There were four things from which the Master was entirely free. He had no foregone conclusions, no arbitrary predeterminations, no obstinacy, and no egoism.

-<< Confucius>>

Chapter 11 Exception Handling

This chapter discusses exception handling. An exception is a run-time error that occurs during code execution. Java provides a rich number of exception handling classes in its API and powerful exception handling mechanisms to deal with a variety of exceptions. Java also allows programmers to define custom designed classes to handle specific exceptions that may occur in particular applications.

11.1 Exception classes in the Java API

Figure 11.1 lists commonly used exception classes in the Java API and their relational hierarchy.

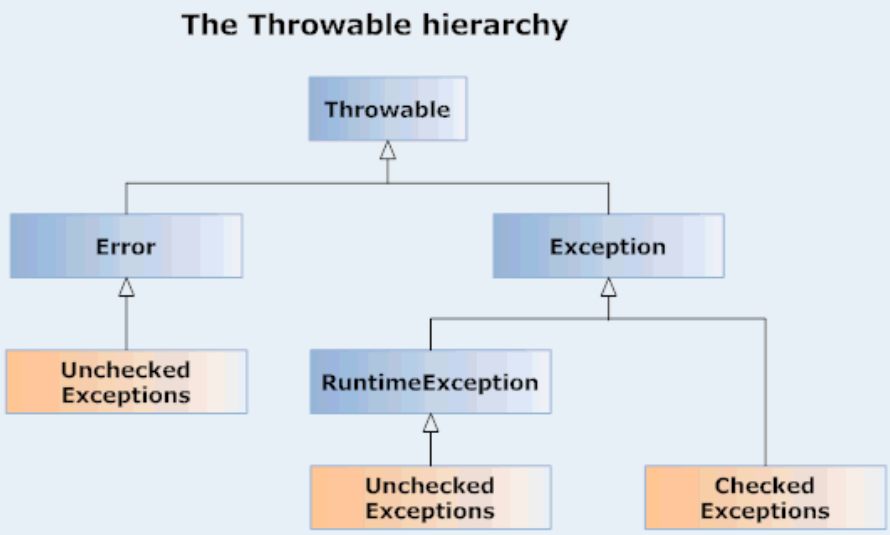

Figure 11.1 Commonly used exception classes in the Java API and their hierarchy

We may see from the chart that all exception classes in the Java API are inherited from the **Throwable** class. These exception classes can be categorized into two types: checked exceptions and unchecked exceptions. All exception classes inherited from the **Error** and **RuntimeException** classes are unchecked exceptions. An unchecked exception typically represents a condition that, as a rule, results from errors in program logic and cannot be reasonably recovered from at run time. The unchecked exception is handled by the JVM whenever it occurs during program execution. Common defined unchecked exception classes inherited from are: [NOTE: Professor Gao: Although your definitions of checked and unchecked exceptions are correct, I found them a bit unclear at first glance—and so may the reader. I hope you do not mind my taking the liberty of writing new definitions. LAS]

```
ArithmeticException
IllegalArgumentException
   NumberFormatException
IndexOutOfBoundsException
```

```
   ArrayIndexOutOfBoundsException
   StringIndexOutOfBoundsException
NullPointException
```

Other unchecked exception classes are inherited from the **Error** class. They deal with system and hardware errors, such as JVM errors, threading errors, I/O device errors, and system configuration errors. Normally these types of unchecked exceptions rarely occur. Until this point, all exceptions covered in our discussions have been unchecked exceptions.

All subclasses directly inherited from the **Exception** class are checked exceptions. The programmer must provide an exception handling mechanism to handle all checked exceptions; failing to provide such mechanisms will result in the generation of syntax errors during compilation. Common checked exception classes in the Java API are:

```
ClassNotFoundException
IOException
   EOFException
   FileNotFoundException
NoSuchMethodException
```

It is important to note that code dealing with file I/O operations may potentially raise checked exceptions. Exception handling mechanisms to handle such exceptions must be provided.

The Java API documentation clearly indicates the type of exception that will be thrown by a method should a potential exception issue be encountered.

In the following sections, we first discuss unchecked exception handling before moving on to checked exception handling.

11.2 Unchecked exception handling

Unchecked exceptions are not checked by the Java compiler. Strictly speaking, it is not necessary for the programmer to write code to handle unchecked exceptions. However, in the absence of programmer-provided code to handle these exceptions, the JVM will produce the following actions upon encountering an unchecked exception:

1. Terminate program execution ungracefully, and
2. Display technical information regarding the source and type of exception causing the execution.

The following are common causes of unchecked exceptions:

- Division by 0
- Array out of bounds
- Data type mismatch

- Use of Illegal argument
- Use of Illegal object

Most unchecked exceptions are the result of typographical or logical errors in coding made by the programmer. Careful programming may avoid the occurrence of unchecked exceptions.

Exceptions may occur due to unforeseeable errors in the code user's entry of data. In an effort to prevent "garbage in, garbage out" scenarios, we should provide exception handling code to verify these data before using them. This action will produce more reliable code.

11.2.1 Discovering unchecked exceptions

A key principle of exception handling is to discover the first place in the code where a given exception may occur and provide code to handle the exception. Common first places an exception may occur are:

- Constructors that instantiate an object.
- Input statements or method calls to acquire user data. Examples include method calls in the **Scanner** class and methods of the form **parseXxx()** that convert the data to other types.
- Statements performing calculations that may produce data out of bound exceptions.
- Statements accessing elements of arrays.
- Method calls associated with objects that do not exist, thereby causing a **NullPointerException**.

We should perform exception analysis using these criteria to make exception handling more efficient.

11.2.2 Why handle unchecked exceptions?

Why should programmers even consider handling unchecked exceptions, given their automatic handling by the JVM? Some literature states unchecked exceptions should not be handled by programmers but only by the JVM.

The primary reason in providing capabilities permitting programmer handling of unchecked exceptions is to allow necessary corrections in data acquisition and computing and let the code continue to execute, thereby avoiding an ungraceful termination of execution. Code reliability and fault-tolerance are our goals in exception handling. A good application should not stop executing due to logic errors or invalid data entry. Rather, it should provide exception handling mechanisms and the opportunity for the user to make corrections for the submission of valid input data.

11.3 Checked exception handling

Checked exceptions are exceptions that must be handled by the programmer. Failure to implement the required exception handling will result in compiler syntax errors. Methods or constructors in the API class documentation explicitly specifying checked exceptions that may be produced must have programmer-provided exception handling mechanisms to deal with such exceptions in the code.

11.3.1 Discovering Checked Exceptions

Checked exceptions may occur in:

- Method calls for file I/O operations, e.g., **FileNotFoundException**, **EOFException**.
- Method calls for threading operations, e.g., **InterruptedException**.
- Method calls for database operations, e.g., **SQLException**.
- Method calls for network connections and operations, e.g., **URLException**, **IOException**.
- Method implementations for interfaces in API, e.g., **CloneNotSupportedException** in the **Cloneable** interface implementation.

Proper exception handling mechanisms for checked exceptions must be implemented by the programmer. The Java compiler requires their proper handling in order to run the code.

11.3.2 Checked Exceptions

The Java API documentation clearly states all information concerning the checked exceptions a method may throw. The names of the checked exceptions are referenced in association with the Java keyword **throws**. For example, in a constructor dealing with the file input process **FileReader()**, the API documentation specifies the following signature:

```
public FileReader(File file) throws FileNotFoundException
```

indicating that in using an instantiation of this constructor for file input the programmer must provide an exception handling mechanism to deal with the **FileNotFoundException** exception. Another example may be found in the **close()** method used in file I/O. The API documentation for the **close()** method specifies:

```
public void close() throws IOException
```

requiring the programmer to provide exception handling code to deal with **IOException**.

***More Info**: In file I/O, especially in sequential file I/O, **EOFException** is commonly used to indicate when the end of the file has been reached; it is a special case. A detailed discussion of this topic is provided in Chapter 21*

11.4 Exception handling mechanisms

Exception handling mechanisms in Java for both checked and unchecked exceptions can be summarized in four basic models:

- Traditional **try-catch-finally** mechanism.
- Use of **throws** to throw a list of exceptions to the calling procedure.
- Combination of the above two models.
- Use of the **throw** statement to throw or re-throw exceptions.

11.4.1 Traditional exception handling mechanism

The "try-catch-finally" model is also known as the traditional exception mechanism. The use of a **finally** block is optional. It can be described as follows:

```
try {   //try block
    //Including all statements that possibly throw exceptions
   ...
}
//catch clause
catch (ExceptionName1 e) {
   //Exception handling message
   ...
}
catch (ExceptionName2 e) {
   //Exception handling message
   ...
}
...
catch (ExceptionNameN e) {
   //Exception handling message
   ...
   }
finally {  //Optional
   //Message that must be processed
   ...
}
```

Important concepts of the traditional exception handling mechanism are:

- A **try** block normally has at least one **catch** block. It may also have an optional **finally** block. If a **try** block does not have a **catch** block, however, it must have a **finally** block, otherwise a syntax error will occur.
- A **catch** block must be associated with an accompanying **try** block.
- A **finally** block is always executed whether or not an exception occurs.

- A **try** block may have multiple **catch** blocks but only one optional **finally** block. The **finally** block must be placed after all **catch** blocks or after the **try** block if no **catch** block is provided.
- A pair of braces must be used in each block of the try-catch-finally mechanism, even if only one statement is present in the block. Missing braces within block will result in the generation of a syntax error.

Let's discuss what happens when an exception occurs in the following code snippet:

```
try {
      int value = Integer.parseInt("12ab");
      // Other statements
      …
}
```

We hard-coded an illegal data element as the parameter for the for **parseInt()** method. Executing the statement will automatically throw a **NumberFormatException** object. This action is also known as an "implicit throw," since it is thrown automatically by the JVM. The flow of code execution will then jump out of the **try** block, ignoring any remaining statements in the **try** block, and resume at the first **catch** block provided in the code. An evaluation is performed at this point to determine if there is a match with the object that has been thrown. If the match is successful, code execution continues within the **catch** block. In the absence of a **catch** block, code execution continues in the mandatory **finally** block. Now, let us assume the code contains two **catch** blocks:

```
catch (InputMismatchException e) {
   //Process the exception and display the message
   …
}
catch (NumberFormatException e) {
   //Process the exception and display the message
   …
}
```

In the evaluation process following the throwing of the exception, code execution once again transfers to the first **catch** statement. The evaluation test determines the exception object does not match the specification of this **catch** statement and execution transfers to the second **catch** block. The second **catch** block is executed because a match exists between the thrown object and the specified argument type. The statements in the second **catch** block are executed and the exception is handled. In situations wherein more than one thrown object matches the argument specified in the **catch** blocks, only the first matched **catch** block will be executed.

Note: *The order of specifying **catch** blocks is important since only the first **catch** block will be executed if more than one **catch** block matches the thrown object. As such, **catch** blocks should be listed in order from most specific to general.*

If a thrown exception fails to match any exception specified in the list of **catch** blocks, the JVM will automatically trigger its own exception handling mechanism. The program will be terminated ungracefully, and the message of the exception handling will be recorded in a **stackTrace** displayed on the output console. The **stackTrace** is discussed in the following section.

If no exception occurs during the execution of statements contained in a **try** block, all listed **catch** blocks will be ignored. The **finally** block is always executed—whether an exception occurs or not.

A **finally** block should be used in the following situations:

- When statements must be executed after completing the successful processing of statements in the **try** block, e.g., to stop thread execution.
- When statements must be executed after an exception has been handled, e.g., issuing a **return** statement of a method class to close a file.
- When statements must be executed in the absence or failure to match the exception specified in the provided list of **catch** blocks before the JVM assumes processing of the exception. Examples include releasing memory space and closing audio, video, or picture display mode.

The following is an example of using a **finally** block:

```
 …
finally {       //Execute the return statement due to exception
  System.out.println("A default null has been returned due to the exception … “);
  return null;
}
…
```

Another example：

```
…
finally {       //Execute the close() due to the exception
   fileOutput.close();
}
```

And another one:

```
…
finally {
   System.out.println("Your verification of the data has failed after 4 attempts. “
                      + "Your session will be logged off. ");
   verifying.off();    //close the operation
}
```

If a **return** statement is used in a try-catch-finally mechanism, good coding practice dictates that a viable **return** statement be processed in the event of an exception to avoid the syntax error: “lack of return

statement." Consider the following code snippet producing a exception in which no **return** statement exists within the **catch** block:

```
public int validateInt() {
    try {
          …
          return Integer.parseInt(intString);  // produce exception
   }
   catch (NumberFormatException e) {

          // absence of return statement produces syntax error
          …
   }
}
```

The code may be modified to avoid the syntax error in the event of handling an exception as follows:

```
public int validateInt() {
   ...
  while (!valid) {
   try {
        intString = requestInput();  //Call requestInput()to receive data
        value = Integer.parseInt(intString); //May throw NumberFormatException

        valid = true;                       //If no exception occurs, stop the loop
   }
   catch (NumberFormatException e) {
        System.out.println(ExceptionMessage);    //Display message
        continue;                                //Continue the loop
   }
}
return value;                                    //Return the verified data
}
```

11.4.2 Propagation of exceptions

Exception propagation refers to the practice of using the Java keyword **throws** after a method name to propagate or pass one or more exceptions to the method caller(s). In nested method calls, exception propagation may propagate the thrown object(s) to its associated hierarchy of methods calls on the stack of nested calls maintained by the JVM.

A common example of exception propagation is to use **throws** to pass exception objects to the JVM:

```
public static void main(String[] args) throws IOException,InputMismatchException {
   ...
```

```
}
```

The above code structure throws both a checked **IOException** object and an unchecked **InputFormatException** object to the JVM. The reasons to do such propagation may be threefold:

- Trigger the JVM to handle the specified exceptions if no exception handling code is provided in the application.;
- Increase code readability. Although it may not be necessary in the event of unchecked exception handling, code readability and understanding is increased by examining the method header to learn what exceptions may potentially exist.; and
- You may still provide a **throws** list of exceptions even though an exception handling mechanism is provided in the code. However, the exception object will not be propagated to its method caller if a successful match of an included **catch** block exists to handle such an exception.

It must be noted passing a propagated exception to another method rather than the JVM requires the presence of code to either handle the passed exception or to propagate the exception to the next level of method call.

Exception propagation in nested method calls is discussed in Section 11.6.2. In the case of implementation of a **Cloneable** interface, particularly in overriding the **clone()** method, we use **throws** to pass **CloneNotSupportedException** to the superclass of the **clone()** method to handle the exception:

```
public Object clone() throws CloneNotSupportedException {
   return super.clone();
}
```

The above code will generate a "missing return statement" error if a traditional try-catch-finally mechanism is used in the code without the use of exception propagation.

11.4.3 More discussion of exception messages

In API exception classes, all subclasses propagate exception messages to the superclass **Throwable** using the statement **super(message)**. This mechanism allows us to easily use the methods provided in **Throwable** to retrieve these messages. Table 11.1 lists the methods commonly used to retrieve these messages.

Table 11.1 Commonly used methods in **Throwable** to retrieve exception handling messages

Method	Description
String getMessage()	Return the current exception handling message.
printStackTrace()	Display the exception handling message associated with this throwable recorded in the stack and its backtrace to the standard error stream.

String toString()	Return the exception handling message including the exception class name associated with this throwable.

For example, after executing the code snippet

```
try {
   int number = Integer.parseInt("123abc");      //Throws NumberFormatException
}
catch (NumberFormatException e) {                //Handle the exception
   System.out.println(e.getMessage());           //Display the exception message
   e.printStackTrace();             //Display the message recorded in the stack
   System.out.println(e);           //Or e.toString()
}
```

statement **System.out.println(e.getMessage())** produces the following information:

```
For input string: "123abc"
e.printStackTrace()displays the following message:
java.lang.NumberFormatException: For input string: "123abc"
        at java.lang.NumberFormatException.forInputString
        (NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:456)
        at java.lang.Integer.parseInt(Integer.java:497)
        at ExceptionHandlingTest1.main(ExceptionHandlingTest1.java:7)
```

and **System.out.println(e.toString())** or **System.out.println(e)** generates the following output:

```
java.lang.NumberFormatException: For input string: "123abc"
```

11.4.4 Examples

In this section more examples are presented using the concepts and coding techniques of exception handling previously discussed.

Example 1. Exception handling in **main()** method. Assume the code demonstrates sorting an array and the size of the array is determined by the user's data entry:

```
//Complete code is called ArrayValidationTest1.java in Ch11 from author's Website
public class ArrayValidationTest1 {
   public static void main(String[] args) {
   …
     while (choice.equals("y")) {
     try {
          System.out.print("Please enter an integer for the size of the array: " );
          size = sc.nextInt();      //May throw InputMismatchException
          int[] intArray = new int[size];
```

```
                                    //May throw NegativeArraySizeException
        System.out.println();

        ArrayDemo.fillArray(intArray);     //Call the static method
        Arrays.sort(intArray);             //Array sorting
        ArrayDemo.display(intArray);       //Display the sorting result
     }
     catch (InputMismatchException e) {  //Handle InputMismatchException
        System.out.println("You must enter an integer for array size...");
        count++;                                //Increase the count
        sc.nextLine();                          //Clear the buffer
        continue;                               //Continue to loop
    }
    catch (NegativeArraySizeException e) {
        //Handle NegativeArraySizeException
        System.out.println("You must enter a positive integer for array size.”);
        count++;                                //Increase the count
        sc.nextLine();                          //Clear the buffer
        continue;                               //Continue to loop
    }
    finally {
        if (count >= 3) {
              System.out.println("The application is terminated now due to 3
              wrong entries...");
              System.out.println("Review your entries and try running
              the program again.  Bye!”);
        break;             //Or: System.exit(0); Terminate the loop
        }
   }
...
```

The code handles two types of exceptions: **InputMismatchException** and **NegativeArraySizeException**. It provides three opportunities for the user to correct erroneous data entry. A counter is increased by 1 in each **catch** block to keep track of the quantity of user data entries. The **continue** statement resumes the next loop iteration for data verification. Since the **finally** block is always executed whether or not an exception has occurred, it will print the message prompting the user to review the input data only if three or more attempts have been made. A **break** (or **System.exit(0)**) statement terminates the code execution.

Example 2. Modify the code in **Example 1** to use a default array size if the user enters invalid data more than three times.

```
//Complete code called ArrayValidationTest2.java in Ch11 from author’s Website
…
    finally {
    if (count >= 3) {
        System.out.println("You’ve entered 3 wrong entries...");
```

```
         System.out.println("A default array size of 100 has been assigned.”);
         int[] intArray = new int[100];
         ArrayDemo.fillArray(intArray);         //Call static method
         Arrays.sort(intArray);                 //Array sorting
         ArrayDemo.display(intArray);           //Display result
         break;
    }
}  …
```

Example 3. Another coding technique to handle exceptions is to design a verification class, e.g., **Validator**, as a utility class containing static methods to verify data and handle a variety of exceptions. For example, in the following code:

```
//Complete code called ArrayValidationTest3.java in Ch11 from author’s Website
while (choice.equals("y")) {
   size = Validator4.arraySize(sc, "Please enter an integer for the array size: “);
   int[] intArray = new int[size];        //size is already verified

   ArrayDemo.fillArray(intArray);
   Arrays.sort(intArray);
   ArrayDemo.display(intArray);

   System.out.print("Continue? (y/n): ");
   choice = sc.next();
}
...
```

static method **Validator4.arraySize()** is called to verify the size of the array using an application-specific user prompt as the method parameter. We code method **arraySize()** in the **Validator4** class as follows:

```
//Complete code called Validator4.java in Ch11 from author’s Website
public static int arraySize(Scanner sc, String prompt) {
         boolean done = false;
         int count = 0;
         int size = 0;
         while (!done) {
          try {
             System.out.print(prompt );
             size = sc.nextInt();  //May throw InputMismatchException
             if (size < 0)
                 throw new NegativeArraySizeException();

             System.out.println();
             done = true;
          }
          catch (InputMismatchException e){//Handle InputMismatchException
                System.out.println("You must enter an integer for array size.”);
```

```
                count++;                          //Increase count
                sc.nextLine();                    //Clear buffer
                continue;                         //Continue to loop
            }
            catch (NegativeArraySizeException e) {
                                              //Handle NegativeArraySizeException
                System.out.println("You must enter a positive integer for array
                size...");
                count++;                          //Increase count
                sc.nextLine();                    //Clear buffer
                continue;                         //Continue to loop
             }
             finally {
                if (count >= 3) {
                      System.out.println("You've entered 3 wrong entries.");
                      System.out.println("A default array size 100 has been
                      assigned.");
                      size = 100;
                      break;
                }
             }
           }
       return size;
   }
```

In handling the situation in which an array size is less than 0, a **throw** statement is used to throw an anonymous object of **NegativeArraySizeException**. We discuss the **throw** statement in the next section. In the **finally** block, if **count >= 3**, a default size of 100 is used to establish the array length.

Example 4. It is recommended that an **Exception** class object be used in the last **catch** block of an exception handling "sequence" to handle any exception that may not be explicitly handled (or missed) in the code:

```
…
catch (Exception e)     {                 //The last catch block
   System.out.println("A exception occurred that cannot be handled in the code.\n
                      The following is the stack trace information: \n");
   System.out.println(e.print Stack Trace());          //Or: e.getMessage();
}
```

Since **Exception** or **Throwable** is a superclass of all exception classes, it will catch any type of exception. As such, it will not miss "capturing" any exception that has occurred and handled in the code.

11.5 Use of the throw statement

It is important to note the difference between the **throw** statement and the **throws** statement. A **throws** statement is declarative for the method. It tells the compiler the particular exception will be handled by the calling method. The **throw** statement, however, is used to force the code to explicitly throw an anonymous object of the exception if some condition is satisfied. It can also be used in a **catch** block or a **finally** block in which an object of the exception could be (re-)thrown to propagate the exception.

11.5.1 Exceptions JVM automatically throw

As previously discussed, if an unchecked exception occurs during code execution the JVM, or system, will automatically and implicitly throw an object of the exception. The JVM will also automatically handle the processing of the exception. This form of exception is known as an implicitly thrown exception. An implicitly thrown exception is handled by the JVM if the user does not provide code to handle the exception. For example, the JVM will automatically, i.e., implicitly, throw an object of **InputMismatchException** if an invalid data, such as a floating-point number, is entered in the statement:

```
int value = sc.nextInt();          //sc is an object of Scanner
```

In effect, the JVM uses a **throw** statement to throw this exception as:

```
throw new InputMistachException();
```

Although the object of **InputMismatchException** does not have a name (it is not important here), the JVM will retrieve the message recorded in the stack and display it if no exception handling code is provided in the program.

11.5.2 I can throw, too

Programmers may use a **throw** statement to explicitly throw an exception. For example:

```
try {
   if (!sc.hasNextInt())           //May produce InputMismatchException
          throw new InputMismatchException(); //Throw this exception
}
catch (InputMismatchException e) {  //Handle this exception
   System.out.println("Incorrect integer entry.  Please check and try again...");
}
...
```

There is another constructor with a **String** type argument provided in any API exception classes

```
public InputMismatchException(String message)
```

that may be used to throw an exception with a specified message, for example:

```
throw new InputMismatchException("Incorrect integer entry.  Please check and try
```

```
again...");
```

In the **catch** block associated with the exception,

```
catch (InputMismatchException e) {
   System.out.println(e);
}
```

the following message will be displayed:

```
Incorrect integer entry.  Please check and try again...
```

A **throw** statement can also be used in a **catch** block to explicitly throw another exception:

```
...
catch (IOExcetion e) {
   System.out.println(e);
   throw new FileNotFoundException();
}
```

In the above code snippet, the **catch** block throws a new **FileNotFoundException** to the method caller or the upper level of exception handling stack after it processes the caught exception. Note that the original exception handling message will be lost if it immediately throws a new exception. We discuss this topic in section 11.8.1. The following list describes situations in which a **throw** statement might be used in a **catch** block:

- Although the **catch** block handles the caught exception, more exception processing is required to produce a more accurate or specific exception handling message.
- The matched **catch** block may not suitable to handle the matched exception and will need to throw it or throw a new exception to the next level of exception handling code.

***Note**: When using a **throw** statement to throw an exception, it must be the last statement in the **try, catch**, or **finally** block; otherwise a syntax error will result. If an exception is **RuntimeException**, it is common practice to use a **throws** statement to throw it to the JVM rather than use a **try-catch-finally** structure to handle it. An example can be found in **RuntimeExceptionTest.java** in Ch11 from the author's Website.*

11.5.3 Re-throw an exception

Re-throwing an exception refers to the situation in which a **catch** block, after catching an object of an exception from a **try** block, uses a **throw** statement to re-throw the object yet again to the calling method or upper level of exception handling stack maintained by the JVM to be handled. It should be noted when a **catch** block re-throws such an exception a **throws** statement must be included in the method header listing the exception:

```
...
public void someMethod() throws Exception {//List the re-thrown exception
   try {
          ...
          throw new Exception ();    //Throw an exception
   }
   catch (Exception e) {
          ...
          throw e;                   //Re-throw the exception
   }
}
```

In the **catch** block, statement **throw e** re-throws **Exception**, thereby throwing the object of **Exception** to its caller or the next upper level of exception handling in the stack. If the re-thrown event occurs in the **main()** method, the JVM will handle this exception.

The purpose of re-throwing is similar to the use of the **throw** statement in a **catch** block—both throw an object of an existing exception.

11.6 Nested exception handling

An exception handling mechanism can be of a nested structure format within the coding. An example of this was discussed in the last section. The purposes of using nested exception handling are:

- Inner exception handling code is used to handle a more specified exception than the outer exception handling code.; and
- If the inner exception handling mechanism cannot fully handle the exception it can automatically propagate to its outer exception handling code.

Nested exception handling can also happen in nested method calls. It has its own features that are different from the regular exception propagation discussed at the beginning of this chapter. In this section, we discuss these important concepts and coding techniques.

11.6.1 Nested models

Nested exception handling may have the following models:

Model 1:

```
try {                        //Outer exception handling
   ...
   try {                     //Inner exception handling
          ...
          }
   catch (...) {             //Inner catch block
```

```
        ...
   }
   ...
   finally {                //Inner finally block (optional)
        ...
   }
}                           //End of inner try block
catch (...) {               //Outer catch block
   ...
}
...
finally {                   //Outer finally block (optional)
   ...
}
```

The following possibilities may occur during code execution:

- If no exception occurs, the program will execute normally and the **finally** blocks (if present) in both the inner and outer exception handlers are always executed.
- If the inner exception mechanism does not match any thrown exception, its **finally** block (if present) is always executed. Program control will propagate the exception to the first **catch** block in the outer exception handling, with each **catch** block will be evaluated and the **finally** block being executed (if present). The JVM, however, will step in to complete the exception handling if no matched **catch** block in outer exception handling mechanism is executed.
- If the exception occurs in the outer exception handling mechanism, each **catch** block will be evaluated and its **finally** block executed (if presented). If there is no matched **catch** block, the JVM will step in after the **finally** block is executed and all inner exception handling code, including its **finally** block, will be omitted.
- A **throw** statement may be used to perform an explicit exception throw in an outer or inner exception handling mechanism. For example, a **throw** statement used in the inner exception handling code will throw the exception to its outer exception handling code.
- The code may also re-throw an existing exception in nested exception handling. For example, a **catch** block in the inner exception handling mechanism may re-throw an exception to its outer exception handling code, and a **catch** block in the outer exception handling code may re-throw an exception to its caller or the next level of the exception handling in the stack.

It is important to note that the abuse of a nested exception handling mechanism will reduce execution speed and code readability. It should be used properly.

Model 2:

```
...
try  {
   method1();          //Call method1()
   ...
```

```
   method2();          //Call method2()
}
catch (...) {
   ...
}
finally {              //Optional
   ...
}
...
public void method1() {
   try {
          ...
   }
   catch (...) {
          ...
   }
   ...
}
public void method2() {
   try {
          ...
   }
   catch (...) {
          ...
   }
   ...
}
```

In this model, each method has its own exception handling mechanism. This form is commonly used in more sophisticated coding applications. Similarly, it may explicitly throw an exception or re-throw an existing exception in any **catch** block to the next level of exception handling.

11.6.2 Propagation in nested exception handling

We first discuss a typical example of exception handling propagation in a nested exception handling mechanism using Model 2 (see above) wherein each method throws the same exception to its caller. Figure 11.2 illustrates this concept.

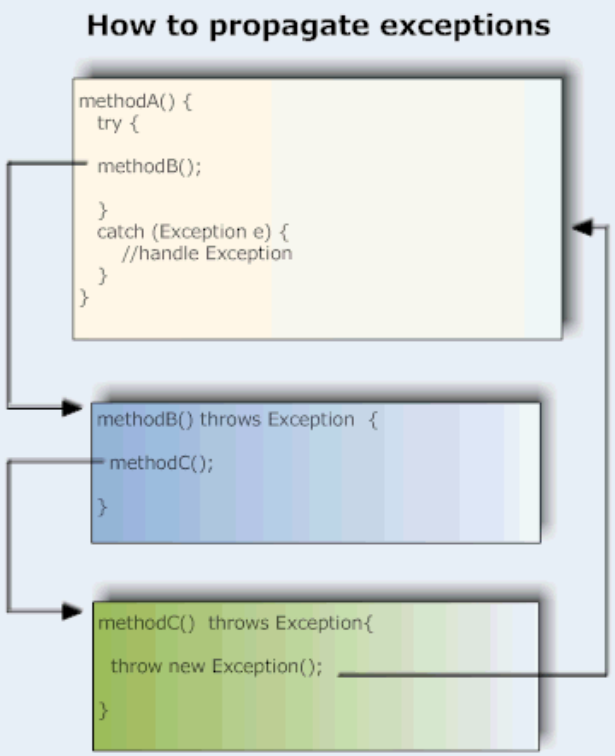

Figure 11.2 Propagation in a nested exception handling

In Figure 11.2, **methodA()** provides the uppermost level of exception handling below the JVM. Should **methodA()** throw this exception, the JVM will step in to handle the propagated exception. The propagation changes the normal execution flow: instead of executing the unfinished code in each nested method, the execution directly jumps up to the exception handling mechanism, **methodA()**, to handle the exception.

Propagation in a nested exception handling is commonly used in complicated networking programming, remote method calling, and Web services applications.

11.6.3 Nested exception re-throw

Nested exception re-throw refers to a coding scenario involving nested method calls in which each method provides an exception handling mechanism. In each **catch** block, a **throw** statement is used to re-throw the existing exception to the method's caller. Figure 11.3 illustrates this concept.

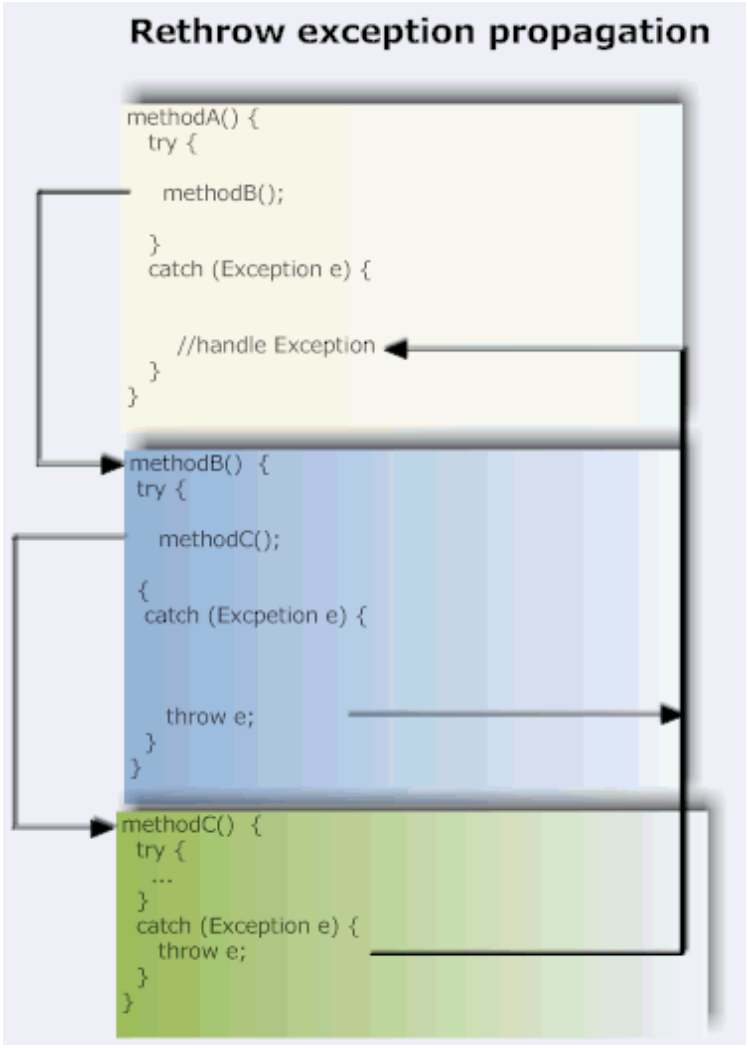

Figure 11.3 Re-throw the exception in nested method calls

In Figure 11.3, the **catch** block of **methodC()** re-throws a caught exception to its caller, **methodB()**. In **methodB()**, the exception is, in turn, propagated to **methodA()**. The exception may be handled in **methodA()** or re-thrown to the JVM.

Propagation in a nested re-throw is similar to propagation in a nested exception handling. The key difference, however, is that propagation in a nested re-throw will cause the flow of execution to travel through each nested method until it reaches the exception handling code. In the propagation of nested exception handling, execution will jump directly to the origin of the occurring exception. This technique is commonly used in large scale applications in networking, remote method calls, and Web services.

11.7 User-defined exception classes

Although we may use the standard **Exception** class to handle a specified exception:

```
try {
```

```
    // If there is an exception handle with custom handler
    //  NOTE:  We may use any API exception class
    if (someCustomException)
        throw Exception ("Custom message about the exception...");
}
catch (Exception e) {
   System.out.println(e);            //Display the specified message
}
```

in this example we "borrow **Exception** to make a gift to our application" by using the **Exception** constructor form with a **String** argument to pass the particular custom message as result of the exception handling. Programmers often define their own exception classes using inheritance from the API **Exception** class to handle more specified or custom exceptions for particular applications. Of course, we may use any of standard exception class to do the same thing above.

11.7.1 Define your own exception classes

The commonly used template to code a user-defined exception class is:

```
public class CustomException extends Exception { //Inherit an API exception class
   public CustomException()  {}                  //No argument constructor
   public CustomException(String message) {      //An argument constructor
          super(message);                        //Call the super constructor
   }
}
```

You may inherit **Throwable** as the superclass to define your own exception class. Although the constructor without an argument may not be used very often, it does provide a measure of convenience in the instantiation of the exception object. The second constructor will pass the particular message of the exception handling to the superclass where the message will be recorded. The message can be retrieved by calling **toString()**, **getMessage()**, or **printStackTrace()**.

Let's discuss an example. Assume we need to verify if a user-provided data for age is a positive value. We may code this exception using the template:

```
public class NegativeAgeException extends Exception { //Or: extends Throwable
   public NegativeAgeException()  {}
   public NegativeAgeException(String message) {
          super(message);
   }
}
```

The following code snippet uses an exception handling code that applies this user-defined exception class, **NegativeAgeException**:

```
//Complete code called NegativeAgeExceptionTest.java in Ch11 from author's Website
...
```

```
try{
   String ageString = JOptionPane.showInputDialog("Enter your age: ");

   if (Integer.parseInt(ageString) < 0)
        throw new NegativeAgeException("Please enter a positive age.");
   else
        JOptionPane.showMessageDialog(null, ageString, "Age", 1);
}
catch(NegativeAgeException e){
    System.out.println(e);
}
...
```

You may also use the constructor without an argument to create an object of **NegativeAgeException** and then print the specified message in the **catch** block:

```
try{
   String ageString = JOptionPane.showInputDialog("Enter your age: ");

   if (Integer.parseInt(ageString) < 0)
        throw new NegativeAgeException();
   else
        JOptionPane.showMessageDialog(null, ageString, "Age", 1);
}
catch(NegativeAgeException e){
    System.out.println("Please enter a positive age.");
}
...
```

This will produce the same result as previous example.

11.7.2 User-defined exception handling

The mechanism of user-defined exception handling refers to a code structure wherein a **try** block containing an **if** statement is used to test and throw a user-defined exception and an associated **catch** block "matching" the thrown exception object handles the exception in a more specific manner. This can be defined as follows:

```
try {
   ...
   if (someExceptionConditon == true) {
          throw new CustomException("A custom exception xxx occurred. Please
          check your entry...")
   ...
   }
catch (CustomException e) {
   ...
```

```
}
```

The exception handling message can be included in the constructor as its argument or provided in the **catch** block with code to print out the message.

It is very important to note that a standard exception may occur before the custom exception. For example, in a code needing to verify that an age is a positive integer, user-defined data for the age may not be a legal integer type. As such, a **NumberFormatException** exception will occur before a (possible) **NegativeAgeException** exception. You must provide code to handle both exceptions. Placing the **catch** block associated with handling the **NumberFormatException** exception first may provide better coding efficiency and readability.

```
try {
   ...
   if (Integer.parseInt(ageString) < 0)
          throw NegativeAgeException("Please enter a positive age.");
   else
          ...
   }
   catch (NumberFormatException e) {
          System.out.println(e);
   }
   catch (NegativeAgeException e) {
          System.out.println(e);
   }
   ...
```

Using the above code snippet as an example, if **ageString** receives an illegal integer data, e.g., "25abc", method **parseInt()** will throw **NumberFormatException** automatically and the first **catch** block will handle this exception. The **throw** statement relating to **NegativeAgeException** will not be executed.

11.7.3 Modified Validator class

Based on our discussion of exception handling so far, we may modify the **Validator5** class to include more exception handling functions and also use of regular expressions discussed in Chapter 10:

```
//Complete code called Validator5.java in Ch11 from author's Website
public class Validator4 {
   …
public static int intWithRange(Scanner sc, String prompt, int min, int max)
{
   boolean done = false;
   int count = 0;
   int data = 0;
   while (!done) {
     try {
      System.out.print(prompt );
      data = sc.nextInt();    //May produce InputMismatchException
```

```
     if (data < min)        //Exceeded the min
       throw new IntegerOutOfRangeException("Data out of minimum "+ min +
       " range exception.");
     if (data > max)        //Exceeded the max
          throw new IntegerOutOfRangeException("Data out of maximum "+ max
          + " range exception.");
     System.out.println();
     done = true;
    }
    catch (InputMismatchException e) {//Handle InputMismatchException
         System.out.println("You must enter an integer...");
         count++;                      //Increase count
         sc.nextLine();                //Clear buffer
         continue;                     //Continue loop
    }
    catch (IntegerOutOfRangeException e) {
                                       //Handle IntegerOutOfRangeException
         System.out.println(e);
         count++;                      //Increase count
         sc.nextLine();                //Clear buffer
         continue;                     //Continue loop
    }
   }
   return data;
   }
   ...
}
//User-defined exception class
class IntegerOutOfRangeException extends Throwable {
                                                          //Or: extends Exception
   public IntegerOutOfRangeException() {}
   public IntegerOutOfRangeException(String message) {
         super(message);
   }
}
```

11.8 Exception chaining

The purpose of exception chaining is to prevent the loss of a previously occurring exception when a successive exception is thrown. It is a common situation in which a user-defined exception is thrown in response to another exception occurring. For example, in the **catch** block responding to **NumberFormatException** you may subsequently throw **NotIntegerException**. Exception chaining uses the constructor and methods provided in class **Throwable** to chain these two related exceptions together so information in **NumberFormatException** will not be lost.

11.8.1 "Losing" an Exception

The following code

```
catch (IOException e) {                    //Handle IOException
   throw new CustomIOException();          //Throw another exception
}
```

illustrates a typical example of losing an exception. After a new exception, **CustomIOException**, has been thrown the message stored in the previous exception, **IOException**, is erased by the JVM.

11.8.2 Realizing exception chaining

The **Throwable** class was introduced in JDK1.4. It can be used to realize exception chaining, thereby providing a successful means of avoiding the loss of previously occurring exceptions. Commonly used constructors and methods used in exception chaining are listed in Table 11.2.

Table 11.2 Constructors and methods used for exception chaining in **Throwable**

Constructor/Method	Description
Throwable(Throwable cause)	Create an object of Throwable with the specified cause and a detail message of (cause==null ? Null : cause.toString()) which typically contains the class and a detail message of cause.
Throwable(String message, Throwable cause)	Create an object of Throwable including the message and the exception object
Throwable getCause()	Retrieve the object of exception from the upper level or null if the cause is nonexistent or unknown.
initCause(Throwable cause)	Set the object as a upper level exception source. Initializes the cause of this throwable to the specified value. It cannot be used if a constructor already created an object of the exception from the upper level.

Exception chaining may be performed in two different ways:

Way 1:

- In user-defined exception handling, use the **Throwable(Throwable)** constructor to chain or record the previous exception together with the custom exception handling. For example:

```
public class CustomIOException exdends Exception {
   public CustomIOException() {}
   public CustomIOException(Throwable cause) {//cause is previous exception
          super(cause);                        //Record or save this exception
   //other exception handling statements
   ...
   }
}
```

- In the **catch** block, **throw** the user-defined exception with the information of the previous exception:

```
catch (IOException e) {
 throw CustomIOException(e); //Message in IOException recorded to CustomException
}
```

- When the method caller or the exception handling in the next level of exception handling stack handles this thrown exception it will call method **getCause()** method to display all messages associated with the chained exceptions:

```
catch (CustomIOException e) {
   System.out.println("Custom IO Exception info: " + e);
                                    //Display the custom exception message
    System.out.println("Previous IO Exception info: " + e.getCause());
                                    //Display the previous exception message
}
```

Way 2:

Assume there is no **Throwable** constructor used in a user-defined exception class. We may directly use the **initCause()** method in class **Throwable** to perform exception chaining as follows:

```
try {
  ...
  if (!inputFile.canRead())  {     //If the file cannot be read in
     cannotRead = new CustomIOException();  //Create an object of custom exception
     cannotRead.initCause(new IOException()); //Chained with IOException
     throw cannotRead;                        //Throw both exceptions
  }
  ...
```

Since method **initCause()** accepts **Throwable** as its argument it can be used when a previous exception is passed into the current exception object, thereby chaining together the exceptions. This form of exception chaining is commonly used in exception handling requiring the processing of both exceptions.

11.9 Assertion

The **assert** statement was introduced in JDK1.4. It is specifically designed for code debugging and testing. It may be placed at any place within the code as a statement, and is often referred to as a "run break." A condition for a run break is provided after the keyword **assert** to declare a termination of the execution if the condition is not satisfied. For example, the run break code statement:

```
assert age > 18
```

will cause code execution to stop if the assertion "**age** is greater than 18" fails.

Assertion evaluation must be enabled before code execution. The following sections discuss the use of assertion.

11.9.2 Coding an assert statement

The syntax of the **assert** statement is:

```
assert booleanExpression [: message];
```

wherein

assert – Java keyword.

booleanExpression – Java boolean expression to declare a condition that must be satisfied.

[:**message**] – optional display message.

As with other Java statements, the **assert** statement must end with a semicolon.

For purposes of code readability, we use parentheses to identify the boolean expression in an **assert** statement.

Let's discuss several examples using assertions:

Example 1.

```
//Complete code called AssertTest.java in Ch11 from author's Website
int age = 17;
assert (age > 18) : "Age must be greater than 18.";
```

The code will terminate because **age** is less than 18. The JVM will display the following message with the specified information:

```
Exception in thread "main" java.lang.AssertionError: Age must be greater than 18.
at AssertTest.main(AssertTest.java:11)
```

In actuality, the JVM treats the assertion failure as a thrown **AssertionError** exception. The **assert** statement will be omitted and the code will continue to execute if age is greater than 18.

Example 2.

```
//Complete code called AssertTest.java in Ch11 from author's Website
double total = 219.98;
assert (total > 0 .0 && total < 200.0) : "total: " + total + " – out of range.";
```

Code execution will halt because the value of **total** (219.98) fails to meet the requirements of the **assert** statement. The JVM will display the following message:

```
total: 219.98 – out of range.
at AssertTest.main(AssertTest.java:10)
```

11.9.3 Enabling and Disabling Assertion Processing

The methods of enabling and disabling assertions depend on the manner in which the code is to be executed. If the program is executed in command line console mode under the operating system, the following command will enable the processing of assertions and execute the code:

java –ea ClassName

Option **ea** is short for enable. This option will enable assertion processing in the Java execution command. The following command sequence

java ClassName

without the **ea** option disables assertion processing.

More info: *You may also use **java –da ClassName** to execute the code disabling the assertion. Option **da** is short for disable.*

Executing code in an IDE, e.g., Eclipse or TextPad, requires that assertions be enabled in the following configurations:

NetBeans:

In the File menu, select Project Properties, select run, and enter **–enableassertions** in VM Options.

Eclipse:

In the **Run** menu,

```
Run...
Arguments
```

select parameter **arguments.** In the **VM Arguments** window, enter:

-ea (or **enableassertion**)

and then select **apply** and **run**.

If you would like to disable the assertion, go to VM window and delete the **–ea** option.

TextPad:

To enable assertions, access the **Configure** menu, select **Preference**, **Tools**, and **Run Java Application**. In the **Parameter** window, enter:

-ea followed by a space

Then click **OK** to confirm.

If you would like to disable the assertion, go to the **Parameter** window and delete the **–ea** option.

Exercises

1. What is a checked exception? What is an unchecked exception? Use examples to explain, compare and contrast.
2. What are commonly used exception handling mechanisms? Use examples to explain ~~them~~.
3. What is a clause of **catch** blocks? Use examples to explain. Explain the best way in which to order a clause of **catch** blocks in a Java program.
4. Use examples to explain the features of a **finally** block. What code should be included in a **finally** block?
5. What is exception propagation? Use examples to explain how you would propagate an exception.
6. Open file **ArrayValidationTest.java** in Ch11 from the author's website. Add code to verify that the maximum size of an array does not exceed 1000 elements using the following two methods:
    a. Verify using the **Throwable** and **Exception** classes; and
    b. Verify by using a class called **ArrayOutOfMaxException**.
7. Open file **Validator4.java** in Ch11 from the author's Website. Using the example shown in Section 11.4.4 as a guide, add the following static methods into the class to validate positive integer and double precision values:
    a. ValidatePositiveInt(Scanner sc, String prompt, int num)
    b. **ValidatePositiveDouble(Scanner sc, String prompt, double num)**

    Parameter definitions of the above methods are:

    - sc a scanner breaks its input into tokens using a delimiter pattern, which by default is whitespace. The resulting tokens may then be converted into values of different types using various methods.
    - prompt an input message to be displayed for data entry.
    - num data to be verified.

    See Section 11.4.4 for further information.

8. Write a driver code to test the modifications made to **Validator4.java** in Question 7. Document your source code and save the files.

9. Use examples to explain the differences between the **throws** and **throw** statements.

10. What is meant by the concept of re-throwing an exception? Why is re-throwing needed?

11. What is a nested exception? Use examples to explain the different ways in which exceptions can be nested.

12. What is the propagation of a nested exception? Explain the differences between a nested exception and the propagation of a nested exception.

13. What is meant by the concept of re-throwing in a nested exception? Explain the differences between a re-throw in a nested exception and nested exception propagation.

14. Write the following user-defined exception classes to handle the specified exception. Each class must include a non-argument form of a constructor:

    a. IllegalSelectionException – handle an illegal selection exception associated with a list of items from which the user must specify an item selection.

    b. IntOutOfRangeException – an integer out of range exception; and

    c. PriceOutOfRangeException – a price outside a specified range.

    Write a driver code to test the three user-defined exception classes. Hint: You may wish to provide a series of prompts to the user regarding a list of items from which to select, a range of integers, and a range of prices. Code should be written to handle any pattern of occurring exceptions.

15. What is meant by the term “exception losing?” Explain this concept using examples.

16. What is exception chaining? How do you realize or implement exception chaining?

17. What is an assert statement? What is the purpose of assertion?

18. Open file **ArrayValidatorTest.java** in Ch11 from the author's Website. Add an **assert** statement to verify the specified array size has not exceeded 1000 elements. If the assertion fails, display a message indicating the array size has exceeded 1000 elements. Document your code and save the files.