Appendix C
Throwing and Catching Java Exceptions

An **exception** is a Java `Object` that is used to indicate abnormal conditions. When a method detects an abnormal condition, it can create an exception object, and pass this object upwards to the place where the method was activated—the whole process is called **throwing** an exception.

When an exception is thrown, the problem can sometimes be corrected—a technique called **catching the exception**. Other times, the problem is too serious to correct, and it will remain uncaught, eventually causing the Java runtime system to print an error message and halt the program.

How to Throw an Exception

Some Java statements automatically throw an exception when they are incorrectly used. For example, the evaluation of the expression `(42/0)` will throw an `ArithmeticException` because an integer division by zero is illegal. At other times, a method may detect a problem itself and throw an exception to indicate this problem. For example, Chapter 1 has a method with a specification that begins like this:

- **celsiusToFahrenheit**
  `public static double celsiusToFahrenheit(double c)`
  Convert a temperature from Celsius degrees to Fahrenheit degrees.
  Parameters:
  `c` – a temperature in Celsius degrees
  Precondition:
  `c >= -273.16`.

It is a programming error to call `celsiusToFahrenheit` with an argument that is below –273.16. In such a case, the `celsiusToFahrenheit` method will detect that the precondition has been violated and throw an `IllegalArgumentException`, with statements such as these:

```
if (c < -273.16)
   throw new IllegalArgumentException("Temperature too small.");
```

The general form for throwing an exception uses the keyword `throw`, following this pattern:

```
throw new ____________________ ("____________________");
```

This is the type of the exception that we are throwing.

This is an error message that will be passed as part of the exception.

Figure C.1 shows a list of common exceptions that a programmer may use to indicate problems. In Java terminology, these are actually called `Throwable` objects, and the list also indicates a further classification for each kind of `Throwable` object. We'll look at the meaning of that further classification on page 734.

FIGURE C.1 Partial List of Java's `Throwable` Objects

Name of the Class	Typical Meaning	Examples or Discussion	Further Classification
`ClassCast Exception`	Attempting to use a reference as if it referred to an object of some class—but it doesn't.	Page 75	`Exception`
`CloneNotSupported Exception`	The `clone` method was activated in a class that forgot to implement Cloneable	Page 79	`Exception (not Runtime)`
`EmptyStack Exception*`	Tried to `pop` or `peek` from an empty `Stack`.	Page 281	`Runtime Exception`
`IllegalArgument Exception`	An argument to a method violates the precondition.	Page 8	`Runtime Exception`
`IllegalState Exception`	A method was activated when the object was in a state that violates the method's precondition.	Page 137	`Runtime Exception`

`IndexOutOfBounds Exception`	An array index was used beyond the array's capacity.	`int[ ] a;`
		`a = new int[40];`
		`a[42] = 0;`
			`Runtime Exception`

`IOException`	An input or output error.	Trying to read after the end of a file.	`Exception (not Runtime)`

`NegaviteArraySize Exception`	Allocating an array with a negative number of components.	`int[ ] a;`
		`int s = -1;`
		`a = new int[s];`
			`Runtime Exception`

`NoSuchElement Exception*`	Attempting to get another element out of an `Iterator` or other collection class when there are no more elements.	Page 269 or Page 327	`Runtime Exception`
`NullPointer Exception`	Attempting to access a method or instance variable of the null reference.	Page 51	`Runtime Exception`
`OutOfMemory Error`	The heap has no more memory.	Page 106	`Error`
`StackOverflow Error`	The execution stack has no more memory.	Page 381	`Error`
`UnsupportedOperation Exception`	A method of a class is not being provided by the class's programmer.	Page 268	`Runtime Exception`

*`EmptyStackException` and `NoSuchElementException` are part of `java.util`.

See http://www.cs.colorado.edu/~main/java.html for a listing of all Java `Throwable` objects.

The `RuntimeException` and `Error` Classes

Most of the exceptions listed in Figure C.1 have a "further classification" of `RuntimeException` or `Error`. These are two particular kinds of `Throwable` objects, with conventional meanings for programmers:

- `RuntimeException`: These tend to be exceptions that are caused by programming mistakes, such as the violation of a precondition.
- `Error`: These are problems with resources and the Java Virtual Machine. For example, running out of memory is a resource problem. Not having a needed class is an example of a problem with the Java Virtual Machine.

Catching an Exception

Sometimes an exception occurs, but the programmer has some way to handle the problem. In this situation, the programmer can use `try-catch` blocks of the following form:

```
try
{
    Statements that might cause an exception to be thrown.
}
catch ( Type of the possible exception  e)
{
    These are statements to handle the problem. Within these
    statements, the name e refers to the exception object. For example,
    e.toString( ) is a message that's attached to the exception.
}
```

In this example, the variable name `e` was used for the name of the exception, but you can choose whatever name you like. The "type of the possible exception" is one of the exception data types such as `CloneNotSupportedException`.

As a specific example, consider the following code that tries to fill an array with clones of a given object. However, if the object is not clonable, then the code fills the array with references to the actual object (rather than clones of the object):

```
// In this code, obj is a non-null reference to a Java Object and copies is an array of objects.
// The variable i is an int.
try
{
   for (i = 0; i < copies.length; i++)
      copies[i] = obj.clone( );
}
catch (CloneNotSupportedException e)
{  // Fill the array with references to the actual object instead of clones.
   for (i = 0; i < copies.length; i++)
      copies[i] = obj;
}
```

If a section of code has the possibility of throwing several different types of exceptions, then the `try`-block may be followed by several different `catch`-blocks. After the final catch-block, there can be one more block that starts with the keyword `finally`. The `finally`-block is executed at the end, whether or not the exception is caught. For example, the following format has two catch-blocks and a `finally`-block:

```
try
{
    Statements that might cause an exception to be thrown.
}
catch ( Type of the first possible exception  e1)
{
    These are statements to handle the problem. Within these
    statements, the name e1 refers to the exception object. For example,
    e1.toString( ) is a message that's attached to the exception.
}
catch ( Type of the second possible exception  e2)
{
    These are statements to handle the problem. Within these
    statements, the name e2 refers to the exception object.
}
finally
{
    These are statements that will always be executed after the above
    try- and catch-blocks. Note that this code is executed in all cases:
    when no exception occurs, when an exception is thrown and caught,
    or when an uncaught exception is thrown.
}
```

The `throws` Clause

Java has some exceptions that are neither a `RuntimeException` nor an `Error`. The examples in our list are `CloneNotSupportedException` and `IOException`, but there are several dozen more and programmers can even create new classes of exceptions. You must follow a special rule when you write a statement that might throw one of these exceptions:

> When a method includes a statement that might throw an exception that is neither `RuntimeException` nor an `Error`, then there are two possibilities: (1) Catch the exception, or (2) Include the name of the exception in a `throws` clause after the heading of the method.

The format of a `throws` clause is the keyword `throws` followed by the type of the possible exception. If there are several possible exceptions, then their types may be written separated by commas in a single `throws` clause. For example, the following `main` method has a `throws` clause indicating that it may throw a `CloneNotSupportedException` or an `IOException`:

```
public static main(String[ ] args)
throws CloneNotSupportedException, IOException
{
    These are statements that might throw an uncaught
    CloneNotSupportedException or an uncaught IOException.
}
```

Further Information

For a complete list of Java exceptions, follow the `Exceptions` link in the file
`http://www.cs.colorado.edu/~main/java.html`