11.5 Java Exception Hierarchy

All Java exception classes inherit directly or indirectly from class `Exception`, forming an *inheritance hierarchy*. You can extend this hierarchy with your own exception classes.

Figure 11.4 shows a small portion of the inheritance hierarchy for class `Throwable` (a subclass of `Object`), which is the superclass of class `Exception`. Only `Throwable` objects can be used with the exception-handling mechanism. Class `Throwable` has two direct subclasses: `Exception` and `Error`. Class `Exception` and its subclasses—for example, `RuntimeException` (package `java.lang`) and `IOException` (package `java.io`)—represent exceptional situations that can occur in a Java program and that can be caught by the application. Class `Error` and its subclasses represent *abnormal situations* that happen in the JVM. Most `Error`*s happen infrequently and should not be caught by applications—it's usually not possible for applications to recover from* `Error`*s.*

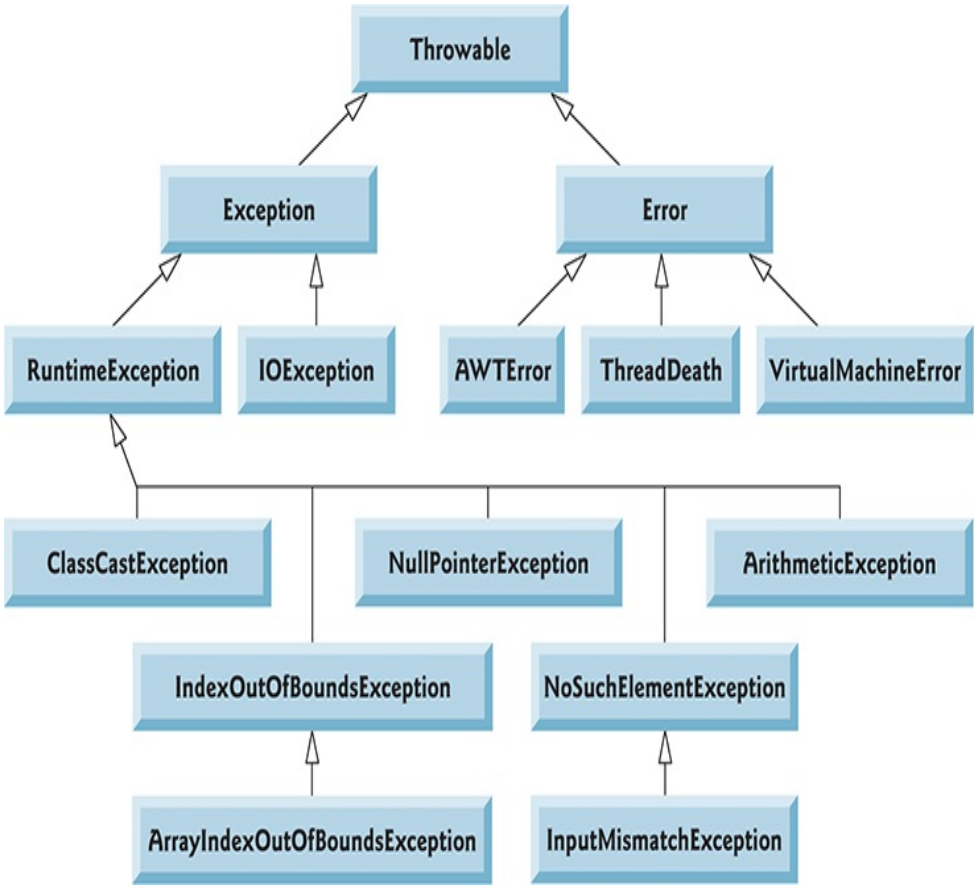

Fig. 11.4

Portion of class `Throwable`'s inheritance hierarchy.

Description

The Java exception hierarchy contains hundreds of classes. Information about Java's exception classes can be found throughout the Java API. You can view `Throwable`'s documentation at

```
http://docs.oracle.com/javase/8/docs/api/java/lang/Th
```

From there, you can look at this class's subclasses to get more information about Java's `Exceptions` and `Errors`.

Checked vs. Unchecked Exceptions

Java distinguishes between **checked exceptions** and **unchecked exceptions**. This distinction is important, because the Java compiler enforces special requirements for *checked* exceptions (discussed momentarily). An exception's type determines whether it's checked or unchecked.

`RuntimeExceptions` Are Unchecked Exceptions

All exception types that are direct or indirect subclasses of `RuntimeException` (package `java.lang`) are *unchecked* exceptions. These are typically caused by defects in your program's code. Examples of unchecked exceptions include:

- `ArrayIndexOutOfBoundsExceptions` (discussed in Chapter 7)—You can avoid these by ensuring that your array indices are always greater than or equal to 0 and less than the array's `length`.
- `ArithmeticExceptions` (shown in Fig. 11.3)—You can avoid the `ArithmeticException` that occurs when you divide by zero by checking the denominator to determine whether it's 0 *before* performing the calculation.

Classes that inherit directly or indirectly from class `Error` (Fig. 11.4) are *unchecked*, because `Error`s typically are unrecoverable, so your program should not even attempt to deal with them. For example, the documentation for `VirtualMachineError` says that these are "thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating." At this point, there's nothing your program can do.

Checked Exceptions

All classes that inherit from class `Exception` but *not* directly or indirectly from class `RuntimeException` are considered to be *checked* exceptions. Such exceptions are typically caused by conditions that are not under the control of the program—for example, in file processing, the program can't open a file if it does not exist.

The Compiler and Checked Exceptions

The compiler checks each method call and method declaration to determine whether the method throws a checked exception. If so, the compiler verifies that the checked exception is *caught* or is *declared* in a `throws` clause—this is known as the **catch-or-declare requirement**. We show how to catch or declare checked exceptions in the next several examples.

Recall from Section 11.3 that the `throws` clause specifies the exceptions a method throws. Such exceptions are not caught in the method's body. To satisfy the *catch* part of the *catch-or-declare requirement*, the code that generates the exception must be wrapped in a `try` block and must provide a catch `handler` for the checked-exception type (or one of its superclasses). To satisfy the *declare* part of the catch-or-declare requirement, the method containing the code that generates the exception must provide a `throws` clause containing the checked-exception type after its parameter list and before its method body. If the catch-or-declare requirement is not satisfied, the compiler will issue an error message. This forces you to think about the problems that may occur when a method that throws checked exceptions is called.

Error-Prevention Tip 11.2

You must deal with checked exceptions. This results in more robust code than would be created if you were able to simply ignore them.

Common Programming Error 11.2

If a subclass method overrides a superclass method, it's an error for the subclass method to list more exceptions in its `throws` *clause than the superclass method does. However, a subclass's* `throws` *clause can contain a subset of a superclass's* `throws` *clause.*

Software Engineering Observation 11.7

If your method calls other methods that throw checked exceptions, those exceptions must be caught or declared. If an exception can be handled meaningfully in a method, the method should catch the exception rather than declare it.

Software Engineering Observation 11.8

Checked exceptions represent problems from which programs often can recover, so programmers are required to deal with them.

The Compiler and Unchecked Exceptions

Unlike checked exceptions, the Java compiler does *not* examine the code to determine whether an unchecked exception is caught or declared. Unchecked exceptions typically can be *prevented* by proper coding. For example, the unchecked `ArithmeticException` thrown by method `quotient` (lines 9–12) in Fig. 11.3 can be avoided if the method ensures that the denominator is not zero *before* performing the division. Unchecked exceptions are *not* required to be listed in a method's `throws` clause—even if they are, it's *not* required that such exceptions be caught by an application.

Software Engineering Observation 11.9

Although the compiler does not enforce the `catch`*-or-declare requirement for unchecked exceptions, provide appropriate exception-handling code when it's known that such exceptions might occur. For example, a program should process the* `NumberFormatException` *from* `Integer` *method* `parseInt`*, even though* `NumberFormatException` *is an indirect subclass of* `RuntimeException` *(and thus an unchecked exception). This makes your programs more robust.*

Catching Subclass

Exceptions

If a `catch` handler is written to catch *superclass* exception objects, it can also catch all objects of that class's *subclasses*. This enables `catch` to handle related exceptions polymorphically. You can catch each subclass individually if they require different processing.

Only the First Matching `catch` Executes

If *multiple* `catch` blocks match a particular exception type, only the *first* matching `catch` block executes when an exception of that type occurs. It's a compilation error to catch the *exact same type* in two different `catch` blocks associated with a particular `try` block. However, there can be several `catch` blocks that match an exception—i.e., several `catch` blocks whose types are the same as the exception type or a superclass of that type. For example, we could follow a `catch` block for type `ArithmeticException` with a `catch` block for type `Exception`—both would match `ArithmeticExceptions`, but only the first matching `catch` block would execute.

Common Programming

Error 11.3

Placing a `catch` *block for a superclass exception type before other* `catch` *blocks that catch subclass exception types would prevent those* `catch` *blocks from executing, so a compilation error occurs.*

 Error-Prevention Tip 11.3

Catching subclass types individually is subject to error if you forget to test for one or more of the subclass types explicitly; catching the superclass guarantees that objects of all subclasses will be caught. Positioning a `catch` *block for the superclass type after all other subclass* `catch` *blocks ensures that all subclass exceptions are eventually caught.*

 Software Engineering Observation 11.10

In industry, throwing or catching type `Exception` *is discouraged—we use it in this chapter simply to demonstrate exception-handling mechanics. In subsequent chapters, we generally throw and catch more specific exception types.*