20 Exceptions

1 Handling exceptions

A program will sometimes inadvertently ask the machine to do something which it cannot reasonably do, such as dividing by zero, or attempting to access a non-existent array component, or attempting to create a file when there is no more room on the disk. Such potentially fatal errors are called *exceptions*. When an exception arises, Java usually applies a default mechanism which typically prints out some information about the exception and aborts the program. For example, the following program generates the output shown:

```
class DefaultExceptionHandling {
   public static void main(String[] args) {
      int x = 5/0;  /Oh, Oh!
      System.out.println("You won't see this!");
   }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DefaultExceptionHandling.main(DefaultExceptionHandling.java:3)
```

The output is printed by the Java run-time system when it encounters the attempted division by zero in the first statement in `main()`. It then aborts the program and so the message `"You won't see this!"` never appears. The first line of the output states that an arithmetic exception occurred, and describes the particular problem as an attempted division by zero. The

second line states that the exception occurred in method `main()` of class `DefaultExceptionHandling`, and (in brackets) that the offending statement occurs in line 3 of the file called `DefaultExceptionHandling.java`.

Aborting the program at the first sign of trouble is often needlessly drastic. For example, if a disk is full it might be sensible to invite the user to delete some unwanted files. Fortunately, Java provides a mechanism whereby the programmer can specify what should happen when an exception arises. Indeed, for some types of exceptions, most typically those associated with file handling, Java virtually *requires* that the programmer includes code to handle the exception. A piece of code that handles an exception is called an *exception handler*. To handle an exception that may arise in a particular piece of code, you enclose the code in what is called a *try-block*, and follow this immediately with the exception-handling code wrapped up in what is called a *catch-block*. These blocks occur together and are known collectively as a try-catch block. The syntax of a try-catch block is:

```
try {
    ... code that may give rise to an exception ...
}
catch (Exception e) {
    ... code for handling exceptions  ...
}
```

The words `try` and `catch` are keywords of the language. `Exception` is the name of a pre-defined class in Java and `e` is a variable of type `Exception` (the name `e` is freely chosen). For example, the following program includes an exception handler for an attempted division by zero (we explain its operation shortly):

```
class ExceptionHandling {
   public static void main(String[] args) {
      try {
         int x = 5/0; // Oh, Oh!
         System.out.println("You won't see this!");
      }
      catch (Exception e) {
         System.out.println("An arithmetic error occurred!");
      }
      System.out.println("Bye Bye!");
   }
}
```

(The program may be rejected by some compilers because they are smart enough to recognise `5/0` as rubbish. In that case, the zero divisor can be disguised by introducing an integer variable `k` and replacing `5/0` with `5/(k-k)`.) When the program is run it generates the following output:

```
An arithmetic error occurred!
Bye Bye!
```

When an exception occurs in a try-block, any remaining code in the try-block is abandoned. For example, the above program does not display the message `"You won't see this!"` because the statement that precedes it generates an exception by attempting to divide by zero. As soon as that happens, the catch-block is executed, and in this case displays a simple message. A try-catch block is a statement, no different in status from a while- or an if-statement – after it has completed, execution continues at the statement that follows (in the example above this is the statement which displays the message `"Bye Bye!"`). If no exception arises in the try-block, the code in the catch-block is not executed, and execution continues with the statement following the try-catch block.

When an exception occurs, the system creates an object of type **`Exception`** which encapsulates information about the source of the problem. A reference to the object is assigned to the variable `e` declared at the entrance to the catch-block. The `Exception` class has two methods of interest:

```
String getMessage()
void printStackTrace()
```

`e.getMessage()` returns a short string describing the problem that gave rise to `e`, while **`e.printStackTrace()`** prints more detailed information. For example, if the statement `system.out.println("An arithmetic error occurred!")` in the catch-block above is replaced with `e.printStackTrace()`, the program generates the following output:

```
java.lang.ArithmeticException: / by zero
     at ExceptionHandling.main(ExceptionHandling.java:5)
Bye Bye!
```

Although the code in the try- and catch-blocks above is quite simple, in general it can be as complex as you like. If the code in a try-block invokes a method, any exception arising in the method is treated as though it arose in the try-block. For example, the program below results in the message `"Exception in zeroDiv!"` (and no other) being displayed.

```
class ExceptionInMethod {

   static void zeroDiv() {
    System.out.println(5/0);  // Oh, Oh!
    System.out.println("You won't see this!");
   }

   public static void main(String[] args)  {
     try {
        zeroDiv();  // an exception will be generated
        System.out.println("You won't see this!");
```

```
        }
        catch (Exception e) {
          System.out.println("Exception in zeroDiv!");
        }
      }
  }
```

To be precise, we should have said that any *uncaught* exception arising in a method called from within a try-block is treated as though it arose in the try-block. A method may well provide its own handler, and if so that will take precedence.

Instead of saying that a piece of code generates or gives rise to an exception, we sometimes say the code *throws* or *raises* an exception. A piece of code which provides a handler for an exception is said to *catch* or *trap* the exception.

Avoid exceptions!

A potential error can be handled by testing in advance and avoiding the problem, or by letting it happen and catching the exception. Above we catered for a possible zero-divide by exception handling. We might alternatively have tested in advance with an if-statement:

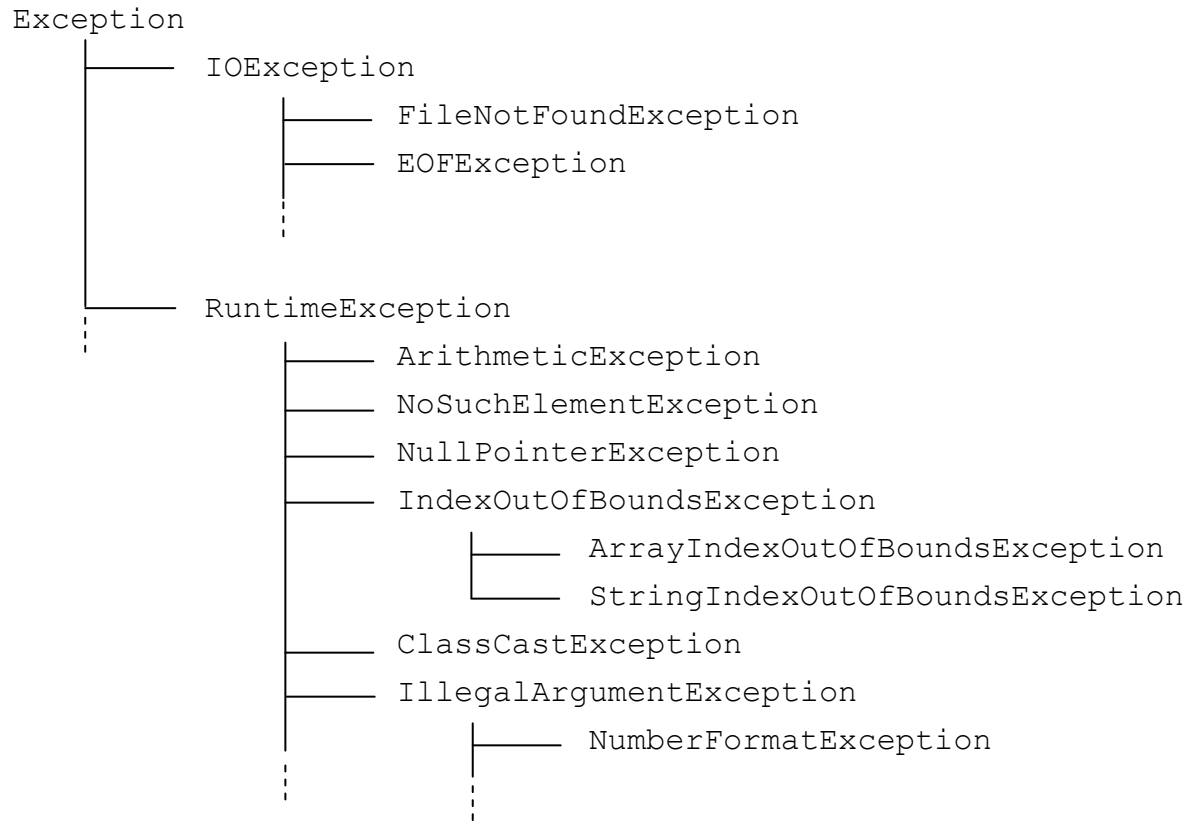
```
if (y!=0)  x = x/y;
else System.out.println("Attempted zero-divide.");
```

Whenever you have the choice, test for possible problems in advance and handle them before they happen. Exceptions are clumsy and expensive coding devices and should only be used when there is no alternative, primarily when the programming language requires their use. Our interest in them is motivated primarily as a preparation for studying files where their use is mandatory.

2 Exception types

Exceptions are categorised into several types of which the most important are **`IOException`** and **`RuntimeException`**. Exceptions of type `IOException` are more informally called *I/O exceptions*. They arise when an input/output operation fails, such as an attempt to write to a USB flash drive which is already full. Exceptions of type `RuntimeException` are more informally called *run-time exceptions*. They arise because of a failure of the program to handle data correctly, such as attempting to divide by zero. An understanding of I/O exceptions is important because Java requires the programmer to handle all I/O exceptions that can arise, and that means that most code that engages in input/output will be in a try-block. Run-time exceptions, however, are of minor importance and knowledge of them is of little significance other than in understanding error messages displayed by the system when your program fails.

I/O exceptions and run-time exceptions are further categorised as depicted below:



The two important types of I/O exception are:

`FileNotFoundException`	A file-not-found exception, caused by an attempt to access a file that cannot be located by the system.
`EOFException`	An end-of-file exception, caused by an attempt to read more data from a file than it contains (EOF stands for *end of file*).

We will learn more about these later. There are I/O exceptions that do not belong to any particular sub-type of `IOException`. The meaning of the various run-time exception types is as follows:

`ArithmeticException`	An arithmetic exception, caused by an attempt to perform an illegal arithmetic operation such as an attempted division by zero.
`IndexOutOfBoundsException`	An index out-of-bounds exception which is comprised of the two types that follow.
`ArrayIndexOutOfBoundsException`	An array index out-of-bounds exception, caused by an attempt to access a non-existent array element, as in `b[-1]`.
`StringIndexOutOfBoundsException`	A string index out-of-bounds exception,

	caused by an attempt to access a non-existent string element, as in `"Hi!".charAt(-1)`.
NullPointerException	A null pointer exception, caused by an attempt to reference an object via a reference variable that contains `null`, as in `Point p; p.x=0;`
ClassCastException	A class cast exception, caused by an attempt to typecast incorrectly (in certain circumstances).
IllegalArgumentException	An illegal argument exception. Some methods in the Java library throw this when they have been invoked with one or more improper arguments (e.g. an array argument is expected to be sorted, but isn't).
NumberFormatException	A number format exception, which is a type of illegal argument exception thrown by some methods in the Java library when an argument of type `String` is expected to represent a number of some kind, but doesn't.

We can restrict a catch block to handling just one particular type of exception by replacing `Exception` at its entrance with the name of the exception type. For example, the following includes an exception handler for I/O exceptions only – run-time exceptions will be handled by the default handler:

```
try {
   .... code that may give rise to an exception ....
}
catch (IOException e) {
   e.printStackTrace();
}
```

All exception types (such as `FileNotFoundException` and `IOException`) are names of predefined classes in Java. Like `Exception`, they have methods `printStackTrace()` and `getMessage()`. A handler for a particular exception type also handles subtypes of the exception. For example, a file-not-found exception arising in the try-block above will be handled by the catch-block.

A run-time exception is nearly always a result of poor programming – if the divisor in a division has the potential to be zero, for example, the programmer should check for it and avoid the problem. It is not necessary (or usual) to supply exception handlers for run-time exceptions – the system will always supply a default handler (as in the example at the start of the chapter). Perhaps one of the few instances where run-time exceptions are appropriate is in detecting format errors in strings. Java supplies method `Double.parseDouble(s)` which returns a

value of type `Double` that is represented by string `s`. If `s` does not represent a real number `parseDouble()` throws a `NumberFormatException`. `Integer.parseInt()` behaves similarly for integers.

Example 1: temperature converter

We write a program which converts Fahrenheit temperatures to Centigrade and vice versa. The user types in a real number followed by C (for Centigrade) or F (for Fahrenheit) and the temperature is displayed on the alternative scale. The following is a typical interaction at the keyboard.

```
Welcome to the Centigrade-Fahrenheit converter.
212F
=100.0C
59;8C
Sample input: 27.23C or 98.2F
59.8C
=139.64F
45.6K
End with C (centigrade) or F (Fahrenheit)
```

The program cannot use `Console.readDouble()` to read the temperature because it does not end with a space or end-of-line, but with a letter. We have to read a string and decompose it into a real number (the temperature) and a character (‘C’ or ‘F’).

```
class TempConverter {
    public static void main(String[] args) {
        System.out.println("Welcome to the Centigrade-Fahrenheit converter.");
        while (!Console.endOfFile()) {
            try {
                String s = Console.readToken();  // the user's input
                char scale = s.charAt(s.length()-1); // 'C' or 'F' (?)
                String number = s.substring(0,s.length()-1);  // in real format (?)
                double temp = Double.parseDouble(number); // temperature as Double
                if (scale=='c' || scale == 'C')
                    System.out.println("=" + (32.0+temp*1.8) + "F");
                else if (scale=='f' || scale == 'F')
                    System.out.println("=" + ((temp-32.0)/1.8) + "C");
                else
                    System.out.println ("End with C (centigrade) or F (Fahrenheit)");
            }
            catch (NumberFormatException e) {
                System.out.println ("Sample input: 27.23C or 98.2F");
            }
        }
    }
```

```
}
```

Notational convention

The Java class library provides various methods and constructors that may generate I/O exceptions and hence must be invoked from within try-blocks. Whenever we introduce them, we will indicate the exceptions they may generate by a *throws-clause* in the header. The following method and constructor are typical:

```
void close() throws IOException
FileReader(String) throws FileNotFoundException
```

Some methods may generate many different kinds of I/O exception, not all of them classified. When that is the case and an end-of-file or a file-not-found exception is among the possibilities, we will indicate that as in

```
char readChar() throws IOException, EOFException
```

This declares that `readChar()` may generate an I/O exception of potentially any type, including in particular an end-of-file exception.

3 Multiple exception handlers

Instead of one monolithic exception handler, we can provide several, each one tailored to a particular type of exception. We do this by following a try-block with a succession of catch-blocks, one for each type of exception of interest. The following is typical:

```
try {
    ... code that may give rise to an exception ...
}
catch (ArithmeticException e) {
    ... handle an arithmetic exception  ...
}
catch (FileNotFoundException e) {
    ... handle a file-not-found exception  ...
}
catch (IOException e) {
    ... handle any I/O exception other than a file-not-found exception ...
}
```

(The exception variable has been called `e` in all catch-blocks above, but different names could have been chosen.) The code in at most one catch-block is executed. If no exception arises then no catch-block is executed. Otherwise control passes to the first catch-block that deals with the exception type. If no catch-block is provided for the type, the default handler is applied. When a handler is provided for an exception and one or more of its sub-types, the handlers for the sub-types must be placed first. For example, in the preceding code the catch-block for file-not-

found exceptions is placed before the one for I/O exceptions. The following trivial program illustrates multiple catch-blocks:

```
class ExceptionHandling2 {
   public static void main(String[] args) {
      try {
         int k = (int)(Math.random()*2);   // k = 0 or 1
         k = 5/k;   // Oh, Oh if k=0
         char ch = "X".charAt(k);  // Oh, Oh if k=1
      }
      catch (ArithmeticException ex) {
         System.out.println("An attempted zero divide!");
      }
      catch (StringIndexOutOfBoundsException ex) {
         System.out.println("A string index is out of bounds!");
      }
   }
}
```

If `k` is assigned 0 the first catch block is executed, and if `k` is assigned 1 the second catch-block is executed. In this example, the behaviour of the program would not be altered by replacing `StringIndexOutOfBoundsException` in the second catch-block with any of its parent types, such as `IndexOutOfBoundsException` or `RuntimeException`. However, it is probably best to prefer the more precise `StringIndexOutOfBoundsException`.

There is no need to provide a family of tailored exception handlers unless it brings some advantage. In particular, a single handler for all I/O exceptions often suffices. When a block of code may give rise to several exceptions (of the same or different types), it is best to wrap up the entire code in a single try block followed by one or more catch-blocks. Lots of small try-catch blocks can make the code difficult to read.