Managing Errors and Exceptions

13.1 INTRODUCTION

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. *Errors* are the wrongs that can make a program go wrong.

An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

13.2 TYPES OF ERRORS

Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

Compile-Time Errors

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the .class file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

Program 13.1 Illustration of compile-time errors

```
/* This program contains an error */

class Error1
{
        public static void main(String args[ ])
        {
              System.out.println("Hello Java!")    // Missing;
        }
}
```

The Java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of print statement in Program 13.1, the following message will be displayed in the screen:

```
Error1.java :7: ';' expected
System.out.println ("Hello Java!")
^

1 error
```

We can now go to the appropriate line, correct the error, and recompile the program. Sometimes, a single error may be the source of multiple errors later in the compilation. For example, use of an undeclared variable in a number of places will cause a series of errors of type "undefined variable". We should generally consider the earliest errors as the major source of our problem. After we fix such an error, we should recompile the program and look for other errors.

Most of the compile-time errors are due to typing mistakes. Typographical errors are hard to find. We may have to check the code word by word, or even character by character. The most common problems are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments / initialization
- Bad references to objects
- Use of = in place of == operator
- And so on

Other errors we may encounter are related to directory paths. An error such as

```
javac : command not found
```

means that we have not set the path correctly. We must ensure that the path includes the directory where the Java executables are stored.

Run-Time Errors

Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid range or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Using a null object reference as a legitimate object reference to access a method or a variable
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string
- And many more

When such errors are encountered, Java typically generates an error message and aborts the program. Program 13.2 illustrates how a run-time error causes termination of execution of the program.

Program 13.2 ***Illustration of run-time errors***

```
class  Error2
{
    public static void main(String args[ ])
    {
        int a = 10;
        int b =  5;
        int c =  5;

        int x = a/(b-c);     // Division by zero
        System.out.println("x = " + x);

        int y =  a/(b+c);
        System.out.println("y = " + y);
    }
}
```

Program 13.2 is syntactically correct and therefore does not cause any problem durin compilation. However, while executing, it displays the following message and stops withou executing further statements.

```
java.lang.ArithmeticException:  /  by  zero
        at Error2.main(Error2.java:10)
```

When Java run-time tries to execute a division by zero, it generates an error condition, which causes the program to stop after displaying an appropriate message.

13.3 EXCEPTIONS

An *exception* is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it(i.e., informs us that an error has occurred).

If the exception object is not caught and handled properly, the interpreter will display an error message as shown in the output of Program 13.2 and will terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as *exception handling*.

The purpose of exception handling mechanism is to provide a means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

1. Find the problem (*Hit* the exception).
2. Inform that an error has occurred (*Throw* the exception)
3. Receive the error information (*Catch* the exception)
4. Take corrective actions (*Handle* the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.

When writing programs, we must always be on the lookout for places in the program where an exception could be generated. Some common exceptions that we must watch out for catching are listed in Table 13.1.

Table 13.1 Common Java Exceptions

Exception Type	Cause of Exception
`ArithmeticException`	Caused by math errors such as division by zero
`ArrayIndexOutOfBoundsException`	Caused by bad array indexes
`ArrayStoreException`	Caused when a program tries to store the wrong type of data in an array
`FileNotFoundException`	Caused by an attempt to access a nonexistent file

(Continued)

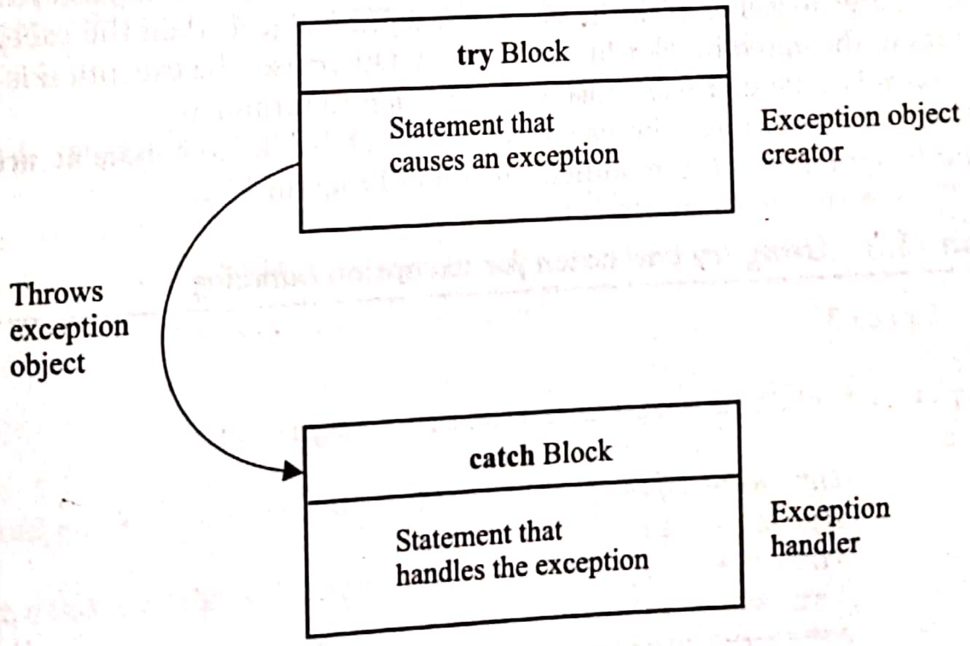
Table 13.1 *(Continued)*

Exception Type	Cause of Exception
`IOException`	Caused by general I/O failures, such as inability to read from a file
`NullPointerException`	Caused by referencing a null object
`NumberFormatException`	Caused when a conversion between strings and number fails
`OutOfMemoryException`	Caused when there's not enough memory to allocate a new object
`SecurityException`	Caused when an applet tries to perform an action not allowed by the browser's security setting
`StackOverflowException`	Caused when the system runs out of stack space
`StringIndexOutOfBoundsException`	Caused when a program attempts to access a nonexistent character position in a string

13.4 SYNTAX OF EXCEPTION HANDLING CODE

The basic concepts of exception handling are *throwing* an exception and *catching* it. This is illustrated in Fig. 13.1.

Fig. 13.1



Exception handling mechanism

Java uses a keyword **try** to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword **catch** "catches" the exception "thrown" by the try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple **try** and **catch** statements:

```
..........
..........
try
{
    statement;      // generates an exception
}
catch (Exception-type e)
{
    statement;      // processes the exception
}
..........
..........
```

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

The catch block too can have one or more statements that are necessary to process the exception. Remember that every **try** statement should be followed by *at least one* catch statement; otherwise compilation error will occur.

Note that the catch statement works like a method definition. The catch statement is passed a single parameter, which is reference to the exception object thrown (by the try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.