Exceptions, Assertions, and Logging

Topics in This Chapter

Chapter 5

In many programs, dealing with the unexpected can be more complex than implementing the "happy day" scenarios. Like most modern programming languages, Java has a robust exception-handling mechanism for transferring control from the point of failure to a competent handler. In addition, the `assert` statement provides a structured and efficient way of expressing internal assumptions. Finally, you will see how to use the logging API to keep a record of the various events, be they routine or suspicious, in the execution of your programs.

The key points of this chapter are:

1. When you throw an exception, control is transferred to the nearest handler of the exception.
2. In Java, checked exceptions are tracked by the compiler.
3. Use the `try`/`catch` construct to handle exceptions.
4. The try-with-resources statement automatically closes resources after normal execution or when an exception occurred.
5. Use the `try`/`finally` construct to deal with other actions that must occur whether or not execution proceeded normally.
6. You can catch and rethrow an exception, or chain it to another exception.
7. A stack trace describes all method calls that are pending at a point of execution.

8. An assertion checks a condition, provided that assertion checking is enabled for the class, and throws an error if the condition is not fulfilled.
9. Loggers are arranged in a hierarchy, and they can receive logging messages with levels ranging from `SEVERE` to `FINEST`.
10. Log handlers can send logging messages to alternate destinations, and formatters control the message format.
11. You can control logging properties with a log configuration file.

5.1 Exception Handling

What should a method do when it encounters a situation in which it cannot fulfill its contract? The traditional answer was that the method should return some error code. But that is cumbersome for the programmer calling the method. The caller is obliged to check for errors, and if it can't handle them, return an error code to its own caller. Not unsurprisingly, programmers didn't always check and propagate return codes, and errors went undetected, causing havoc later.

Instead of having error codes bubble up the chain of method calls, Java supports *exception handling* where a method can signal a serious problem by "throwing" an exception. One of the methods in the call chain, but not necessarily the direct caller, is responsible for handling the exception by "catching" it. The fundamental advantage of exception handling is that it decouples the processes of detecting and handling errors. In the following sections, you will see how to work with exceptions in Java.

5.1.1 Throwing Exceptions

A method may find itself in a situation where it cannot carry out the task at hand. Perhaps a required resource is missing, or it was supplied with inconsistent parameters. In such a case, it is best to throw an exception.

Suppose you implement a method that yields a random integer between two bounds:

```
public static int randInt(int low, int high) {
    return low + (int) (Math.random() * (high - low + 1));
}
```

What should happen if someone calls `randInt(10, 5)`? Trying to fix this is probably not a good idea because the caller might have been confused in more than one way. Instead, throw an appropriate exception:

```
if (low > high)
    throw new IllegalArgumentException(
        String.format("low should be <= high but low is %d and high is %d",
            low, high));
```

As you can see, the `throw` statement is used to “throw” an object of a class—here, `IllegalArgumentException`. The object is constructed with a debugging message. You will see in the next section how to pick an appropriate exception class.

When a `throw` statement executes, the normal flow of execution is interrupted immediately. The `randInt` method stops executing and does not return a value to its caller. Instead, control is transferred to a handler, as you will see in Section 5.1.4, “Catching Exceptions” (page 186).

5.1.2 The Exception Hierarchy

Figure 5-1 shows the hierarchy of exceptions in Java. All exceptions are subclasses of the class `Throwable`. Subclasses of `Error` are exceptions that are thrown when something exceptional happens that the program cannot be expected to handle, such as memory exhaustion. There is not much you can do about errors other than giving a message to the user that things have gone very wrong.

Programmer-reported exceptions are subclasses of the class `Exception`. These exceptions fall into two categories:

- *Unchecked* exceptions are subclasses of `RuntimeException`.
- All other exceptions are *checked* exceptions.

As you will see in the next section, programmers must either catch checked exceptions or declare them in the method header. The compiler checks that these exceptions are handled properly.

NOTE: The name `RuntimeException` is unfortunate. Of course, all exceptions occur at runtime. However, the exceptions that are subclasses of `RuntimeException` are not checked during compilation.

Checked exceptions are used in situations where failure should be anticipated. One common reason for failure is input and output. Files may be damaged, and network connections may fail. A number of exception classes extend `IOException`, and you should use an appropriate one to report any errors that you encounter. For example, when a file that should be there turns out not be, throw a `FileNotFoundException`.

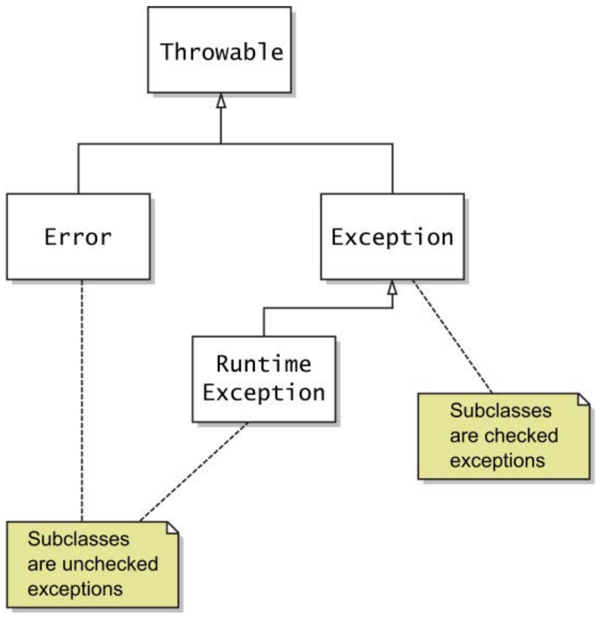
Throwable
Error
Exception
Runtime Exception
Subclasses are checked exceptions
Subclasses are unchecked exceptions

Figure 5–1 The exception hierarchy

Unchecked exceptions indicate logic errors caused by programmers, not by unavoidable external risks. For example, a `NullPointerException` is not checked. Just about any method might throw one, and programmers shouldn't spend time on catching them. Instead, they should make sure that no `null`s are dereferenced in the first place.

Sometimes, implementors need to use their judgment to make a distinction between checked and unchecked exceptions. Consider the call `Integer.parseInt(str)`. It throws an unchecked `NumberFormatException` when `str` doesn't contain a valid integer. On the other hand, `Class.forName(str)` throws a checked `ClassNotFoundException` when `str` doesn't contain a valid class name.

Why the difference? The reason is that it is possible to check whether a string is a valid integer before calling `Integer.parseInt`, but it is not possible to know whether a class can be loaded until you actually try to load it.

The Java API provides many exception classes, such as `IOException`, `IllegalArgumentException`, and so on. You should use these when appropriate. However, if none of the standard exception classes is suitable for your purpose, you can create your own by extending `Exception`, `RuntimeException`, or another existing exception class.

When you do so, it is a good idea to supply both a no-argument constructor and a constructor with a message string. For example,

```
public class FileFormatException extends IOException {
    public FileFormatException() {}
    public FileFormatException(String message) {
        super(message);
    }
    // Also add constructors for chained exceptions—see Section 5.1.7
}
```

5.1.3 Declaring Checked Exceptions

Any method that might give rise to a checked exception must declare it in the method header with a `throws` clause:

```
public void write(Object obj, String filename)
    throws IOException, ReflectiveOperationException
```

List the exceptions that the method might throw, either because of a `throw` statement or because it calls another method with a `throws` clause.

In the `throws` clause, you can combine exceptions into a common superclass. Whether or not that is a good idea depends on the exceptions. For example, if a method can throw multiple subclasses of `IOException`, it makes sense to cover them all in a clause `throws IOException`. But if the exceptions are unrelated, don't combine them into `throws Exception`—that would defeat the purpose of exception checking.

TIP: Some programmers think it is shameful to admit that a method might throw an exception. Wouldn't it be better to handle it instead? Actually, the opposite is true. You should allow each exception to find its way to a competent handler. The golden rule of exceptions is, "Throw early, catch late."

When you override a method, it cannot throw more checked exceptions than those declared by the superclass method. For example, if you extend the `write` method from the beginning of this section, the overriding method can throw fewer exceptions:

```
public void write(Object obj, String filename)
    throws FileNotFoundException
```

But if the method tried to throw an unrelated checked exception, such as an `InterruptedException`, it would not compile.

CAUTION: If the superclass method has no `throws` clause, then no overriding method can throw a checked exception.

You can use the javadoc `@throws` tag to document when a method throws a (checked or unchecked) exception. Most programmers only do this when there is something meaningful to document. For example, there is little value in telling users that an `IOException` is thrown when there is a problem with input/output. But comments such as the following can be meaningful:

```
@throws NullPointerException if filename is null
@throws FileNotFoundException if there is no file with name filename
```

NOTE: You never specify the exception type of a lambda expression. However, if a lambda expression can throw a checked exception, you can only pass it to a functional interface whose method declares that exception. For example, the call

```
list.forEach(obj -> write(obj, "output.dat"));
```

is an error. The parameter of the `forEach` method is the functional interface

```
public interface Consumer<T> {
    void accept(T t);
}
```

The `accept` method is declared not to throw any checked exception.

5.1.4 Catching Exceptions

To catch an exception, set up a `try` block. In its simplest form, it looks like this:

```
try {
    statements
} catch (ExceptionClass ex) {
    handler
}
```

If an exception of the given class occurs as the statements in the `try` block are executed, control transfers to the handler. The exception variable (`ex` in our example) refers to the exception object which the handler can inspect if desired.

There are two modifications that you can make to this basic structure. You can have multiple handlers for different exception classes:

```
try {
    statements
} catch (ExceptionClass1 ex) {
    handler1
} catch (ExceptionClass2 ex) {
    handler2
} catch (ExceptionClass3 ex) {
    handler3
}
```

The catch clauses are matched top to bottom, so the most specific exception classes must come first.

Alternatively, you can share one handler among multiple exception classes:

```
try {
    statements
} catch (ExceptionClass1 | ExceptionClass2 | ExceptionClass3 ex) {
    handler
}
```

In that case, the handler can only call those methods on the exception variable that belong to all exception classes.

5.1.5 The Try-with-Resources Statement

One problem with exception handling is resource management. Suppose you write to a file and close it when you are done:

```
ArrayList<String> lines = ...;
PrintWriter out = new PrintWriter("output.txt");
for (String line : lines) {
    out.println(line.toLowerCase());
}
out.close();
```

This code has a hidden danger. If any method throws an exception, the call to `out.close()` never happens. That is bad. Output could be lost, or if the exception is triggered many times, the system could run out of file handles.

A special form of the `try` statement can solve this issue. You can specify *resources* in the header of the `try` statement. A resource must belong to a class implementing the `AutoCloseable` interface. You can declare variables in the `try` block header:

```
ArrayList<String> lines = ...;
try (PrintWriter out = new PrintWriter("output.txt")) { // Variable declaration
    for (String line : lines)
        out.println(line.toLowerCase());
}
```

Alternatively, you can provide previously declared effectively final variables in the header:

```
PrintWriter out = new PrintWriter("output.txt");
try (out) { // Effectively final variable
    for (String line : lines)
        out.println(line.toLowerCase());
}
```

The `AutoCloseable` interface has a single method

```
public void close() throws Exception
```

NOTE: There is also a `Closeable` interface. It is a subinterface of `AutoCloseable`, also with a single `close` method. However, that method is declared to throw an `IOException`.

When the `try` block exits, either because its end is reached normally or because an exception is thrown, the `close` methods of the resource objects are invoked. For example:

```
try (PrintWriter out = new PrintWriter("output.txt")) {
    for (String line : lines) {
        out.println(line.toLowerCase());
    }
} // out.close() called here
```

You can declare multiple resources, separated by semicolons. Here is an example with two resource declarations:

```
try (Scanner in = new Scanner(Paths.get("/usr/share/dict/words"));
        PrintWriter out = new PrintWriter("output.txt")) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

The resources are closed in reverse order of their initialization—that is, `out.close()` is called before `in.close()`.

Suppose that the `PrintWriter` constructor throws an exception. Now `in` is already initialized but `out` is not. The `try` statement does the right thing: calls `in.close()` and propagates the exception.

Some `close` methods can throw exceptions. If that happens when the `try` block completed normally, the exception is thrown to the caller. However, if another exception had been thrown, causing the `close` methods of the resources to be called, and one of them throws an exception, that exception is likely to be of lesser importance than the original one.

In this situation, the original exception gets rethrown, and the exceptions from calling `close` are caught and attached as "suppressed" exceptions. This is a very useful mechanism that would be tedious to implement by hand (see Exercise 5). When you catch the primary exception, you can retrieve the secondary exceptions by calling the `getSuppressed` method:

```
try {
    ...
} catch (IOException ex) {
    Throwable[] secondaryExceptions = ex.getSuppressed();
    ...
}
```

If you want to implement such a mechanism yourself in a (hopefully rare) situation when you can't use the try-with-resources statement, call `ex.addSuppressed(secondaryException)`.

A try-with-resources statement can optionally have `catch` clauses that catch any exceptions in the statement.

5.1.6 The `finally` Clause

As you have seen, the try-with-resources statement automatically closes resources whether or not an exception occurs. Sometimes, you need to clean up something that isn't an `AutoCloseable`. In that case, use the `finally` clause:

```
try {
    Do work
} finally {
    Clean up
}
```

The `finally` clause is executed when the `try` block comes to an end, either normally or due to an exception.

This pattern occurs whenever you need to acquire and release a lock, or increment and decrement a counter, or push something on a stack and pop it off when you are done. You want to make sure that these actions happen regardless of what exceptions might be thrown.

You should avoid throwing an exception in the `finally` clause. If the body of the `try` block was terminated due to an exception, it is masked by an exception in the `finally` clause. The suppression mechanism that you saw in the preceding section only works for try-with-resources statements.

Similarly, a `finally` clause should not contain a `return` statement. If the body of the `try` block also has a `return` statement, the one in the `finally` clause replaces the return value.

It is possible to form `try` statements with `catch` clauses followed by a `finally` clause. But you have to be careful with exceptions in the `finally` clause. For example, have a look at this `try` block adapted from an online tutorial:

```
BufferedReader in = null;
try {
    in = Files.newBufferedReader(path, StandardCharsets.UTF_8);
    Read from in
} catch (IOException ex) {
    System.err.println("Caught IOException: " + ex.getMessage());
} finally {
    if (in != null) {
        in.close(); // Caution—might throw an exception
    }
}
```

The programmer clearly thought about the case when the `Files.newBufferedReader` method throws an exception. It appears as if this code would catch and print all I/O exceptions, but it actually misses one: the one that might be thrown by `in.close()`. It is often better to rewrite a complex `try`/`catch`/`finally` statement as a try-with-resources statement or by nesting a `try`/`finally` inside a `try`/`catch` statement—see Exercise 6.

5.1.7 Rethrowing and Chaining Exceptions

When an exception occurs, you may not know what to do about it, but you may want to log the failure. In that case, rethrow the exception so that a competent handler can deal with it:

```
try {
    Do work
}
catch (Exception ex) {
    logger.log(level, message, ex);
    throw ex;
}
```

NOTE: Something subtle is going on when this code is inside a method that may throw a checked exception. Suppose the enclosing method is declared as

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

At first glance, it looks as if one would need to change the `throws` clause to `throws Exception`. However, the Java compiler carefully tracks the flow and realizes that `ex` could only have been an exception thrown by one of the statements in the `try` block, not an arbitrary `Exception`.

Sometimes, you want to change the class of a thrown exception. For example, you may need to report a failure of a subsystem with an exception class that makes sense to the user of the subsystem. Suppose you encounter a database error in a servlet. The code that executes the servlet may not want to know in detail what went wrong, but it definitely wants to know that the servlet is at fault. In this case, catch the original exception and chain it to a higher-level one:

```
try {
    Access the database
}
catch (SQLException ex) {
    throw new ServletException("database error", ex);
}
```

When the `ServletException` is caught, the original exception can be retrieved as follows:

```
Throwable cause = ex.getCause();
```

The `ServletException` class has a constructor that takes as a parameter the cause of the exception. Not all exception classes do that. In that case, you have to call the `initCause` method, like this:

```
try {
    Access the database
}
catch (SQLexception ex) {
    Throwable ex2 = new CruftyOldException("database error");
    ex2.initCause(ex);
    throw ex2;
}
```

If you provide your own exception class, you should provide, in addition to the two constructors described in Section 5.1.2, "The Exception Hierarchy" (page 183), the following constructors:

```
public class FileFormatException extends IOException {
    ...
    public FileFormatException(Throwable cause) { initCause(cause); }
    public FileFormatException(String message, Throwable cause) {
        super(message);
        initCause(cause);
    }
}
```

TIP: The chaining technique is also useful if a checked exception occurs in a method that is not allowed to throw a checked exception. You can catch the checked exception and chain it to an unchecked one.

5.1.8 Uncaught Exceptions and the Stack Trace

If an exception is not caught anywhere, a *stack trace* is displayed—a listing of all pending method calls at the point where the exception was thrown. The stack trace is sent to `System.err`, the stream for error messages.

If you want to save the exception somewhere else, perhaps for inspection by your tech support staff, set the default uncaught exception handler:

```
Thread.setDefaultUncaughtExceptionHandler((thread, ex) -> {
    Record the exception
});
```

NOTE: An uncaught exception terminates the thread in which it occurred. If your application only has one thread (which is the case for the programs that you have seen so far), the program exits after invoking the uncaught exception handler.

Sometimes, you are forced to catch an exception and don't really know what to do with it. For example, the `Class.forName` method throws a checked exception that you need to handle. Instead of ignoring the exception, at least print the stack trace:

```
try {
    Class<?> cl = Class.forName(className);
    ...
} catch (ClassNotFoundException ex) {
    ex.printStackTrace();
}
```

If you want to store the stack trace of an exception, you can put it into a string as follows:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
ex.printStackTrace(new PrintWriter(out));
String description = out.toString();
```

NOTE: If you need to process the stack trace in more detail, use the `StackWalker` class. For example, the following prints all stack frames:

```
StackWalker walker = StackWalker.getInstance();
walker.forEach(frame -> System.err.println("Frame: " + frame));
```

You can also analyze the `StackWalker.StackFrame` instances in detail. See the API documentation for details.

5.1.9 The `Objects.requireNonNull` Method

The `Objects` class has a method for convenient null checks of parameters. Here is a sample usage:

```
public void process(String direction) {
    this.direction = Objects.requireNonNull(direction);
    ...
}
```

If `direction` is `null`, a `NullPointerException` is thrown—which doesn't seem like a huge improvement at first. But consider working back from a stack trace. When you see a call to `requireNonNull` as the culprit, you know right away what you did wrong.

You can also supply a message string for the exception:

```
this.direction = Objects.requireNonNull(direction, "direction must not be null");
```

A variant of this method allows you to supply an alternate value instead of throwing an exception:

```
this.direction = Objects.requireNonNullElse(direction, "North");
```

If the default is costly to compute, use yet another variant:

```
this.direction = Objects.requireNonNullElseGet(direction,
   () -> System.getProperty("com.horstmann.direction.default"));
```

The lambda expression is only evaluated if `direction` is `null`.

5.2 Assertions

Assertions are a commonly used idiom of defensive programming. 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. Still, you want to double-check rather than have "not a number" floating-point values creep into your computation. You could, of course, throw an exception:

```
if (x < 0) throw new IllegalStateException(x + " < 0");
```

But this condition stays in the program, even after testing is complete, slowing it down. The assertion mechanism allows you to put in checks during testing and to have them automatically removed in the production code.

NOTE: In Java, assertions are intended as a debugging aid for validating internal assumptions, not as a mechanism for enforcing contracts. For example, if you want to report an inappropriate parameter of a public method, don't use an assertion but throw an `IllegalArgumentException`.

5.2.1 Using Assertions

There are two forms of the assertion statement in Java:

```
assert condition;
assert condition : expression;
```

The `assert` statement evaluates the condition and throws an `AssertionError` if it is false. In the second form, the expression is turned into a string that becomes the message of the error object.

NOTE: If the expression is a `Throwable`, it is also set as the cause of the assertion error (see Section 5.1.7, "Rethrowing and Chaining Exceptions," page 190).

For example, to assert that `x` is non-negative, you can simply use the statement

```
assert x >= 0;
```

Or you can pass the actual value of `x` into the `AssertionError` object so it gets displayed later:

```
assert x >= 0 : x;
```

5.2.2 Enabling and Disabling Assertions

By default, assertions are disabled. Enable them by running the program with the `-enableassertions` or `-ea` option:

```
java -ea MainClass
```

You do not have to recompile your program because enabling or disabling assertions is handled by the class loader. When assertions are disabled, the class loader strips out the assertion code so that it won't slow execution. You can even enable assertions in specific classes or in entire packages, for example:

```
java -ea:MyClass -ea:com.mycompany.mylib... MainClass
```

This command turns on assertions for the class `MyClass` and all classes in the `com.mycompany.mylib` package *and its subpackages*. The option `-ea...` turns on assertions in all classes of the default package.

You can also disable assertions in certain classes and packages with the `-disableassertions` or `-da` option:

```
java -ea:... -da:MyClass MainClass
```

When you use the `-ea` and `-da` switches to enable or disable all assertions (and not just specific classes or packages), they do not apply to the "system classes" that are loaded without class loaders. Use the `-enablesystemassertions/-esa` switch to enable assertions in system classes.

It is also possible to programmatically control the assertion status of class loaders with the following methods:

```
void ClassLoader.setDefaultAssertionStatus(boolean enabled);
void ClassLoader.setClassAssertionStatus(String className, boolean enabled);
void ClassLoader.setPackageAssertionStatus(String packageName, boolean enabled);
```

As with the `-enableassertions` command-line option, the `setPackageAssertionStatus` method sets the assertion status for the given package and its subpackages.

5.3 Logging

Every Java programmer is familiar with the process of inserting `System.out.println` calls into troublesome code to gain insight into program behavior. Of course, once you have figured out the cause of trouble, you remove the print statements—only to put them back in when the next problem surfaces. The logging API is designed to overcome this problem.

5.3.1 Using Loggers

Let's get started with the simplest possible case. The logging system manages a default logger that you get by calling `Logger.getGlobal()`. Use the `info` method to log an information message:

```
Logger.getGlobal().info("Opening file " + filename);
```

The record is printed like this:

```
Aug 04, 2014 09:53:34 AM com.mycompany.MyClass read INFO: Opening file data.txt
```

Note that the time and the names of the calling class and method are automatically included.

However, if you call

```
Logger.getGlobal().setLevel(Level.OFF);
```

then calls to `info` have no effect.

NOTE: In the above example, the message `"Opening file " + filename` is created even if logging is disabled. If you are concerned about the cost of creating the message, you can use a lambda expression instead:

```
Logger.getGlobal().info(() -> "Opening file " + filename);
```

5.3.2 Loggers

In a professional application, you wouldn't want to log all records to a single global logger. Instead, you can define your own loggers.

When you request a logger with a given name for the first time, it is created.

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
```

Subsequent calls to the same name yield the same logger object.

Similar to package names, logger names are hierarchical. In fact, they are more hierarchical than packages. There is no semantic relationship between a package and its parent, but logger parents and children share certain properties. For example, if you turn off messages to the logger `"com.mycompany"`, then the child loggers are also deactivated.

NOTE: In this section, we introduce the `java.util.logging` framework that is a part of the JDK. This framework is not universally loved, and there are alternatives with better performance and more flexibility. Many projects use a logging façade such as SLF4J (`https://www.slf4j.org`) that lets users plug in the logging framework of their choice. Nevertheless, `java.util.logging` is fine for many use cases, and learning how it works will help you understand the alternatives.

NOTE: Even the JVM doesn't love `java.util.logging`, but for an entirely different reason. In order to have a minimal footprint, the most basic JVM modules don't want to depend on the `java.logging` module that contains the `java.util.logging` package. There is a lightweight `System.Logger` interface that some JVM modules use for logging. On a full JVM, these logs are redirected to `java.util.logging`, but they can also be redirected elsewhere. This is not a facility that is intended for application programmers, so you should use `java.util.logging` or a logging façade.

5.3.3 Logging Levels

There are seven logging levels: `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`. By default, the top three levels are actually logged. You can set a different threshold, for example:

```
logger.setLevel(Level.FINE);
```

Now `FINE` and all levels above it are logged.

You can also use `Level.ALL` to turn on logging for all levels or `Level.OFF` to turn all logging off.

There are logging methods corresponding to each level, such as

```
logger.warning(message);
logger.fine(message);
```

and so on. Alternatively, if the level is variable, you can use the `log` method and supply the level:

```
Level level = ...;
logger.log(level, message);
```

TIP: The default logging configuration logs all records with the level of `INFO` or higher. Therefore, you should use the levels `CONFIG`, `FINE`, `FINER`, and `FINEST` for debugging messages that are useful for diagnostics but meaningless to the user.

CAUTION: If you set the logging level to a value finer than `INFO`, you also need to change the log handler configuration. The default log handler suppresses messages below `INFO`. See Section 5.3.6, "Log Handlers" (page 200) for details.

5.3.4 Other Logging Methods

There are convenience methods for tracing execution flow:

```
void entering(String className, String methodName)
void entering(String className, String methodName, Object param)
void entering(String className, String methodName, Object[] params)
void exiting(String className, String methodName)
void exiting(String className, String methodName, Object result)
```

For example:

```
public int read(String file, String pattern) {
    logger.entering("com.mycompany.mylib.Reader", "read",
        new Object[] { file, pattern });
    ...
    logger.exiting("com.mycompany.mylib.Reader", "read", count);
    return count;
}
```

These calls generate log records of level `FINER` that start with the strings `ENTRY` and `RETURN`.

NOTE: Oddly enough, these methods have never been turned into methods with variable arguments.

A common use for logging is to log unexpected exceptions. Two convenience methods include a description of the exception in the log record.

```
void log(Level l, String message, Throwable t)
void throwing(String className, String methodName, Throwable t)
```

Typical uses are

```
try {
    ...
}
catch (IOException ex) {
    logger.log(Level.SEVERE, "Cannot read configuration", ex);
}
```

and

```
if (...) {
    IOException ex = new IOException("Cannot read configuration");
    logger.throwing("com.mycompany.mylib.Reader", "read", ex);
    throw ex;
}
```

The `throwing` call logs a record with level `FINER` and a message that starts with `THROW`.

NOTE: The default log record shows the name of the class and method that contain the logging call, as inferred from the call stack. However, if the virtual machine optimizes execution, accurate call information may not be available. You can use the `logp` method to give the precise location of the calling class and method. The method signature is

```
void logp(Level l, String className, String methodName, String message)
```

NOTE: If you want the logging messages to be understood by users in multiple languages, you can localize them with the methods

```
void logrb(Level level, ResourceBundle bundle,
    String msg, Object... params)
void logrb(Level level, ResourceBundle bundle,
    String msg, Throwable thrown)
```

Resource bundles are described in Chapter 13.

5.3.5 Logging Configuration

You can change various properties of the logging system by editing a configuration file. The default configuration file is located at `jre/lib/logging.properties`. To use another file, set the `java.util.logging.config.file` property to the file location by starting your application with

```
java -Djava.util.logging.config.file=configFile MainClass
```

CAUTION: Calling `System.setProperty("java.util.logging.config.file", configFile)` in `main` has no effect because the log manager is initialized during VM startup, before `main` executes.

To change the default logging level, edit the configuration file and modify the line

```
.level=INFO
```

You can specify the logging levels for your own loggers by adding lines such as

```
com.mycompany.myapp.level=FINE
```

That is, append the `.level` suffix to the logger name.

As you will see in the next section, loggers don't actually send the messages to the console—that is the job of the handlers. Handlers also have levels. To see `FINE` messages on the console, you also need to set

```
java.util.logging.ConsoleHandler.level=FINE
```

CAUTION: The settings in the log manager configuration are not system properties. Starting a program with `-Dcom.mycompany.myapp.level=FINE` does not have any effect on the logger.

It is also possible to change logging levels in a running program by using the `jconsole` program. For details, see `www.oracle.com/technetwork/articles/java/jconsole-1564139.html#LoggingControl` for details.

5.3.6 Log 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. The log manager configuration file sets the logging level of the default console handler as

```
java.util.logging.ConsoleHandler.level=INFO
```

To log records with level `FINE`, change both the default logger level and the handler level in the configuration. Alternatively, you can bypass the configuration file altogether and install your own handler.

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

By default, a logger sends records both to its own handlers and the handlers of the parent. Our logger is a descendant of the ultimate ancestor `""` that sends all records with level `INFO` and above to the console. We don't want to see those records twice, however, so we set the `useParentHandlers` property to `false`.

To send log records elsewhere, add another handler. The logging API provides two handlers for this purpose: a `FileHandler` and a `SocketHandler`. The `SocketHandler` sends records to a specified host and port. Of greater interest is the `FileHandler` that collects records in a file.

You can simply send records to a default file handler, like this:

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

The records are sent to a file `java`*n*`.log` in the user's home directory, where *n* is a number to make the file unique. By default, the records are formatted in XML. A typical log record has the form

```
<record>
    <date>2014-08-04T09:53:34</date>
    <millis>1407146014072</millis>
    <sequence>1</sequence>
    <logger>com.mycompany.myapp</logger>
    <level>INFO</level>
    <class>com.horstmann.corejava.Employee</class>
    <method>read</method>
    <thread>10</thread>
    <message>Opening file staff.txt</message>
</record>
```

You can modify the default behavior of the file handler by setting various parameters in the log manager configuration (see Table 5-1) or by using one of the following constructors:

```
FileHandler(String pattern)
FileHandler(String pattern, boolean append)
FileHandler(String pattern, int limit, int count)
FileHandler(String pattern, int limit, int count, boolean append)
```

See Table 5-1 for the meaning of the construction parameters.

You probably don't want to use the default log file name. Use a pattern such as `%h/myapp.log` (see Table 5-2 for an explanation of the pattern variables.)

If multiple applications (or multiple copies of the same application) use the same log file, you should turn the `append` flag on. Alternatively, use `%u` in the file name pattern so that each application creates a unique copy of the log.

It is also a good idea to turn file rotation on. Log files are kept in a rotation sequence, such as `myapp.log.0`, `myapp.log.1`, `myapp.log.2`, and so on. Whenever a file exceeds the size limit, the oldest log is deleted, the other files are renamed, and a new file with generation number `0` is created.

Table 5–1 File Handler Configuration Parameters

Configuration Property	Description	Default
`java.util.logging.FileHandler.level`	The handler level	`Level.ALL`
`java.util.logging.FileHandler.append`	When `true`, log records are appended to an existing file; otherwise, a new file is opened for each program run.	`false`

(Continues)

Table 5-1 File Handler Configuration Parameters *(Continued)*

Configuration Property	Description	Default
`java.util.logging.FileHandler.limit`	The approximate maximum number of bytes to write in a file before opening another (`0` = no limit).	`0` in the `FileHandler` class, `50000` in the default log manager configuration
`java.util.logging.FileHandler.pattern`	The file name pattern (see Table 5-2)	`%h/java%u.log`
`java.util.logging.FileHandler.count`	The number of logs in a rotation sequence	`1` (no rotation)
`java.util.logging.FileHandler.filter`	The filter for filtering log records (see Section 5.3.7)	No filtering
`java.util.logging.FileHandler.encoding`	The character encoding	The platform character encoding
`java.util.logging.FileHandler.formatter`	The formatter for each log record	`java.util.logging.XMLFormatter`

Table 5-2 Log File Pattern Variables

Variable	Description
`%h`	The user's home directory (the `user.home` property)
`%t`	The system's temporary directory
`%u`	A unique number
`%g`	The generation number for rotated logs (a `.%g` suffix is used if rotation is specified and the pattern doesn't contain `%g`)
`%%`	The percent character

5.3.7 Filters and Formatters

Besides filtering by logging levels, each logger and handler can have an additional filter that implements the `Filter` interface, a functional interface with a method

```
boolean isLoggable(LogRecord record)
```

To install a filter into a logger or handler, call the `setFilter` method. Note that you can have at most one filter at a time.

The `ConsoleHandler` and `FileHandler` classes emit the log records in text and XML formats. However, you can define your own formats as well. Extend the `Formatter` class and override the method

```
String format(LogRecord record)
```

Format the record in any way you like and return the resulting string. In your format method, you may want to call the method

```
String formatMessage(LogRecord record)
```

That method formats the message part of the record, substituting parameters and applying localization.

Many file formats (such as XML) require head and tail parts that surround the formatted records. To achieve this, override the methods

```
String getHead(Handler h)
String getTail(Handler h)
```

Finally, call the `setFormatter` method to install the formatter into the handler.

Exercises

1. Write a method `public ArrayList<Double> readValues(String filename) throws` ... that reads a file containing floating-point numbers. Throw appropriate exceptions if the file could not be opened or if some of the inputs are not floating-point numbers.
2. Write a method `public double sumOfValues(String filename) throws` ... that calls the preceding method and returns the sum of the values in the file. Propagate any exceptions to the caller.
3. Write a program that calls the preceding method and prints the result. Catch the exceptions and provide feedback to the user about any error conditions.
4. Repeat the preceding exercise, but don't use exceptions. Instead, have `readValues` and `sumOfValues` return error codes of some kind.
5. Implement a method that contains the code with a `Scanner` and a `PrintWriter` in Section 5.1.5, "The Try-with-Resources Statement" (page 187). But don't use the try-with-resources statement. Instead, just use `catch` clauses. Be sure to close both objects, provided they have been properly constructed. You need to consider the following conditions:

- The `Scanner` constructor throws an exception.
- The `PrintWriter` constructor throws an exception.
- `hasNext`, `next`, or `println` throw an exception.
- `out.close()` throws an exception.
- `in.close()` throws an exception.

6. Section 5.1.6, "The `finally` Clause" (page 189) has an example of a broken `try` statement with `catch` and `finally` clauses. Fix the code with (a) catching the exception in the `finally` clause, (b) a `try`/`catch` statement containing a `try`/`finally` statement, and (c) a try-with-resources statement with a `catch` clause.
7. Explain why

```
try (Scanner in = new Scanner(Paths.get("/usr/share/dict/words"));
        PrintWriter out = new PrintWriter(outputFile)) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

is better than

```
Scanner in = new Scanner(Paths.get("/usr/share/dict/words"));
PrintWriter out = new PrintWriter(outputFile);
try (in; out) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

8. For this exercise, you'll need to read through the source code of the `java.util.Scanner` class. If input fails when using a `Scanner`, the `Scanner` class catches the input exception and closes the resource from which it consumes input. What happens if closing the resource throws an exception? How does this implementation interact with the handling of suppressed exceptions in the try-with-resources statement?
9. Design a helper method so that one can use a `ReentrantLock` in a try-with-resources statement. Call `lock` and return an `AutoCloseable` whose `close` method calls `unlock` and throws no exceptions.
10. The methods of the `Scanner` and `PrintWriter` classes do not throw checked exceptions to make them easier to use for beginning programmers. How do you find out whether errors occurred during reading or writing? Note that the constructors *can* throw checked exceptions. Why does that defeat the goal of making the classes easier to use for beginners?

11. Write a recursive `factorial` method in which you print all stack frames before you return the value. Construct (but don't throw) an exception object of any kind and get its stack trace, as described in Section 5.1.8, "Uncaught Exceptions and the Stack Trace" (page 192).
12. Compare the use of `Objects.requireNonNull(obj)` and `assert obj != null`. Give a compelling use for each.
13. Write a method `int min(int[] values)` that, just before returning the smallest value, asserts that it is indeed ≤ all values in the array. Use a private helper method or, if you already peeked into Chapter 8, `Stream.allMatch`. Call the method repeatedly on a large array and measure the runtime with assertions enabled, disabled, and removed.
14. Implement and test a log record filter that filters out log records containing bad words such as sex, drugs, and C++.
15. Implement and test a log record formatter that produces an HTML file.