CHAPTER 11

Exceptions

If anything can go wrong, it will.
—Finagle's Law
(often incorrectly attributed to Murphy, whose law is rather
different—which only goes to show that Finagle was right)

WHEN a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an *exception*. An example of such a violation is an attempt to index outside the bounds of an array. Some programming languages and their implementations react to such errors by peremptorily terminating the program; other programming languages allow an implementation to react in an arbitrary or unpredictable way. Neither of these approaches is compatible with the design goals of the Java platform: to provide portability and robustness. Instead, the Java programming language specifies that an exception will be thrown when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be *thrown* from the point where it occurred and is said to be *caught* at the point to which control is transferred.

Programs can also throw exceptions explicitly, using `throw` statements (§14.17).

Explicit use of `throw` statements provides an alternative to the old-fashioned style of handling error conditions by returning funny values, such as the integer value `-1` where a negative value would not normally be expected. Experience shows that too often such funny values are ignored or not checked for by callers, leading to programs that are not robust, exhibit undesirable behavior, or both.

Every exception is represented by an instance of the class `Throwable` or one of its subclasses; such an object can be used to carry information from the point at which an exception occurs to the handler that catches it. Handlers are established by `catch` clauses of `try` statements (§14.19). During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception. If no such handler is found, then the method `uncaughtException` is invoked for the `ThreadGroup` that is the parent of the current thread—thus every effort is made to avoid letting an exception go unhandled.

The exception mechanism of the Java platform is integrated with its synchronization model (§17), so that locks are released as `synchronized` statements (§14.18) and invocations of `synchronized` methods (§8.4.3.6, §15.12) complete abruptly.

This chapter describes the different causes of exceptions (§11.1). It details how exceptions are checked at compile time (§11.2) and processed at run time (§11.3). A detailed example (§11.4) is then followed by an explanation of the exception hierarchy (§11.5).

11.1 The Causes of Exceptions

If we do not succeed, then we run the risk of failure.
—J. Danforth Quayle (1990)

An exception is thrown for one of three *reasons*:

- An abnormal execution condition was synchronously detected by the Java virtual machine. Such conditions arise because:
  - evaluation of an expression violates the normal semantics of the language, such as an integer divide by zero, as summarized in §15.6
  - an error occurs in loading or linking part of the program (§12.2, §12.3)
  - some limitation on a resource is exceeded, such as using too much memory

  These exceptions are not thrown at an arbitrary point in the program, but rather at a point where they are specified as a possible result of an expression evaluation or statement execution.
- A `throw` statement (§14.17) was executed.
- An asynchronous exception occurred either because:
  - the method `stop` of class `Thread` was invoked
  - an internal error has occurred in the virtual machine (§11.5.2)

Exceptions are represented by instances of the class `Throwable` and instances of its subclasses. These classes are, collectively, the *exception classes*.

11.2 Compile-Time Checking of Exceptions

A compiler for the Java programming language checks, at compile time, that a program contains handlers for *checked exceptions*, by analyzing which checked exceptions can result from execution of a method or constructor. For each checked exception which is a possible result, the `throws` clause for the method (§8.4.4) or constructor (§8.8.4) must mention the class of that exception or one of the superclasses of the class of that exception. This compile-time checking for the presence of exception handlers is designed to reduce the number of exceptions which are not properly handled.

The *unchecked exceptions classes* are the class `RuntimeException` and its subclasses, and the class `Error` and its subclasses. All other exception classes are *checked exception classes*. The Java API defines a number of exception classes, both checked and unchecked. Additional exception classes, both checked and unchecked, may be declared by programmers. See §11.5 for a description of the exception class hierarchy and *some of* the exception classes defined by the Java API and Java virtual machine.

The checked exception classes named in the `throws` clause are part of the contract between the implementor and user of the method or constructor. The `throws` clause of an overriding method may not specify that this method will result in throwing any checked exception which the overridden method is not permitted, by its `throws` clause, to throw. When interfaces are involved, more than one method declaration may be overridden by a single overriding declaration. In this case, the overriding declaration must have a `throws` clause that is compatible with *all* the overridden declarations (§9.4).

We say that a statement or expression can throw a checked exception type *E* if, according to the rules given below, the execution of the statement or expression can result in an exception of type *E* being thrown.

11.2.1 Exception Analysis of Expressions

A method invocation expression can throw an exception type *E* iff either:

- The method to be invoked is of the form *Primary.Identifier* and the *Primary* expression can throw *E*; or
- Some expression of the argument list can throw *E*; or
- *E* is listed in the throws clause of the type of method that is invoked.

A class instance creation expression can throw an exception type *E* iff either:

- The expression is a qualified class instance creation expression and the qualifying expression can throw *E*; or
- Some expression of the argument list can throw *E*; or
- *E* is listed in the throws clause of the type of the constructor that is invoked; or
- The class instance creation expression includes a *ClassBody*, and some instnance initializer block or instance variable initializer expression in the *ClassBody* can throw *E*.

For every other kind of expression, the expression can throw type *E* iff one of its immediate subexpressions can throw *E*.

11.2.2 Exception Analysis of Statements

A throw statement can throw an exception type *E* iff the static type of the throw expression is *E* or a subtype of *E*, or the thrown expression can throw *E*.

A explicit constructor invocation statement can throw an exception type *E* iff either:

- Some subexpression of the constructor invocation list can throw *E*; or
- *E* is declared in the throws clause of the constructor that is invoked.

A `try` statement can throw an exception type *E* iff either:

- The `try` block can throw *E* and *E* is not assignable to any `catch` parameter of the `try` statement and either no `finally` block is present or the `finally` block can complete normally; or
- Some `catch` block of the `try` statement can throw E and either no `finally` block is present or the `finally` block can complete normally; or
- A `finally` block is present and can throw *E*.

Any other statement *S* can throw an exception type *E* iff expression or statement immediately contained in *S* can throw *E*.

11.2.3 Exception Checking

It is a compile-time error if a method or constructor body can throw some exception type *E* when both of the following hold:

- *E* is a checked exception type
- *E* is not a subtype of some type declared in the throws clause of the method or constructor.

It is a compile-time error if a static initializer (§8.7) or class variable initializer within a named class or interface (§8.3.2), can throw a checked exception type.

It is compile-time error if an instance variable initializer of a named class can throw a checked exception unless that exception or one of its supertypes is explicitly declared in the `throws` clause of each constructor of its class and the class has at least one explicitly declared constructor. An instance variable initializer in an anonymous class (§15.9.5) can throw any exceptions.

It is compile-time error if an instance initializer of a named class can throw a checked exception unless that exception or one of its supertypes is explicitly declared in the `throws` clause of each constructor of its class and the class has at least one explicitly declared constructor. An instance initializer in an anonymous class (§15.9.5) can throw any exceptions.

It is a compile-time error if a `catch` clause catches checked exception type *E1* but there exists no checked exception type *E2* such that all of the following hold:

- *E2 <: E1*
- The `try` block corresponding to the `catch` clause can throw *E2*
- No preceding `catch` block of the immediately enclosing `try` statement catches *E2* or a supertype of *E2*.

11.2.4 Why Errors are Not Checked

Those unchecked exception classes which are the *error classes* (`Error` and its subclasses) are exempted from compile-time checking because they can occur at many points in the program and recovery from them is difficult or impossible. A program declaring such exceptions would be cluttered, pointlessly.

11.2.5 Why Runtime Exceptions are Not Checked

The *runtime exception classes* (`RuntimeException` and its subclasses) are exempted from compile-time checking because, in the judgment of the designers of the Java programming language, having to declare such exceptions would not aid significantly in establishing the correctness of programs. Many of the operations and constructs of the Java programming language can result in runtime exceptions. The information available to a compiler, and the level of analysis the compiler performs, are usually not sufficient to establish that such run-time exceptions cannot occur, even though this may be obvious to the programmer. Requiring such exception classes to be declared would simply be an irritation to programmers.

For example, certain code might implement a circular data structure that, by construction, can never involve `null` references; the programmer can then be certain that a `NullPointerException` cannot occur, but it would be difficult for a compiler to prove it. The theorem-proving technology that is needed to establish such global properties of data structures is beyond the scope of this specification.

11.3 Handling of an Exception

When an exception is thrown, control is transferred from the code that caused the exception to the nearest dynamically-enclosing `catch` clause of a `try` statement (§14.19) that handles the exception.

A statement or expression is *dynamically enclosed* by a `catch` clause if it appears within the `try` block of the `try` statement of which the `catch` clause is a part, or if the caller of the statement or expression is dynamically enclosed by the `catch` clause.

The *caller* of a statement or expression depends on where it occurs:

- If within a method, then the caller is the method invocation expression (§15.12) that was executed to cause the method to be invoked.
- If within a constructor or an instance initializer or the initializer for an instance variable, then the caller is the class instance creation expression (§15.9) or the method invocation of `newInstance` that was executed to cause an object to be created.
- If within a static initializer or an initializer for a `static` variable, then the caller is the expression that used the class or interface so as to cause it to be initialized.

Whether a particular `catch` clause *handles* an exception is determined by comparing the class of the object that was thrown to the declared type of the parameter of the `catch` clause. The `catch` clause handles the exception if the type of its parameter is the class of the exception or a superclass of the class of the exception. Equivalently, a `catch` clause will catch any exception object that is an `instanceof` (§15.20.2) the declared parameter type.

The control transfer that occurs when an exception is thrown causes abrupt completion of expressions (§15.6) and statements (§14.1) until a `catch` clause is encountered that can handle the exception; execution then continues by executing the block of that `catch` clause. The code that caused the exception is never resumed.

If no `catch` clause handling an exception can be found, then the current thread (the thread that encountered the exception) is terminated, but only after all `finally` clauses have been executed and the method `uncaughtException` has been invoked for the `ThreadGroup` that is the parent of the current thread.

In situations where it is desirable to ensure that one block of code is always executed after another, even if that other block of code completes abruptly, a `try` statement with a `finally` clause (§14.19.2) may be used.

If a `try` or `catch` block in a `try`–`finally` or `try`–`catch`–`finally` statement completes abruptly, then the `finally` clause is executed during propagation of the exception, even if no matching `catch` clause is ultimately found. If a `finally` clause is executed because of abrupt completion of a `try` block and the `finally` clause itself completes abruptly, then the reason for the abrupt completion of the `try` block is discarded and the new reason for abrupt completion is propagated from there.

The exact rules for abrupt completion and for the catching of exceptions are specified in detail with the specification of each statement in §14 and for expressions in §15 (especially §15.6).

11.3.1 Exceptions are Precise

Exceptions are *precise*: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated. If optimized code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the program.

11.3.2 Handling Asynchronous Exceptions

Most exceptions occur synchronously as a result of an action by the thread in which they occur, and at a point in the program that is specified to possibly result in such an exception. An asynchronous exception is, by contrast, an exception that can potentially occur at any point in the execution of a program.

Proper understanding of the semantics of asynchronous exceptions is necessary if high-quality machine code is to be generated.

Asynchronous exceptions are rare. They occur only as a result of:

- An invocation of the `stop` methods of class `Thread` or `ThreadGroup`
- An internal error (§11.5.2) in the Java virtual machine

The `stop` methods may be invoked by one thread to affect another thread or all the threads in a specified thread group. They are asynchronous because they may occur at any point in the execution of the other thread or threads. An `InternalError` is considered asynchronous.

The Java platform permits a small but bounded amount of execution to occur before an asynchronous exception is thrown. This delay is permitted to allow optimized code to detect and throw these exceptions at points where it is practical to handle them while obeying the semantics of the Java programming language.

A simple implementation might poll for asynchronous exceptions at the point of each control transfer instruction. Since a program has a finite size, this provides a bound on the total delay in detecting an asynchronous exception. Since no asynchronous exception will occur between control transfers, the code generator has some flexibility to reorder computation between control transfers for greater performance.

The paper *Polling Efficiently on Stock Hardware* by Marc Feeley, *Proc. 1993 Conference on Functional Programming and Computer Architecture,* Copenhagen, Denmark, pp. 179–187, is recommended as further reading.

Like all exceptions, asynchronous exceptions are precise (§11.3.1).

11.4 An Example of Exceptions

Consider the following example:

```
class TestException extends Exception {

    TestException() { super(); }

    TestException(String s) { super(s); }

}
```

```
class Test {
   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++) {
         try {
           thrower(args[i]);
           System.out.println("Test \"" + args[i] +
             "\" didn't throw an exception");
         } catch (Exception e) {
           System.out.println("Test \"" + args[i] +
             "\" threw a " + e.getClass() +
             "\n    with message: " + e.getMessage());
         }
      }
   }

   static int thrower(String s) throws TestException {
      try {
         if (s.equals("divide")) {
           int i = 0;
           return i/i;
         }
         if (s.equals("null")) {
           s = null;
           return s.length();
         }
         if (s.equals("test"))
           throw new TestException("Test message");
         return 0;
      } finally {
         System.out.println("[thrower(\"" + s +
            "\") done]");
      }
   }

}
```

If we execute the test program, passing it the arguments:

```
divide null not test
```

it produces the output:

```
[thrower("divide") done]
Test "divide" threw a class java.lang.ArithmeticException
    with message: / by zero
[thrower("null") done]
Test "null" threw a class java.lang.NullPointerException
    with message: null
[thrower("not") done]
Test "not" didn't throw an exception
[thrower("test") done]
```

```
Test "test" threw a class TestException
    with message: Test message
```

This example declares an exception class `TestException`. The `main` method of class `Test` invokes the `thrower` method four times, causing exceptions to be thrown three of the four times. The `try` statement in method `main` catches each exception that the `thrower` throws. Whether the invocation of `thrower` completes normally or abruptly, a message is printed describing what happened.

The declaration of the method `thrower` must have a `throws` clause because it can throw instances of `TestException`, which is a checked exception class (§11.2). A compile-time error would occur if the `throws` clause were omitted.

Notice that the `finally` clause is executed on every invocation of `thrower`, whether or not an exception occurs, as shown by the "`[thrower(...) done]`" output that occurs for each invocation.

11.5 The Exception Hierarchy

The possible exceptions in a program are organized in a hierarchy of classes, rooted at class `Throwable` (§11.5), a direct subclass of `Object`. The classes `Exception` and `Error` are direct subclasses of `Throwable`. The class `RuntimeException` is a direct subclass of `Exception`.

Programs can use the pre-existing exception classes in `throw` statements, or define additional exception classes, as subclasses of `Throwable` or of any of its subclasses, as appropriate. To take advantage of the Java platform's compile-time checking for exception handlers, it is typical to define most new exception classes as checked exception classes, specifically as subclasses of `Exception` that are not subclasses of `RuntimeException`.

The class `Exception` is the superclass of all the exceptions that ordinary programs may wish to recover from. The class `RuntimeException` is a subclass of class `Exception`. The subclasses of `RuntimeException` are unchecked exception classes. The subclasses of `Exception` other than `RuntimeException` are all checked exception classes.

The class `Error` and its subclasses are exceptions from which ordinary programs are not ordinarily expected to recover. See the Java API specification for a detailed description of the exception hierarchy.

The class `Error` is a separate subclass of `Throwable`, distinct from `Exception` in the class hierarchy, to allow programs to use the idiom:

```
} catch (Exception e) {
```

to catch all exceptions from which recovery may be possible without catching errors from which recovery is typically not possible.

11.5.1 Loading and Linkage Errors

The Java virtual machine throws an object that is an instance of a subclass of `LinkageError` when a loading, linkage, preparation, verification or initialization error occurs:

- The loading process is described in §12.2.
- The linking process is described in §12.3.
- The class verification process is described in §12.3.1.
- The class preparation process is described in §12.3.2.
- The class initialization process is described in §12.4.

11.5.2 Virtual Machine Errors

The Java virtual machine throws an object that is an instance of a subclass of the class `VirtualMachineError` when an internal error or resource limitation prevents it from implementing the semantics of the Java programming language. See *The Java™ Virtual Machine Specification Second Edition* for the definitive discussion of these errors.

I never forget a face—but in your case I'll be glad to make an exception.
—Groucho Marx