, you can put a unit test stub that lets you test the class in isolation.

public class MyClass

{

*methods and fields*

   . . .

   public static void main(String[] args)

   {

*test code*

   }

}

Make a few objects, call all methods, and check that each of them does the right thing. You can leave all these main methods in place and launch the Java virtual machine separately on each of the files to run the tests. When you run an applet, none of these main methods are ever called. When you run an application, the Java virtual machine calls only the main method of the startup class.

3. If you liked the preceding tip, you should check out JUnit from [http://junit.org](http://junit.org/). JUnit is a very popular unit testing framework that makes it easy to organize suites of test cases. Run the tests whenever you make changes to a class, and add another test case whenever you find a bug.

4. A *logging proxy* is an object of a subclass that intercepts method calls, logs them, and then calls the superclass. For example, if you have trouble with the nextDouble method of the Random class, you can create a proxy object as an instance of an anonymous subclass:

var generator = new Random()

   {

      public double nextDouble()

      {

         double result = super.nextDouble();

         Logger.getGlobal().info('nextDouble: ' + result);

         return result;

      }

   };

Whenever the nextDouble method is called, a log message is generated.

To find out who called the method, generate a stack trace.

5. You can get a stack trace from any exception object with the printStackTrace method in the Throwable class. The following code catches any exception, prints the exception object and the stack trace, and rethrows the exception so it can find its intended handler.

try

{

   . . .

}

catch (Throwable t)

{

   t.printStackTrace();

   throw t;

}

You don’t even need to catch an exception to generate a stack trace. Simply insert the statement

Thread.dumpStack();

anywhere into your code to get a stack trace.

6. Normally, the stack trace is displayed on System.err. If you want to log or display the stack trace, here is how you can capture it into a string:

var out = new StringWriter();

new Throwable().printStackTrace(new PrintWriter(out));

String description = out.toString();

7. It is often handy to trap program errors in a file. However, errors are sent to System.err, not System.out. Therefore, you cannot simply trap them by running

java MyProgram > errors.txt

Instead, capture the error stream as

java MyProgram 2> errors.txt

To capture both System.err and System.out in the same file, use

java MyProgram 1> errors.txt 2>&1

This works in bash and the Windows shell.

8. Having the stack traces of uncaught exceptions show up in System.err is not ideal. These messages are confusing to end users if they happen to see them, and they are not available for diagnostic purposes when you need them. A better approach is to log them to a file. You can change the handler for uncaught exceptions with the static Thread.setDefaultUncaughtExceptionHandler method:

Thread.setDefaultUncaughtExceptionHandler(

   new Thread.UncaughtExceptionHandler()

   {

      public void uncaughtException(Thread t, Throwable e)

      {

*save information in log file*

      };

   });

9. To watch class loading, launch the Java virtual machine with the -verbose flag. You will get a printout such as the following:

[0.012s][info][class,load] opened: /opt/jdk-11.0.1/lib/modules

[0.034s][info][class,load] java.lang.Object source: jrt:/java.base

[0.035s][info][class,load] java.io.Serializable source: jrt:/java.base

[0.035s][info][class,load] java.lang.Comparable source: jrt:/java.base

[0.035s][info][class,load] java.lang.CharSequence source: jrt:/java.base

[0.035s][info][class,load] java.lang.String source: jrt:/java.base

[0.036s][info][class,load] java.lang.reflect.AnnotatedElement source: jrt:/java.base

[0.036s][info][class,load] java.lang.reflect.GenericDeclaration source: jrt:/java.base

[0.036s][info][class,load] java.lang.reflect.Type source: jrt:/java.base

[0.036s][info][class,load] java.lang.Class source: jrt:/java.base

[0.036s][info][class,load] java.lang.Cloneable source: jrt:/java.base

[0.037s][info][class,load] java.lang.ClassLoader source: jrt:/java.base

[0.037s][info][class,load] java.lang.System source: jrt:/java.base

[0.037s][info][class,load] java.lang.Throwable source: jrt:/java.base

[0.037s][info][class,load] java.lang.Error source: jrt:/java.base

[0.037s][info][class,load] java.lang.ThreadDeath source: jrt:/java.base

[0.037s][info][class,load] java.lang.Exception source: jrt:/java.base[0.037s][info][class,load] java.lang.RuntimeException source: jrt:/java.base

[0.038s][info][class,load] java.lang.SecurityManager source: jrt:/java.base . . .

This can occasionally be helpful to diagnose class path problems.

10. The -Xlint option tells the compiler to spot common code problems. For example, if you compile with the command

javac -Xlint *sourceFiles*

the compiler will report missing break statements in switch statements. (The term “lint” originally described a tool for locating potential problems in C programs, but is now generically applied to any tools that flag constructs that are questionable but not illegal.)

You will get messages such as

warning: [fallthrough] possible fall-through into case

The string in square brackets identifies the warning category. You can enable and disable each category. Since most of them are quite useful, it seems best to leave them all in place and disable only those that you don’t care about, like this:

javac -Xlint:all,-fallthrough,-serial *sourceFiles*

You get a list of all warnings from the command

javac --help -X

11. The Java VM has support for *monitoring and management* of Java applications, allowing the installation of agents in the virtual machine that track memory consumption, thread usage, class loading, and so on. This feature is particularly important for large and long-running Java programs, such as application servers. As a demonstration of these capabilities, the JDK ships with a graphical tool called jconsole that displays statistics about the performance of a virtual machine (see Figure 7.3). Start your program, then start jconsole and pick your program from the list of running Java programs.

The console gives you a wealth of information about your running program. See [www.oracle.com/technetwork/articles/java/jconsole-1564139.html](http://www.oracle.com/technetwork/articles/java/jconsole-1564139.html) for more information.

12. Java Mission Control is a professional-level profiling and diagnostics tool, available at <https://adoptopenjdk.net/jmc.html>. Like jconsole, Java Mission Control can attach to a running virtual machine. It can also analyze the output from Java Flight Recorder, a tool that collects diagnostic and profiling data from a running Java application. See <https://github.com/thegreystone/jmc-tutorial> for a comprehensive tutorial.