Mathematics defines two types of functions, total functions and partial functions. In total functions the value of the function is defined for all values of the variables or arguments provided to the function. In case of partial functions, there are domains of variable values, where the function behavior becomes indeterminate.

Similar to mathematical concept of total and partial functions, Java has methods, which can be categorized as total methods and partial methods. The total methods are such that there is no restriction on the variables passed to the method at run time, as method would handle all errors related to the data passed to it. This is akin to a total mathematical function being defined for all values of the variables. A partial Java method on the other hand “requires” user to refrain from passing certain variable values as arguments, as the method is not designed to handle such values. If a forbidden value were passed to a partial Java method then error condition would result. A simple example of a partial method is a method called divide (Listing 12.1), which takes two integers num and den as arguments and returns the result num/den. The pre-condition to be followed in using this method is that user never passes a zero value for den to the method.

```java
public class DivideProblem1
{
    /*
     * pre-condition: The value of den passed to the method shall never be zero.
     * Returns the integer result after dividing argument num by argument den.
     */
    public static int divide (int num, int den)
    {
        return num/den;
    }
} //Listing 12.1
```

The method divide in Listing 12.1 has undefined result for den equal to zero and would require error handling when user violates the pre-condition. Traditional language like C could pass an extra argument, a pointer to a partial method like divide to carry the results of the division of num and den. The method could return a zero if pre-condition is violated. In that case the method would not alter the value of the pointee of the pointer passed to it. If denominator is non-zero then the method

1 It just so happens that Java would crash the program if a zero value for den were passed.
could return a value of one and store the result of division in the pointee of the pointer passed to it. Listing 12.2 shows the details.

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int divide(int num, int den, int * result)
{
    if(den == 0)
        return 0;
    else
    {
        *result = (num/den);
        return 1;
    }
}

void main()
{
    int * result = (int*) malloc(sizeof(int));
    int num = 5;
    int den = 0;

    if(!divide(num,den,result))
        printf("The illegal denominator of value = %d has been passed to the method divide",den);
    else
    {
        printf("\nThe division %d/%d = %d\n",num,den,*result);
        num = 25;
        den = 5;

        if(!divide(num,den,result))
            printf("The illegal denominator of value = %d has been passed to the method divide",den);
        else
            printf("\nThe division %d/%d = %d\n",num,den,*result);
    }
} //Listing 12.2

Figure 12.1 shows the results of Listing 12.2.

The illegal denominator of value = 0 has been passed to the method divide
The division 25/5 = 5
Press any key to continue

FIG. 12.1
In Listing 12.2, the method divide tests if the denominator is zero, in which case the method returns a zero, and the pointee of int pointer result is left unaltered. However, if denominator is non-zero, then the pointee of pointer result is set equal to the result of division, and it is printed in the caller method.

Due to security considerations, Java does not allow pointers to primitives, and for the same reason all wrappers to primitives are also immutable. Therefore technique used in C programming for error handling (where an error code is returned) cannot be used in Java without elaborate measure, such as designing a method header as given below.

```java
boolean divide (int num, int den, StringBuffer result)
```

In a divide method like above, if den is not zero, then method would return a true, and the value num/den would be appended to (an initially empty) StringBuffer object result. Using the algorithm similar to the one shown in main method in Listing 12.2, the caller method can correctly parse the result of the division.

The use of C like technology for error handling is tedious, as it requires the client to write extra code for error checking. Therefore inventor of Java and other Object Oriented Languages designed a technology, which allows the methods to return an error condition to the caller method without using the traditional error code return mechanism. This technology is called exception handling/error handling. Side bar 12.1 provides few examples of error or exception conditions.

**Sidebar 12.1**

- In an integer arithmetical division, user provides a zero value for the denominator.
- In a file input, the wrong file name or non-existent disk drive is provided.
- When program expects input of a numeric, a string composed entirely of alphabets is entered.
- When entering a person’s name in the customer database, only digits are entered.
- Disk drive has a physical crash during the data read/write process.
- Programmer makes a method call using an object reference, which is a null pointer.
- The program tries to access an out of bounds array index.
So far we have been using words error and exception as if they are same thing. Java, however, makes a subtle difference between them and designs group of classes accordingly.

Error is a program condition, which when manifested, is generally not correctable by adding a piece of code in a class or in the client code. For example if during execution of a Java program, the JVM breaks down, then user code cannot handle such error, and JVM would shut down the program after informing user of the error.

On the other hand exceptions are conditions that are to some degree controllable, and in some cases correctable by adding an exception handling code in the classes and/or in the client program. In this chapter we elucidate exception handling by examples and discuss error handling. Diagram in Figure 12.2 shows Java’s top-level design of classes for error/exception handling.

![Diagram of Java's top-level design of classes for error/exception handling](image)

At the top of hierarchy (derived from class java.lang.Object) is the class Throwable, which also implements the interface serializable. Figure 12.2 shows all the methods/constructors provided and overridden in the class Throwable. The two subclasses that extend Throwable are java.lang.Error and java.lang.Exception.

By now you already have some experience with messages that appear during your program execution as a result of Java virtual machine informing you, of error
incurred during the program. We show an example in Listing 12.3 where a class
called SomeExceptions shows the Error message generated by JVM, when the user
attempts division by zero.

```java
import javax.swing.*;

public class SomeExceptions
{
    public static void main(String[] args)
    {
        int numerator = 0;
        int denominator = 1;
        String Input =
        JOptionPane.showMessageDialog("Please enter the numerator!");
        numerator = Integer.parseInt(Input);
        Input =
        JOptionPane.showMessageDialog("Please enter the NON-ZERO denominator!");
        denominator = Integer.parseInt(Input);
        JOptionPane.showMessageDialog(null, "The result of division is = "+
+ new Integer(numerator/denominator).toString());
        System.out.println("The result of division is = "+
+ new Integer(numerator/denominator).toString());
        System.exit(0);
    }
}
```

Listing 12.3

The above program will give expected results if user enters a non-zero denominator.
However, Java crashes the program when a zero denominator is entered and all the
code from the point of division by zero and on is skipped. The message on the
standard output is shown below in Figure 12.3.

FIG. 12.3

Java virtual machine crashes the program and points to the line number where it
found the division by zero error. It tells us that a division by zero was attempted at
the line number 15 in the file SomeExceptions.java in the method main. (Java refers
to methods as threads).
This type of error checking mechanism, which is in-built in Java, is absent\(^2\) in C++ and C, where programmer is responsible for all error checking. Java virtual machine accomplishes the error checking in Listing 12.3 by throwing (returning) an object of class java.lang.ArithmeticException, which if uncaught will end the program. This process is called exception throwing. Another name for the same process is exception raising. There are two points that we would like to make based on the Listing 12.3 (Sidebar 12.2).

### Sidebar 12.2

- Some exception and error handling is in-built in java. All user has to do is to learn to use it.
- Java accomplishes the error checking and exception handling by instantiating an object of a Java class, which closely relates to the error (or exception) and then throwing that object at the error location.

We shall see shortly, what the mechanism of “throwing” an object encompasses. But before we learn to throw objects to recover from errors, we would like to learn, as how to handle thrown exceptions. This is done by putting the error causing code inside a block after the keyword try (referred to as a try block), and then catching any exceptions thrown by that code in a block called catch. The syntax of try – catch blocks is shown as follows (Figure 12.4).

```
try
{
    //One or more program statements that may throw an exception
}
catch(Exception Ex)
{
    //Zero or more statements that execute after thrown object is caught
}
```

**FIG. 12.4**

If an exception condition takes place inside the try block, then an exception object is returned, and control of program is immediately transferred to the first statement in the catch block and all the statements inside try block, after the exception throwing statement are skipped. The catch block is also called an “exception

\(^2\) C++ has a limited implementation of automatic exception handling.
handler”. The try and catch blocks are generally, used as a bonded pairs – at least one catch block following a try block. The catch block acts almost like a method call, in the sense that it takes an object of type Exception as argument and then executes the code inside the catch block. You may surmise a pair of try and catch block as the try block returning a value of type Exception, and passing it to the catch block immediately following it. Understand that an attempt to write a lone try or catch block or changing their prescribed order would be a compile error. To show the above mechanism we modify the code in main of the Listing 12.3 and run the program again (Listing 12.4– We only show the modified version of main.

```java
public static void main(String[] args) {
    int numerator = 0;
    int denominator = 1;
    String Input = JOptionPane.showMessageDialog(null, "Please enter the numerator!");
    numerator = Integer.parseInt(Input);
    Input = JOptionPane.showMessageDialog(null, "Please enter the NON ZERO - denominator!");
    denominator = Integer.parseInt(Input);
    try {
        JOptionPane.showMessageDialog(null, "The result of division is = " + numerator/denominator);
        System.out.println("The result of division is = " + numerator/denominator);
    } catch(Exception e) {
        JOptionPane.showMessageDialog(null, "A Zero denominator has been entered.");
    } System.exit (0);
}
```

Listing. 12.4
When user enters a zero value of denominator, the second statement inside the try block is skipped and control transfer to the catch block, where a pop-up message is displayed. The behavior of Listing 12.4 is different from 12.3 in the sense that now user gets a more descriptive message shown through a JOptionPane shown in Figure 12.5 and the program exits gracefully without a cryptic message that JVM would have produced.

---

3 When the code inside the try block throws a “un-checked exception”, the try clause, if needed, may be followed by a finally clause (missing a catch block). Missing a catch block immediately after a try block is, however, a non-standard practice. We shall discuss finally clause shortly.
FIG. 12.5
Sidebar 12.3 summarizes some rules for using try-catch blocks for exception handling in Java.

Sidebar 12.3
1. Existence of a lone try or catch block is a compile error. There shall be no other code between the try catch blocks (works similar to if/else structure).
2. One or more catch blocks may follow a try block. Java will match the type of object (Exception) thrown in the try block with the objects in the arguments of various catch blocks, and wherever it finds the first best match, it will execute the code in that catch block.
3. In the try block, once the exception is thrown, the code after that point is skipped and java proceeds to execute the first matching catch block.
4. The object thrown by the code in try block must be of type java.lang.Exception. (Throwing the objects of classes Throwable and of type Error is syntactically correct, but there is no added value in doing so).
Types of Exceptions in Java
Java neatly divides the exceptions in two categories:

1. Exceptions that would be checked by the Java compiler (compile time exceptions).
2. Exceptions that would be thrown by Java Virtual Machine (JVM) at run time (also called run time exceptions or un-checked exceptions).

Compile time exceptions are also called **checked** exceptions. Compiler checks the Java code to ascertain that if the method called throws a checked exception then the client of that method is forced to call that method only inside a try block. Compiler does not require the code that could throw a run time or un-checked exceptions to be placed in the try block. The objects of type `java.lang.Error`, and `java.lang.RuntimeException` form the un-checked exception objects. Figure 12.6 expands further on Figure 12.2, and shows schematically, the class hierarchy of checked and un-checked exception classes.

![Hierarchy of exception classes](#)

**FIG. 12.6**
All classes that are of type `RuntimeException` or `Error` form the un-checked exception classes, whereas the remaining classes form the checked exception types.

Listing 12.5 shows as how both checked exceptions, and run time exceptions can manifest, or be handled. The example in Listing 12.5 is a toy example used just for illustration purpose.
public class ExceptionTypes {
    private ExceptionTypes (){}
    /**
     * Method method1 throws an exception of type
     * Exception, which is a checked exception.
     */
    public static int method1(int val ) throws Exception
    {
        System.out.println("From method1, that throws an Exception object");
        if(val == 0)
            throw new Exception("From method method1");
        return 1;
    }
    /**
     * Method method2 throws an exception of type
     * RuntimeException, which is an un-checked exception.
     */
    public static int method2(int val ) //throws clause optional
    {
        System.out.println("From method2, that throws a RuntimeException object");
        if(val == 0)
            throw new RuntimeException("From method method2");
        return 1;
    }
    public static void main (String[] args)
    {
        //uncommenting the line below would cause a compile error
        //int val1 = method1(0);

        //However, the below will compile.
        int val2 = method2 (0);
        //The call above must be commented out to execute the try catch block below.
    }
}
//call to method1 requires a try-catch block
try
{
    int val3 = method1(0);
}
catch (Exception ex)
{
    System.out.println(ex);
}
}

Listing 12.5
Listing 12.5 shows two methods, method1 and method2. Inside method1, there is a statement:

throw new Exception("From method method1");

The above statement uses the Java reserved word throw followed by the call to the constructor of Java class java.lang.Exception. This results in returning an object of type Exception, with the string “From method method1” as being part of the thrown object. method2 also has a throw statement, which throws an object of type RuntimeException. Understand that during execution, method1 or method2, could only do one of the two things: either perform a normal return, which would be return an integer value of one, or return an exception object following the word throw.

Let us scrutinize the header of method1 below.

public static int method1( int val ) throws Exception //throws clause required

Since method1 throws an exception of type java.lang.Exception (which is a checked exception), its header must broadcast that with the clause throws ClassName after the argument list. Skipping the throws clause in the header of methods that throw a checked exception would be a compile error. Reverse is also true. If a method broadcasts a checked exception to be thrown, then such exception or its super type must be thrown. A method can throw more than one exception, in which case they are listed after the keyword throws separated by comma. For example a
method that throws three exceptions Exception1, Exception2, and Exception3 (all checked exceptions), would require a header similar to the one given below:

```java
[Access specifier] return_type methodName ([Argument List]) throws
Exception1, Exception2, Exception3, ………………..
```

Also clearly understand the difference between reserved words throw and throws. The word throw is used preceding the instantiation or name of the exception object to be thrown. On the other hand, the word throws is used as the first word to broadcast the exception(s) that a method would throw. The word throws is never used inside a method body and word throw is never used next to the method header after the argument list.

A compile error would occur, if caller method (main method in this case) tries to call a method that throws a checked exception with out placing such call in a try block followed by a catch block that can process the thrown exception. Therefore uncommenting the statement

```java
int val1 = method1 (0);
```

in main would be a compile error. However, the method2 throws an object of type RuntimeException. Call to methods that throw an object of type \(^4\) RuntimeException is exempted from both requirements – that is to broadcast the exception being thrown (in the method header), and to place the method call statement in a try block. Therefore the call below would compile, though the program execution would terminate right after this call is completed.

```java
int val2 = method2 (0);
```

Last eight lines of the program show the correct technique to call method1 where pair of try catch block is used to and call to method1 is placed in the try block.

Figure 12.7 shows the compiler and JVM messages and outputs for different calls to methods method1 and method2 in Listing 12.5.

---

\(^4\) Recall that the word type means that the object could be an instance of that particular class or any of its subclasses. For example an object of “type” RuntimeException would be either an object of RuntimeException class or of any of its subclasses.
FIG. 12.7A
When the call to method1 is un-commented in the first code line in main (listing 12.5), then compiler flags that call and gives the above compile error message (Figure 12.7A). What compiler is saying here is that either the main method (inside which the method1 is being called) must catch the exception thrown by the method1 (by using try catch blocks), or it must declare in its method header to throw the same object. This brings us to the point in Chapter eight, where we declared the header of main method as follows:

```java
public static void main(String [] args) throws IOException
```

We made the main method throw an exception of type IOException because we were calling the method read or readLine inside the main to get user inputted data from the keyboard. You would recall that these are the methods in BufferedReader class, which can be used to read a character or a line from the keyboard input. Table 12.1 gives the headers of these methods and the hierarchy of class IOException.

```java
public int read() throws IOException
public String readLine() throws IOException
```

<table>
<thead>
<tr>
<th>Java.lang.Object</th>
<th>java.lang.Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Throwable</td>
<td></td>
</tr>
</tbody>
</table>

Table 12.1

The above table shows that the methods read and readLine throw an exception of type IOException that is derived from the class Exception and is not of type RuntimeException (or of type java.lang.Error). Therefore IOException is a checked exception and the methods that throw an object of this type must either be called inside a try block or the caller to these methods itself must throw IOException or its super class object. Therefore using the clause throws IOException next to the header of main method gave us an exemption from placing calls to methods read and readLine inside a try block. Were we to remove the clause throws IOException from the header of main method, we would need to call the methods read and readLine using the syntax similar to the one we used to call the method1 in last eight line of Listing 12.5 Or Figure 12.4). We shall illustrate that shortly in a detailed example.
In Listing 12.5 if we comment out the first line (call to method1) but keep the call to method2 then output generated is shown by Figure 12.7B.

**FIG. 12.7B**
The call to method2 with out using the try block compiles and runs since method2 throws an unchecked exception. The first line of method2 prints a statement on the console. However, the program terminates since an exception is thrown and it is not handled in the way to recover from this error. This is the general characteristic of all Java programs. When a Throwable object is not caught (or is not catch-able) in ways that program can recover from such error, then the program terminates.

Finally if we only keep the last eight lines of code in Listing 12.5, where the call to method1 is placed inside a try block followed by the catch block, we see the output shown in Figure 12.7C.

**FIG. 12.7C**
The output first prints the statement from the method1, and then it executes the catch block, where the toString method for object ex (passed to catch block) is called and string returned is printed. The toString method of Exception class prints the qualified class name (java.lang.Exception) and appends the colon and the message passed to the constructor when the thrown object was instantiated. Notice that the part of last output line “From method method1” came from the string that was passed during the following constructor call in method1.

```java
throw new Exception("From method method1");
```

The formal parameter in the catch clause must match the “type” of actual exception object thrown from the code inside the try block. For example in Listing 12.5, writing the catch block as below would result in a compile error.
catch(RuntimeException ex)  
{  
    System.out.println(ex);  
}  

The reason for this compile error is that compiler can determine that  
RuntimeException is a subclass of Exception and it is in violation of laws of  
inheritance to pass a super class object, when a sub class object is expected.  
However a catch block such as below would not be a compile error as when a super  
class object is expected by a block or method, a sub class object can always be  
passed at the run time.  

catch(Throwable ex)  
{  
    System.out.println(ex);  
}

Table 12.2 gives some examples of un-checked and checked exceptions commonly  
encountered. A verbal description of some of them was given in sidebar 12.1.

<table>
<thead>
<tr>
<th>Exception throwing code</th>
<th>Exception class object thrown</th>
<th>Type of the Object thrown</th>
</tr>
</thead>
<tbody>
<tr>
<td>int val = Integer.parseInt(&quot;Hello&quot;);</td>
<td>NumberFormatException</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>int [] arr = new int [5]; arr[10] = 10;</td>
<td>ArrayIndexOutOfBoundsException</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>String Str = null; Str.length ();</td>
<td>NullPointerException</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>Vector V= new Vector (); V.firstElement();</td>
<td>NoSuchElementException</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>Object I = new Integer(5); Double Dbl = (Double)(I);</td>
<td>ClassCastException</td>
<td>RuntimeException</td>
</tr>
</tbody>
</table>
| BufferedReader BR = null; try  
{  
    BR = new BufferedReader (new FileReader("G:\text.txt"));  
}  
catch(Exception ex)  
{  
    System.out.println (ex);  
}  
| FileNotFoundException | Exception (checked exception) |

Assume that no drive with letter G exists.
1- All are subclasses of class RuntimeException, not requiring the placement of exception causing code in the try block.
2- Subclass of Exception but not the sub class of RuntimeException or of Error. Table A12.1.1, in appendix A, shows the complete inheritance hierarchy of all of the exception classes in column two of this table.

Table 12.2

Based on discussion so far, Java exception handling principles dictate that class designers and clients:

- are required to provide exception handlers for checked exceptions. The methods throwing checked exception must broadcast them in the method header.
- are not expected to provide any handlers for the exceptions of type java.lang.Error, since recovery from them is generally not possible.
- may provide exception handlers for exceptions of type java.lang.RuntimeException, if programmer from the design of program, can conclude that such handler would provide adequate program recovery. The broadcasting of thrown unchecked exceptions in the method header is optional. However, it is a good programming practice to do so.

Multiple catch clauses
Java allows programmer to place multiple catch clauses following a single try block. Reverse however, is never true. A catch block can only be preceded either by a single try block or by another catch block. Multiple catch blocks improve the granularity of exception handling and pinpoint the error location precisely, which in turns helps recovery from run time errors. Figure 12.7 shows the syntax of multiple catch blocks following a try block.
try
{
    //statements;
}
catch (exceptionType1 identifier1)
{
    // zero or multiple statements;
}
catch (exceptionType2 identifier2)
{
    // zero or multiple statements;
}
//More catch blocks as needed
//Fig.12.7

We design a class that would have code and methods to prompt user for the name of an input file and simply print its contents to the console. All input and output, involving user would be standard I/O type. Let us define the steps in our overall strategy.

1. Set a boolean variable done to false.
2. Inside a do while loop do the followings:
3. Prompt user for input file name by displaying a message on console.
4. Accept user input for file name from the keyboard.
5. Create a File class object using the string entered in step four.
6. Create a FileReader object from the File object created in step five.
7. Test if file has the read permission – if not then display the related message and set done (step one) to false.
8. Ascertain that file is not hidden. If hidden then set done to false.
9. Ascertain that file has some data in it. If file is empty then set done to false.
10. Prompt user to provide the size of input buffer.
11. Accept the user-inputted string.
12. Parse the String entered in step 11 into an integer.
13. Create a BufferedReader object by calling the constructor that would take the FileReader object created in step six and buffer size in step 12.
14. Pass the object created in step 13 to method readAndPrintFile
15. Set done to true
16. Handle all exceptions.
17. Post-test the loop condition.

The step 16 requires that we take an inventory of steps one to 14 and tabulate as to which steps can throw an exception and what kinds would those be? Table 12.3 shows the step number, which method or constructor would be used in that step, and what exception they can throw and their type.

<table>
<thead>
<tr>
<th>Step Number</th>
<th>Class and method/constructor used</th>
<th>Name of the Exception thrown</th>
<th>Type of Exception (Checked or unchecked)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>BufferedReader.readLine</td>
<td>IOException</td>
<td>checked</td>
</tr>
<tr>
<td>5</td>
<td>File/Constructor that takes file name string as argument</td>
<td>NullPointerException</td>
<td>unchecked</td>
</tr>
<tr>
<td>6</td>
<td>FileReader/Constructor that takes File object as argument</td>
<td>FileNotFoundException</td>
<td>checked</td>
</tr>
<tr>
<td>7</td>
<td>File/canRead</td>
<td>SecurityException</td>
<td>unchecked</td>
</tr>
<tr>
<td>8</td>
<td>File/isHidden</td>
<td>SecurityException</td>
<td>unchecked</td>
</tr>
<tr>
<td>9</td>
<td>File/length</td>
<td>SecurityException</td>
<td>unchecked</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>BufferedReader.readLine</td>
<td>IOException</td>
<td>checked</td>
</tr>
<tr>
<td>12</td>
<td>Integer/parseInt</td>
<td>NumberFormatException</td>
<td>unchecked</td>
</tr>
<tr>
<td>13</td>
<td>BufferedReader/constructor that takes FileReader and integer as buffer size as arguments</td>
<td>IllegalArgumentException</td>
<td>unchecked</td>
</tr>
</tbody>
</table>

Table 12.3

All the exceptions listed in the column three of Table 12.3 are already defined by Java for us. They are either checked exceptions or unchecked exceptions. In order to place the catch blocks so that all exceptions are “reachable”, we must look at the inheritance hierarchy of all of them. Table 12.4 provides this information.
<table>
<thead>
<tr>
<th>Exception Names</th>
<th>Inheritance Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IllegalArgumentException</code>, and</td>
<td><code>java.lang.Object</code></td>
</tr>
<tr>
<td><code>NumberFormatException</code></td>
<td><code>java.lang.Throwable</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Exception</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.RuntimeException</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.IllegalArgumentException</code></td>
</tr>
<tr>
<td></td>
<td><code>java.langNumberFormatException</code></td>
</tr>
<tr>
<td><code>NullPointerException</code></td>
<td><code>java.lang.Object</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Throwable</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Exception</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.RuntimeException</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.NullPointerException</code></td>
</tr>
<tr>
<td><code>SecurityException</code></td>
<td><code>java.lang.Object</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Throwable</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Exception</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.RuntimeException</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.SecurityException</code></td>
</tr>
<tr>
<td><code>IOException</code> and <code>FileNotFoundException</code></td>
<td><code>java.lang.Object</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Throwable</code></td>
</tr>
<tr>
<td></td>
<td><code>java.lang.Exception</code></td>
</tr>
<tr>
<td></td>
<td><code>java.io.IOException</code></td>
</tr>
<tr>
<td></td>
<td><code>java.io.FileNotFoundException</code></td>
</tr>
</tbody>
</table>

**Table 12.4**

**Reachable vs. Unreachable Exceptions**

When arranging the multiple catch blocks to handle exceptions, Java requires that a subclass exception be caught first, before a super class exception is caught. For example based on the hierarchy of exception classes in Table 12.4 it is required that for the checked exceptions the catch blocks be arranged in the following order (Figure 12.8A).
try
{
    //code
}
catch(FileNotFoundException ex)//sub class
{
    //code
}
catch(IOException ex)//super class
{
    //code
}//Fig. 12.8A

Notice that the catch blocks for exceptions are arranged in the order (from top to bottom) that is exact reverse of their inheritance hierarchy (See last row and last column of Table 12.4). If that were not the case and we were to change the order to the one given in the Figure 12.8B, then last catch block becomes “unreachable” code as a super class catch block (IOException in this case) would also catch the subclass exception (FileNotFoundException).

try
{
    //code
}
catch(IOException ex)
{
    //code
}
catch(FileNotFoundException ex)
{
    //code
}//Fig. 12.8B

Java would issue a compile error when such “unreachable” code (Figure 12.8B) is encountered. Similarly Figure 12.8C shows as how to arrange the handlers for unchecked exception classes.
try
{
   //code
}
catch(NumberFormatException ex) //sub class
{
   //code
}
catch(IllegalArgumentException ex) //super class
{
   //code
}
//Fig. 12.8C

This is so because the IllegalArgumentException is the super class of
NumberFormatException. Therefore, enforcing the “reverse inheritance hierarchy”
rule for the placement of exception handlers, the handler for
NumberFormatException precedes its super class IllegalArgumentException. Notice
that the handlers for the remaining Exceptions, NullPointerException, and
SecurityException have only one requirement, that they be placed before the super
class of all exceptions – Exception (if used). Their placement with respect to other
unrelated classes is unimportant. Also since the checked and unchecked exceptions
are unrelated with respect to their inheritance hierarchy, their relative order of
placement is also unimportant. We follow this arbitrary order, in which we place
the unchecked exception handlers first and then place the checked exception
handlers afterwards. Figure 12.8D shows the overall placement of all exception
handlers in our forthcoming listing.

try{
   //code
   catch(NumberFormatException NFEx) //Sub class of IllegalArgumentException
   {
      //code
   }
   catch(IllegalArgumentException IllegalEx) //unchecked exception
   {
      //code
   }
   catch(NullPointerException NPex) //unchecked exception
   {
      //code
   }
   catch(SecurityException SC) //unchecked exception
   {
      //code
   }
}
We place the last catch clause to catch any remaining exception that our code may throw. A handler with Exception as its formal argument would catch any exception that were not caught by other handlers preceding it. Listing 12.6 shows the source code for the class ReadDataFile, which codes the algorithm described earlier and uses the exception handling strategy developed in Figure 12.8D.

```java
import java.io.*;

public class ReadDataFile {
    private ReadDataFile(){
        public static void main (String[] args) {
            BufferedReader KB = new BufferedReader(new InputStreamReader(System.in));
            String Input = new String();
            boolean done = false;
            do {
                try {
                    System.err.println("Please type the name of data input file and then press enter key.");
                    Input = KB.readLine();//readLine can throw IOException
                    File FL = new File(Input);//File class constructor can throw NullPointerException
                    FileReader FR = new FileReader(FL);//Can throw FileNotFoundException
                    //check for length, hidden status, and read permission.
                    if(FL.canRead())
                    {
                        if(!FL.isHidden())
                        {
```
if(FL.length() != 0)
{
    System.err.println("Please specify the size of read buffer");
    
    Input = KB.readLine();
    
    int buf = Integer.parseInt(Input);
    
    BufferedReader FReader = new BufferedReader(FR, buf);
    
    readAndPrintFile(FReader);
    
    done = true;
}
else
{
    System.err.println("The file has no data in it.");
    done = false;
}
else
{
    System.err.println("The file is hidden.");
    done = false;
}

else
{
    System.err.println("The file does not have read permission");
    done = false;

} //end of try block

catch(NumberFormatException NFEx)//unchecked exception
{
    NFEx.printStackTrace(System.err);
    System.err.println("Non-numeric buffer size has been entered.");
}

catch(IllegalArgumentException IllegalEx)//unchecked exception
{
    IllegalEx.printStackTrace(System.err);
    System.err.println("Illegal buffer size has been entered.");
    done = false;
}

catch(NullPointerException NPex)//unchecked exception
{
    NPex.printStackTrace(System.err);
    System.err.println("Null file name or null string has been entered.");
    done = false;
catch(SecurityException SC)//unchecked exception
00077    {
00078        SC.printStackTrace(System.err);
00079        done = false;
00080    }
catch(FileNotFoundException FX)//checked exception - subclass of IOException
00082    {
00083        FX.printStackTrace(System.err);
00084        done = false;
00085    }
catch(IOException ex)//checked exception - subclass of Exception
00086    {
00087        ex.printStackTrace(System.err);
00088        System.err.println("Input reading error.");
00089        done = false;
00090    }
00091    }
catch(Exception ex)
00092    {
00093        System.err.println("Unknown error/exception has occurred.");
00094        done = false;
00095    }
00096          while(!done);
00097                  }
00098            }
00099               public static void readAndPrintFile(BufferedReader BR)
00100               {
00101                    String Data = null;
00102                    int count = 0;
00103                    try
00104                    {
00105                        System.err.println("Printing your file to screen now.");
00106                        System.err.println("Input reading error.");
00107                        System.err.println("Unknown error/exception has occurred.");
00108                        System.err.println("Press enter to continue printing.");
00109                        }
00110                    }
00111                    }
00112                    }
00113                    }
00114                    }
00115                    }
00116                    }
00117                    }
00118                    }
00119                    }
00120                    }
00121                    System.err.println("Keyboard input error.");
In Listing 12.6 we use the object System.err to print message to console, as in some cases where file processing is involved, the System.out may fail to print the message on the console. On the other hand the System.err.println is always able to print message to the console. For example if a code snippet below (in the main method of Class1) is run from the command line with the following command:

```
System.out.println("Can be directed to a file.");
System.err.println("Cannot be directed to a file.");
>java Class1 > file
```

then the message “Cannot be directed to a file.” Would print to console, whereas the first message would be redirected to the file. Therefore to ascertain that user attention is drawn to messages from exception handlers, then it is best to use System.err.println/print methods.

The algorithm for Listing 12.6 discussed earlier is implemented as follows. User input for the input file name is taken from the standard input (L12.6#22). Since readLine method can throw an IOException, a catch block is included (L12.6#86-91). The user inputted string is passed as an argument to the File class constructor to create a File object FL (L12.6#23). This File class constructor can throw a NullPointerException, for which a handler is placed (L12.6#70-75). The File class constructor throws a NullPointerException, when a null string is passed to it as argument. Then the File object FL is passed to the constructor of FileReader class to create a FileReader object FR (L12.6#24). This constructor can throw a FileNotFoundException, and we add a handler for that as well (L12.6#81-85).

We need to ascertain further that:
- File has read permission
- File is not hidden
- And file is not empty.
We use the File Object FL and File class methods canRead, isHidden, and length for these authentications. However, depending upon whether the file has read permission or not, these three methods can throw an object of class
SecurityException, for which we add a handler (L12.6#76-80). A system of enclosing if/else blocks filters down to the last block (L12.6#31-39), by when the following facts have been established:

- There is no error reading from keyboard.
- A non-null file name has been provided and file exists.
- File has read permission, is not hidden and has data in it.

The user is prompted to provide an integer value for the size of the read buffer (L12.6#33). The user input is parsed and stored into int variable buf, and then FileReader object FR created earlier and buffer size buf are passed to the constructor of BufferedReader class to create an object Freader (L12.6#36). The integer parsing process can throw a NumberFormatException, and BufferedReader constructor used can throw an IllegalArgumentException. Therefore the handlers for both of them are added (L12.6#59-69). If no exceptions are thrown up to code line L12.6#36, then the object Freader is passed to method readAndPrintFile to read the file and print it to console.

**Code inside exception handlers**

Inside each exception handler we add the code of the following type:

```
catch (ExceptionType Ex)
{
    Ex.printStackTrace (System.err);
    System.err.println ("Message");
    done = false;
}
```

All exception classes inherit the overloaded method printStackTrace, and Table 12.5 gives its various versions along with the method getMessage.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void printStackTrace()</td>
<td>Prints this Throwable and its backtrace to the standard error stream. This method prints a stack trace for this Throwable object on the error output stream that is the value of the field System.err. The first line of output contains the result of the toString() method for this object. Remaining lines represent data previously recorded by the method fillInStackTrace().</td>
</tr>
<tr>
<td>public void printStackTrace(PrintStream s)</td>
<td>Prints this Throwable and its backtrace to the specified print stream. Parameters: s - PrintStream to use for output</td>
</tr>
<tr>
<td>public void printStackTrace(PrintWriter s)</td>
<td>Prints this Throwable and its backtrace to the specified print writer. Parameters: s - PrintWriter to use for output</td>
</tr>
<tr>
<td>public String getMessage()</td>
<td>Returns the detail message string of this Throwable. Returns: the detail message string of this Throwable instance (which may be null).</td>
</tr>
</tbody>
</table>

**Table 12.5**
The method printStackTrace first gives message, which is the string returned by its toString method. We saw example of that in the catch block in Listing 12.5, where the toString method prints the full name of the Exception class followed by a colon and the message passed to the constructor (or a default message) of the thrown object. The fillInStackTrace method records the stack trace information in a Throwable object, which is also printed by the method printStackTrace. The method getMessage can also be used. In our opinion, however, it is no more useful than just calling the toString method of exception object and printing that string directly. That using method printStackTrace is a better choice would become clear shortly, when we discuss the output from Listing 12.6.

After printing the message from printStackTrace method, we add any other optional/clarifying message in the catch block. Finally we set the loop control boolean variable to false, as the normal loop processing did not take place.

**Method readAndPrintFile and nested try catch blocks**

The purpose of the method readAndPrintFile is to get a valid BufferedReader reference to a data file (with data in it) and print its contents to console (L12.6#101-132). The method uses a EOF controlled loop to read the input file line by line (L12.6#109-124). Since readLine method can throw an IOException the loop is placed inside a try block and a handler is added (L12.6#126-129). We add additional code to ascertain that user has a chance to pause the program after seven lines. This requires creation of a “nested” try catch block (L12.6#114-122). Notice that there is no limitation in nesting a try-catch block inside another try block as needed. The requirement however is that handlers for the nested try block, also must lie inside the enclosing try block. The nested try block prints a message for the user and prompts them to press enter key to continue printing the input file to console. The user input is consumed by the method call System.in.read. Since call to read method throws IOException, we add the handler for it as well (L12.6#119-122). If execution of the method ends normally then input file is printed to the console.

**Output from Listing 12.6**

We test Listing 12.6 by creating as many error conditions as we can. However, all error conditions are not tested. Figure 12.9 shows the results of our test.
null

Java Exception Handling

Please type the name of data input file and then press enter key.

at java.io.File.<init>(Unknown Source)
at ReadDataFile.main(ReadDataFile.java:23)

File name or null string has been entered

Please type the name of data input file and then press enter key.

g:\xyz

java.io.FileNotFoundException: G:\xyz (The system cannot find the path specified)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(Unknown Source)
at java.io.FileReader.<init>(Unknown Source)
at ReadDataFile.main(ReadDataFile.java:24)

Please type the name of data input file and then press enter key.

Please specify the size of read buffer

java.lang.IllegalArgumentException: Buffer size < 0
at java.io.BufferedReader.<init>(Unknown Source)
at ReadDataFile.main(ReadDataFile.java:36)

Illegal buffer size has been entered.

Please type the name of data input file and then press enter key.

Please specify the size of read buffer

java.lang.NumberFormatException: For input string: "scccccccccc"
at java.lang.NumberFormatException.forInputString(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(String, int)
at ReadDataFile.main(ReadDataFile.java:35)

Non-numeric buffer size has been entered.

Please type the name of data input file and then press enter key.

Please specify the size of read buffer

java.lang.IllegalArgumentException: Buffer size < 0
at java.io.BufferedReader.<init>(Unknown Source)
at ReadDataFile.main(ReadDataFile.java:36)

Illegal buffer size has been entered.

Please type the name of data input file and then press enter key.

Please specify the size of read buffer

Printing your file to screen now.

"""
Making Hay Out Of Straw Men

By Dana Milbank

Tuesday, June 1, 2004; Page A21

"""

"...would have meant more families would have been out of work."

Now who could argue with that?

"""

Done printing.

FIG. 12.9
The program prompts user to input a file name, and if user presses control Z keys on DOS (or control D in Unix), then a null string is entered. The File class constructor (L12.6#23) throws a NullPointerException and the corresponding catch block prints the stack trace and displays the message from that block (L12.6#70-75). The following lines printed by printStackTrace method elucidate the source of error.

```
at java.io.File.<init>(Unknown Source)
at ReadDataFile.main(ReadDataFile.java:23)
```

The first line says that the constructor of class java.io.File threw an exception. (Java indicates constructors by expression <init>). Second line gives the location of the line in the class and method, where the exception was thrown.

When user provides a non-existent file name then the FileNotFoundException is thrown (bubble #2: Figure 12.9). The program prints back the file name provided and prints the message that file is not found. The stack trace indicates that exception was thrown by the call to “native” open method by the object of FileInputStream class, and propagated thru the constructors of classes FileInputStream and FileReader classes.

After providing a valid file name the user is prompted to enter the size for the read buffer. A NumberFormatException is thrown when a non-numeric string is entered (bubble #3 Figure 12.9). And an IllegalArgumentException is thrown by the constructor of class BufferedReader when the buffer size entered is zero or negative (bubble #4 Figure 12.9). Program prints out the file provided when a valid buffer size is entered. Notice that in this listing we were not able to test throwing of SecurityException and IOException as special device and file permission conditions need to be created to throw those exceptions.

**finally clause**

There are situations when the exceptions are thrown, user is informed of the error condition, and if program resumption is not possible then in the way of recovery the program tries to reclaim resources as soon as possible. This reclamation of resources may be connected to for example:

- Closing output file, so that the data written so far is saved safely.
- Closing the input file safely.
- Closing connections to databases.
- Close a socket or network connection.
- Disposing the graphical components using the program memory..
  etc.

---

5 A native Java method calls an existing method that may have been written another programming language, residing on the operating system or platform on which the program is being run.
Java allows one to put a clause labeled **finally** after the try catch blocks. The characteristic of **finally** block is that as long as `System.exit(0)` is not called from inside of any of the preceding try and catch blocks then the code inside the **finally** block is always executed! Figure 12.10 shows the syntax of using **finally** block.

```java
try {
    //statements;
} 
catch (exceptionType1 identifier1) {
    // zero or multiple statements;
} 
catch (exceptionType2 identifier2) {
    // zero or multiple statements;
} 
//more catch blocks if needed
finally {
    //code
}
```

**FIG. 12.10**

Mostly the **finally** clause is used after a set of try catch blocks. However Java allows finally block to be placed right after a try block – missing a catch block altogether. Therefore the following will compile and could be useful in situations, where catching an exception does not help the situation and finally block executes the rescue code (Figure 12.11).

```java
try {
    //statements;
} 
finally {
    //code
}
```

**FIG. 12.11**

You have perhaps noticed by now that we did not actually close the input file in the Listing 12.6. In order to safely close the input file in Listing 12.6 we add the
following piece of code using a finally clause between the lines 129 and 130 (Figure 12.12).

```java
finally {
    try {
        BR.close();
    } catch (IOException Ex) {
        Ex.printStackTrace(System.err);
        System.err.println("File closing error");
    }
}
```

Notice that in this case the try catch block inside the finally block is needed as the call to method close would throw an IOException. A finally block can also contain try catch and finally blocks. Use of finally block is very useful when the program is writing to an output file and file need to be closed safely, so that user data are saved in spite of program crash. Similar situation exists when the program is writing to the database and connection to database is needed to be closed lest the database gets corrupted upon program crash.

**Coping with un-checked exceptions**

Generally checked exceptions have a well-defined hierarchy. Thus placing the catch blocks for them in order reverse of their inheritance hierarchy may locate their origin rather well. Un-checked exceptions do not always have this advantage. For example in our Listing 12.6 a NullPointerException can be thrown when either File class constructor gets a null string (L12.6#35), or a null string is entered for the buffer size (L12.6#35). Thus in the handler for NullPointerException, we have to be satisfied with the message that ""Null file name or null string has been entered." Fortunately, the call to printStackTrace method would pinpoint the location as to where exception was thrown; still it is an inconvenient feature to use. For an un-checked exception that can be thrown at various locations, each such location needs to be isolated in a separate try block. Such “error-trapping” can be expensive in terms of program overhead, however in mission critical software applications, it may be a necessity. For example, very often in databases all fields of stored objects are not required to be filled\(^6\). Let us assume that our database keeps records of student names, and their GPA. We may end up creating two vectors from the queries made to our database, that are followings.

---

\(^6\) In relational databases, it is desirable to have primary key, but it is not required.
Vector Names = new Vector();
//Fill Names Vector with student names for a certain class
Vector GPA = new Vector();
//Fill GPA Vector with the GPA of students.

If the two vectors Names and GPA are used independently (as if they have nothing to do with each other) then there is no problem. For example, the Names vector can print student names and GPA vector can print a list of GPA and find average GPA, etc. The problem comes when we try operations like below:

```java
int ind = 5;
String Student_Name = (String)Names.elementAt(ind);
double current_gpa = Double.parseDouble(GPA.elementAt(ind).toString());
System.out.println("The GPA of student " + Student_Name + " is = " + current_gpa);
```

The code above tries to print the GPA of student at the location index = 5, in the Names vector. Even after assuming that we have confirmed by calling the size method to make sure that ind = 5, is a valid element in vector Names, we are naively assuming that database stores GPA for all students! For example students who are attending college for the first time may not have a GPA, and database may not require a number to be stored in that field. As a result the elementAt method called second time may throw ArrayIndexOutOfBoundsException. Worse yet, knowing that Vector stores Objects, there is no certainty that element at location index = 5 is indeed a Double type element. This can happen if somehow the vector GPA became corrupted in the process of adding elements to it. Figure 12.13 shows, as how one would separate this code into neat pieces for exception handling and error trapping.

```java
//Assume that Vector Names and GPA are already populated with student names //and GPA respectively. Code tasks to print the GPA of student at index = 5 in //Names Vector. If all students have GPA then size of both vector must be same.
if(Names.size() != GPA.size())
    throw new RuntimeException("Vectors storing student names and GPA “ + “have different sizes.");

String Student_Name = new String();
int ind = 5;
```
try
{
    Student_Name = (String)Names.elementAt (ind); //Line #1
    Double Temp = GPA.elementAt (ind); //Line #2
    try
    {
        double current_gpa = Double.parseDouble(Temp); //Line#3
        System.out.println("The GPA of student " + Student_Name
         + " is = " + current_gpa);
    }
    catch(NumberFormatException Ex)
    {
        Ex.printStackTrace (System.err);
    }
}
catch(ArrayIndexOutOfBoundsException Ex)
{
    Ex.printStackTrace (System.err);
}

In the code shown in Figure 12.13, the error trapping strategy uses combination of Vector class method calls, throwing un-checked exception, and use of nested try catch blocks to handle un-checked exceptions. The caller of this code, if they wish, could use a try catch block also to handle the RuntimeException thrown, when Vector sizes are unequal. Since we did not ascertain that ind is positive and less than size, we must use the try catch block when we get the element at the index ind. If an ArrayIndexOutOfBoundsException would be thrown at line #1 then program would abort and execute the catch block for it. If Line#1 does not throw an exception then Line #2 would not either as we have ascertained that size of two vectors are same. However, to trap the parsing error (in case the GPA vector has non Double element at location ind), we call the parseDouble method inside another try block, which is followed by a catch block to handle NumberFormatException. If Line#3 throws the NumberFormatException, then the program is aborted there and catch block after it is executed. However, if no exception is thrown on Line #3, then rest of the code executed error free.

This somewhat contrived example illustrates the strategy for coping from un-checked exceptions and their use for error trapping. In actual application a better technique would be to create a Student class, where Name and gpa fields are populated first, and then use a Vector of Student objects to print record of each student, rather than trying to use two parallel vectors the way we used here.
Designing user defined exception classes

The mechanics of implementing user defined exception classes is quite simple. In order to design a new checked exception user would inherit the new class from the java.lang.Exception class or from any of its non-final unchecked exception subclasses. For new unchecked exception classes one would inherit the new class from java.lang.RuntimeException class. We show the constructors for Exception class in Table 12.6.

<table>
<thead>
<tr>
<th>Constructor Summary for java.lang.Exception class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exception ()</strong></td>
</tr>
<tr>
<td>Constructs a new exception with null as its detail message.</td>
</tr>
<tr>
<td><strong>Exception (String message)</strong></td>
</tr>
<tr>
<td>Constructs a new exception with the specified detail message.</td>
</tr>
<tr>
<td><strong>Exception (String message, Throwable cause)</strong></td>
</tr>
<tr>
<td>Constructs a new exception with the specified detail message and cause.</td>
</tr>
<tr>
<td><strong>Exception (Throwable cause)</strong></td>
</tr>
<tr>
<td>Constructs a new exception with the specified cause and a detail message of (cause==null ? null : cause.toString()) (which typically contains the class and detail message of cause).</td>
</tr>
</tbody>
</table>

Table 12.6

The constructors for class RuntimeException are exactly same type as they take the same arguments. Let us assume that we need to design an exception class to facilitate array processing, over and above what Java already provides for us. For example when an array reference is passed to a method as an argument – two things can happen:

- The array reference may be null, in which case the method can throw a NullPointerException as there is not much processing that method can do with such null reference.
- Array has a zero size. In this case also no processing is possible. Method can check the size of array for non-zero value before proceeding further. However, no standard exception is available that can be thrown if method receives a zero size array. If method must return a value, then it can simulate array access using the following code and since the array has a zero length, it would throw a ArrayIndexOutOfBoundsException.

```java
int [] arr = new int[0];
int val = arr[0];
```

Will throw ArrayIndexOutOfBoundsException since array size is zero and there are no elements in it.

Client of the method can catch that exception and process it further to know that a zero array size was passed to the method. The problem with this is that ArrayIndexOutOfBoundsException is not very informative of the situation that a zero size array was passed to the method! Therefore what is desirable is to define a
new Exception class that would inform user of the fact that a zero size array was passed to the method. Let us call this exception class as ZeroArraySizeException. And we design it to be a sub class of RuntimeException – which means that it is of unchecked type. Listing 12.7A shows this incredibly easy to design exception class.

```java
public class ZeroArraySizeException extends RuntimeException
{
    public ZeroArraySizeException()
    {
        super( );
    }

    public ZeroArraySizeException(String Message)
    {
        super(Message);
    }

    public ZeroArraySizeException(String Message, Throwable Cause)
    {
        super(Message, Cause);
    }

    public ZeroArraySizeException( Throwable Cause)
    {
        super(Cause);
    }
}
```

//Listing 12.7A

All one has to do is to implement the four constructors and call super class constructor inside each! Additional method and fields may be added if further customization is needed. Let us assume that we are writing a method findMin, which takes an integer array as argument and returns lowest value stored in it. Listing 12.7B shows a test class Class1 that contains this method, which in turn uses the exception class in Listing 12.7A.

```java
public class Class1
{
    public static void main (String[] args)
    {
        int [] arr = new int[(int)(10*Math.random( ))];
        boolean OK = true;

        if((int)(10*Math.random())%5 == 0)
        {
            arr=null;
            OK = false;
        }
    }
}
```

Java Exception Handling
if(OK)
{
    for(int ind = 0; ind<arr.length; ++ind)
        arr[ind] = (int)(100*Math.random());

    //print the array.
    System.out.println("Printing your array.");
    for(int ind = 0; ind<arr.length; ++ind)
        System.out.print(arr[ind] + " ");
    System.out.println();
}

    //get and print the smallest value
    int min = 0;
    try
    {
        try
            {
        min = findMin(arr);
        System.out.println("The smallest element in the array printed above is "+ min);
            }
        catch(NullPointerException Ex)
            {
                Ex.printStackTrace(System.err);
            }
        }
        catch(ZeroArraySizeException Ex)
            {
                Ex.printStackTrace(System.err);
            }
    }

public static int findMin(int [] arr)
{
    if(arr == null)
        throw new NullPointerException("findMin:The array passed to method is a null "+ "reference.");
    if(arr.length == 0)
        throw new ZeroArraySizeException("findMin: The size of the array passed to "+ "the method is zero.");

    int min = arr[0];
```java
for(int ind=0; ind<arr.length; ++ind)
    if(arr[ind]<min)
        min = arr[ind];
return min;
```

The method findMin (Listing12.7B#49-66) takes an integer as argument. The method returns the smallest number in the array. However, the array reference passed to the method may be null. Therefore the method examines array arr for null value. If arr is null then the method throws a NullPointerException (Listing12.7B#51-53). Then the method checks if arr has a zero length, and if true, it throws a ZeroArraySizeException object (Listing12.7B#55-57) that we designed in Listing 12.7A. Method uses the standard algorithm to find the smallest value and returns it provided none of the exceptions are thrown.

In order to test the method findMin, we create an array of random length in main method (Listing12.7B#6). During some invocations, method Math.random would supply a zero number, and zero length arrays would be created. We also set the array arr to a null value randomly (Listing12.7B#9-13), in which case we set the boolean flag OK to false. If OK is not set to false, then we have a non-null array, which we fill with random numbers and print it soon after (Listing12.7B#15-25).

Call to method findMin could throw either ZeroArraySizeException or NullPointerException. Since these two exceptions are unrelated, the order in which they are thrown is unimportant. However, since in method findMin the NullPointerException is thrown first, we handle it first in a nested try catch block (Listing12.7B#31-40). (Understand that client may not have the information as to which exception is thrown first by the method findMin). Then outer try catch block handles ZeroArraySizeException. Figure 12.4 shows the results from several run of the main method in Listing 12.7B.
Printing your array.
50  90  70  89  5  97  41
The smallest element in the array printed above is 5

Printing your array.
30  80  48  84  64  15  99  29  35
The smallest element in the array printed above is 15

Printing your array.
35  1  73
The smallest element in the array printed above is 1

ZeroArraySizeException: findMin: The size of the array passed to the method is zero.
    at Class1.findMin(Class1.java:56)
    at Class1.main(Class1.java:33)

java.lang.NullPointerException: findMin:The array passed to method is a null reference.
    at Class1.findMin(Class1.java:52)
    at Class1.main(Class1.java:33)

//FIG. 12.14
When the null or zero size arrays is not passed to the method findMin, it returns a normal value, which is the lowest value in the array. Otherwise it returns exceptional values thru either thrown NullPointerException or ZeroArraySizeException. The method such as findMin is an example of total method (function) that we discussed in first few pages of this chapter as it has a well defined behavior for all inputs provided to it and has no pre-condition for its use. Most methods used by client can be made into total methods by judicious use of exceptions. There is an alternate code for method findMin shown by Figure 12.15.

class Class1
{
    public static int findMin(int[] arr)
    {
        int min = 0;
        try
        {
            min = arr[0];
        }
        catch(ArrayIndexOutOfBoundsException Ex)
        {
            throw new ZeroArraySizeException("findMin: The size of the array passed to" +"\nthe method is zero.",Ex);
        }
    }
}

print your array
for(int ind=0; ind<arr.length; ++ind)
    if(arr[ind]<min)
        min = arr[ind];

    return min;

} //FIG. 12.15

In alternate code we do not explicitly check for arr being null or having zero size. We simply attempt to set min equal to arr [0]. If array size is zero then ArrayIndexOutOfBoundsException would be thrown. In catch block we throw the new exception ZeroArraySizeException, so that user gets the precise information about the nature of exception thrown. Functioning of both methods is similar. The first version is logically easier to understand, whereas the second version is more compact. In second version both, NullPointerException and ArrayIndexOutOfBoundsException are thrown implicitly by JVM. First one is thrown when a null arr is passed and script operator [ ] are applied to the null reference. Second one is thrown when program tries to access the first element in a zero length array.

Checked versus unchecked exceptions
A harder software engineering choice is to decide whether to design the user defined exception class to be an unchecked exception or checked one. There is one school of thought, which includes author Bruce Eckel, who has come out totally in favor of unchecked exceptions. In fact new Microsoft language C# does not have checked exceptions. Other experts, such as Barbara Liskov7 from MIT, prefer a judicious mixture of both.

Liskov view point is to use unchecked exceptions if the client can easily enforce the pre-conditions dictated by the method being called. This may happen if confirmation techniques are available for easy enforcements of pre-conditions. For example in our Listing 12.7B, the client (main method) can easily check if the array passed to the method findMin is a null pointer or a zero size array (in fact the first version of findMin just does that). Then exception condition would arise only occasionally thru programming slips. For example in Java it is quite easy to avoid ArrayIndexOutOfBoundsException as arrays carry their length.

There are situations, however, when techniques to confirm whether the arguments used for the method would meet pre-condition or not may not exist or may be as expensive or more expensive to use than using a compile time exceptions. In such situations use of checked exceptions would be cost effective. For example let us assume that a piece of software searches for a key in an array using a method, whose header is given below.

public int search(int [ ] arr, int key)

7 Program Development in Java by Barbara Liskov, John Guttag Addison Wesley 2001
Method search is supposed to return the index of the array element where key is found scanning the array in the ascending order, otherwise search returns a sentinel value for array index = -1. If numerous calls were made to search then each time checking the return value for its sentinel state before it can be employed in further data processing would also be very expensive. In such situations using checked exceptions becomes cost effective in terms of processing time.

Listing 12.8 shows the design of checked exception class ElementNotFoundException, the test class Class1. Later includes the method search and its application.

```java
public class ElementNotFoundException extends Exception {
    public ElementNotFoundException() {
        super();
    }
    public ElementNotFoundException(String Message) {
        super(Message);
    }
    public ElementNotFoundException(String Message, Throwable Cause) {
        super(Message, Cause);
    }
    public ElementNotFoundException(Throwable Cause) {
        super(Cause);
    }
}

class Class1 {
    public static void main (String[] args) {
        int [] arr = new int[100];
        for(int ind = 0; ind<arr.length; ++ind)
            arr[ind] = (int)(100*Math.random());
        //print the array.
        System.out.println("Printing your array.");
        for(int ind = 0; ind<arr.length; ++ind)
            System.out.print(arr[ind] + " ");
    }
}
```
if(ind%10 == 0)
    System.out.println();
System.out.println();
//Just check for 10 randomly generated keys
for(int index = 0; index<10; ++index)
{
    int key = 0;
    try
    {
        key = (int)(100*Math.random());
        int val = search(arr,key);
        System.out.println("The value " + key + 
" is found in the array at location " + val);
    }
    catch(ElementNotFoundException Ex)
    {
        Ex.printStackTrace(System.err);
    }
}

public static int search(int [] arr, int key)throws ElementNotFoundException
{
    //prime read to throw implicit NullPointerException and
    //ArrayIndexOutOfBoundsException, in case arr is a null
    //reference or zero length array.
    int value = 0;
    try
    {
        value = arr[0];
    }
    catch(ArrayIndexOutOfBoundsException Ex)
    {
        throw new ZeroArraySizeException( 
"search: The size of the array passed to" 
+" the method is zero." ,Ex);
    }

    boolean found = false;
    for(int ind = 0; ind<arr.length & !found; ++ind)
    {
        if(arr[ind] == key)
value = ind;
found = true;
}

if(!found)
    throw new ElementNotFoundException("The element " + key + " was not found in the array.");

return value;

//Listing 12.8

Figure 12.16 shows the output from the main method in Listing 12.8 (from Class1).

Printing your array.
9 6 86 6 0 66 82 83 55 61
78 96 2 28 62 9 51 60 66 82
31 42 5 71 80 61 30 53 22 28
85 37 19 2 33 36 26 97 46 80
38 84 63 5 74 61 55 30 7 70
49 21 91 35 82 93 47 73 18 49
91 18 56 48 83 49 26 95 62 1
86 23 86 19 90 62 2 13 49 36
29 64 21 26 66 73 82 13 80 90
6 74 57 63 25 56 2 42 16

ElementNotFoundException: The element 52 was not found in the array.
    at Class1.search(Class1.java:68)
    at Class1.main(Class1.java:27)

The value 31 is found in the array at location 21
The value 28 is found in the array at location 14

ElementNotFoundException: The element 11 was not found in the array.
    at Class1.search(Class1.java:68)
    at Class1.main(Class1.java:27)

ElementNotFoundException: The element 4 was not found in the array.
    at Class1.search(Class1.java:68)
    at Class1.main(Class1.java:27)

The value 55 is found in the array at location 9
The value 25 is found in the array at location 95
The value 22 is found in the array at location 29
The value 26 is found in the array at location 37
The value 35 is found in the array at location 54

FIG. 12.16
It may be daunting for traditional programmers to see that we are actually using the exception throwing mechanism to print a legitimate message that certain key was not found in the array (Figure 12.16). While this is unconventional, when using
checked exceptions, this mechanism is sound and effective. Since the technology used in the method search in Listing 12.8 never returns an out of bound array index, such index can be used further for any automatic data processing, since a check for its non-sentinel state is not required.

**Re-throwing or reflecting exceptions**
If a method gets a thrown object and it does not have a handler for it, then it would re-throw that object to its caller. If the thrown object is checked exception type, then the method getting the object has two choices: either provide a handler for it or broadcast in its header to throw that object. For example the method search in Listing 12.8 throws a checked exception of type ElementNotFoundException. Its caller main method must either provide the handler for it or re-throw this object. (We use the first choice in main method of Class1). Listing 12.9A shows a system of five methods calling each other. Here main method calls Func1, and Func1 calls Func2 and so on, until finally method Func4 throws an unchecked exception of type ErrType.

```java
public static void main(String[] args) {
    Func1();
}
public static void Func1() {
    Func2();
    System.out.println("In method Func1");
}

public static void Func2() {
    Func3();
    System.out.println("In Method Func2");
}

public static void Func3() {
    Func4();
    System.out.println("In Method Func3");
}

public static void Func4() {
    throw new ErrType();
    System.out.println("In Method Func4");
}
//Listing 12.9A
```
Figure 12.17A shows the mechanism for the propagation of this exception thru various method calls.

FIG. 12.17A
Since none of the methods in Listing 12.9 have a handler for the thrown object ErrType, it propagates from its thrower Func4 to Func3, to Func2, to Func1, and finally to the main method where the program would crash. One can see that in re-throwing an exception, no explicit throw statement is needed. Exceptions automatically propagate thru the hierarchy of method calls – until they find a handler or eventually crashing a program if no handler were found. This propagation hierarchy would terminate, as soon any of the method in the path of hierarchy provides a handler. For example we change the code in Func1 as

```java
try{
    Func2();
} catch(ErrType Ex){ /*code*/ }
System.out.println("In method Func1");
```

The re-throw from Func3, to Func2, to Func1 would continue as it did in Figure 12.17A. However, since Func1, now has a handler, the propagation stops there and
The object is thrown by Func4. Func3 that called Func4 gets the thrown object but does not have a handler. Func3 re-throws the Error Object to its caller Func2. Func2 does not have the handler either. So it also re-throws to its caller Func1. Func1 has the handler so it catches the object. Program completes successfully. Code following the handler in Func1 will be executed.

FIG. 12.17B
Notice that the program in Listing 12.9A would not have an output because all output statements are after the call that throws an exception – and exception never gets caught. However if we alter the code in Func1 to provide a handler (as shown earlier), the output statement from Func1 would be executed.

To re-throw or to provide a handler
Decision whether a method should provide a handler for explicit or implicit exceptions thrown to it depends upon as to where in the call sequence is the best place to process the thrown object. The call sequence we see in Figure 12.17 can be likened to software layers communicating thru each other. The lowest layer where the exception occurs is Func4, and top layer where program begins is the main method. In this sense the main method is the client of all other layers (Func1, Func2, Func3, and Func4). Certainly if Func4 threw a checked exception, the client would need to handle it. Situation is not so clear for unchecked exceptions, which client may or may not choose to handle. The most important consideration here is that if designer of class decides to change the implementation of class API, any design change should not break the older client code. For example if exception ErrType
were an unchecked exception and main method decides to provide a handler for thus altering the code in main as below.

```java
try {
    Func1();
}
catch (ErrType Ex){/*code*/}
```

Now if at some later date the designer of Func4 changes the method design and decided to throw some exception other than ErrType, then it would break the client code. This can also happen if design of other layers (Func1, Func2 etc.) changes to throw some other exception. Therefore the guiding criterion in deciding to re-throw or handle an exception is to ascertain that changes do not break the client code.

**Case Study1: Designing class methods with exceptions to obey class invariants**

In chapter four we discussed the concept of invariants and we showed the partial code for a bank account class. Let us state the invariant for BankAccount class again and redesign it – this time using Java defined exceptions to maintain the class invariant. We define a simple invariant for BankAccount class that at no time the minimum balance for the account holder shall go down below $25.00, and of course methods to make withdrawal and deposit must work correctly. (to complete).

**Case Study2: Improved IOManagement class**

(to complete)

**Questions**

1. (T/F) A try block can be used with out the use of subsequent catch block.
2. (T/F) A catch block can be used without use of try block preceding it.
3. (T/F) try catch blocks must be used as a pair.
4. A method throwing a checked exception must throw an object, which is of type:
   A] Throwable but not of type RuntimeException
   B] RuntimeException
   C] Exception
   D] Throwable
   E] All of above.
5. A method throwing an un-checked or run time exception must throw an object of type? (Select correct answer from the list given for question #4).
6. (T/F) The header of a method throwing a checked exception must broadcast
the exception being thrown.
7. (T/F) The header of a method throwing an un-checked exception must
broadcast the exception being thrown.
8. (T/F) The header of a method throwing an un-checked exception may
broadcast the exception being thrown.
9. (T/F) The call to the method throwing a checked exception must be placed in
a try block.
10. (T/F) The call to the method throwing an un-checked exception must be
placed in a try block.
11. (T/F) The call to the method throwing an un-checked exception may be
placed in a try block.
12. Explain that why the return value from the method alwaysReturnFalse
(shown below) would always be false no matter what boolean value is passed
to the method. (Note: This code will compile with a warning).

```
public static boolean alwaysReturnsFalse (boolean flag)
{
    while(flag)
    {
        try
        {
            return true;
        }
        finally
        {
            break;
        }
    }
    return false;
}
```
### Appendix A12.1

#### Class Inheritance hierarchy

<table>
<thead>
<tr>
<th>Class</th>
<th>Inheritance hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumberFormatException</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ArrayIndexOutOfBoundsException</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NullPointerException</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception Type</td>
<td>Class Hierarchy</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| NoSuchElementException | java.lang.Object  
|                         |  | java.lang.Throwable  
|                         |  | java.lang.Exception  
|                         |  | java.lang.RuntimeException  
|                         |  | java.util.NoSuchElementException  |
| ClassCastException      | java.lang.Object  
|                         |  | java.lang.Throwable  
|                         |  | java.lang.Exception  
|                         |  | java.lang.RuntimeException  
|                         |  | java.lang.ClassCastException  |
| FileNotFoundException  | java.lang.Object  
|                         |  | java.lang.Throwable  
|                         |  | java.lang.Exception  
|                         |  | java.io.IOException  
|                         |  | java.io.FileNotFoundException  |

**Table A12.1.1**