Module 4

Packages: Packages, Packages and Member Access, Importing Packages.

Exceptions: Exception-Handling Fundamentals, Exception Types, Uncaught Exceptions, Using try and catch, Multiple catch Clauses, Nested try Statements, throw, throws, finally, Java's Built-in Exceptions, Creating Your Own Exception Subclasses, Chained Exceptions.

Defining a Package

- **A set of classes and interfaces grouped together are known as Packages in JAVA.**
- **To create a package is quite easy: simply include a package command as the first statement in a Java source file.**
- Any classes declared within that file will belong to the specified package.
- The package statement defines a name space in which classes are stored.

- If you omit the package statement, the class names are put into the default package, which has no name
- While the default package is fine for short, sample programs, it is inadequate for real applications.
- **This is the general form of the package statement:**

package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

package *pkg1*[.*pkg2*[.*pkg3*]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in **java\awt\image** in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }

  void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}

class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];

    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);

  for(int i=0; i<3; i++) current[i].show();
  }
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Packages and Member Access

- **Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods**.
- Packages act as **containers for classes and other subordinate packages.**
- Classes act as containers **for data and code**.
- **The class is Java's smallest unit of abstraction**. Because of the interplay between classes and packages, Java **addresses four categories of visibility for class members**:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Table 9-1 sums up the interactions.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 9-1 Class Member Access

Table 9-1 applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

Importing Packages.

- **Java includes the import statement to bring certain classes, or entire packages, into visibility.**
- Once imported, a **class can be referred to directly, using only its name.**
- The import statement is a convenience to the programmer and is not technically needed to write a complete Java program.
- If you are going to refer to a few dozen classes in your application, however, the import statement will save a lot of typing.
- In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.
- This is the general form of the import statement:

$$\text{import } pkg1\ [.pkg2].(classname \mid *);$$

- **Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.).**
- **you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package.**
- This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

In this version, **Date** is fully-qualified.

```
package MyPack;

/* Now, the Balance class, its constructor, and its
   show() method are public. This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
  String name;
  double bal;

  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
```

As you can see, the **Balance** class is now **public**. Also, its constructor and its **show( )** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;

class TestBalance {
  public static void main(String args[]) {

    /* Because Balance is public, you may use Balance
       class and call its constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // you may also call show()
  }
}
```

As an experiment, remove the **public** specifier from the **Balance** class and then try compiling **TestBalance**. As explained, errors will result.

MODULE-4
CHAPTER-2
Exceptions

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run time error

Exception-Handling Fundamentals

- **Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.**
- Program statements that you want to monitor for exceptions are **contained within a try block**.
- If an exception occurs within the **try block, it is thrown**.
- **Your code can catch this exception** (using catch) and handle it in some rational manner.
- **System-generated exceptions are automatically thrown by the Java run time system.**

- **To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause**.
- Any code that absolutely must be executed after **a try block completes is put in a finally block**.

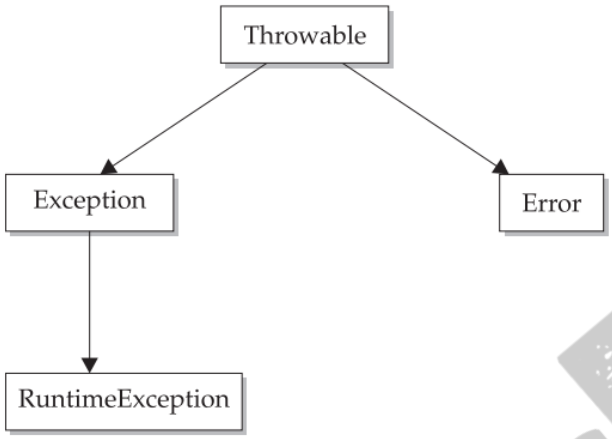
This is the general form of an exception-handling block:

```
try {
    // block of code to monitor for errors
}

catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}

catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// ...
finally {
    // block of code to be executed after try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred.

Exception Types

The top-level exception hierarchy is shown here:



Exception Types

- **All exception types are subclasses of the built-in class Throwable.**
- **Thus, Throwable is at the top of the exception class hierarchy.**
- **Immediately below Throwable are two subclasses that partition exceptions into two distinct branches.**
- **One branch is headed by Exception.**
- **This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.**
- **There is an important subclass of Exception, called RuntimeException.**
- **Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.**
- **The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program.**
- **Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.**
- **Stack overflow is an example of such an error.**

Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
  public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
  }
}
```

- **we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.**
- **Any exception that is not caught by your program will ultimately be processed by the default handler.**
- **The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.**

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:4)
```


Using try and catch

- **Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself.**
- **Doing so provides two benefits.**

  First, it allows you to fix the error.

  Second, it prevents the program from automatically terminating.

- **Most users would be confused, to prevent this Programmers used guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block.**
- **Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.**

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
     d = 0;
     a = 42 / d;
     System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
     System.out.println("Division by zero.");
    }


    System.out.println("After catch statement.");
   }
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Displaying a Description of an Exception

Throwable overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println( )** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
  System.out.println("Exception: " + e);
  a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

Multiple catch Clauses

- **In some cases, more than one exception could be raised by a single piece of code.**
- **To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.**
- **When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.**
- **After one catch statement executes, the others are bypassed, and execution continues after the try / catch block.**
- **The following example traps two different exception types:**

```
// Demonstrate multiple catch statements.
class MultipleCatches {
  public static void main(String args[]) {
    try {
       int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

This program will cause a division-by-zero exception if it is started with no command-line arguments, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

```
C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultipleCatches TestArg
a = 1
Array index oob:  java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

Nested try Statements

- **The try statement can be nested.**
- **That is, a try statement can be inside the block of another try.**
- **Each time a try statement is entered, the context of that exception is pushed on the stack.**
- **If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.**
- **This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.**
- **If no catch statement matches, then the Java run-time system will handle the exception.**
- **Here is an example that uses nested try statements:**

```
// An example of nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command-line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
        /* If one command-line arg is used,
           then a divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero
```

```
            /* If two command-line args are used,
               then generate an out-of-bounds exception. */
            if(a==2) {
               int c[] = { 1 };
               c[42] = 99; // generate an out-of-bounds exception
            }
          } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out-of-bounds: " + e);
          }

        } catch(ArithmeticException e) {
          System.out.println("Divide by 0: " + e);
        }
      }
    }
```

OUTPUT

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException:42
```


throw

- **So far, you have only been catching exceptions that are thrown by the Java run-time system.**
- **However, it is possible for your program to throw an exception explicitly, using the throw statement.**
- **The general form of throw is shown here: throw ThrowableInstance;**
- **Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.**

- **Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter in a catch clause or creating one with the new operator.**
- **The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.**
- **The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.**
- **If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.**
- **Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.**

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // rethrow the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, **new** is used to construct an instance of **NullPointerException**. Many of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**. It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable**.

Throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a throws clause in the method's declaration.
- **A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.**
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
  // body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try** / **catch** statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```


finally

- finally creates a block of code that will be executed after a try /catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown.
- **If an exception is thrown, the finally block will execute even if no catch statement matches the exception**.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The finally clause is optional.

- However, each try statement requires at least one catch or a finally clause

```
// Execute a try block normally.
static void procC() {
  try {
    System.out.println("inside procC");
  } finally {
    System.out.println("procC's finally");
  }
}

public static void main(String args[]) {
  try {
    procA();
  } catch (Exception e) {
    System.out.println("Exception caught");
  }

  procB();
  procC();
}
}
```

In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Java's Built-in Exceptions

- In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions**. The unchecked exceptions defined in java.lang are listed in Table 10-1.**
- Table 10-2 lists those exceptions defined by **java.lang** that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself.

- These are called **checked exceptions.**
- In addition to the exceptions in **java.lang**, Java defines several more that relate to its other standard packages.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 10-2 Java's Checked Exceptions Defined in **java.lang**

Creating Your Own Exception Subclasses

- **The Exception class does not define any methods of its own.**
- It does, of course, inherit those methods provided **by Throwable**.
- Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.
- They are shown in Table 10-3. You may also wish to override one or more of these methods in exception classes that you create.
- Exception **defines four public constructors.**
- Two support chained exceptions.
- The other two are shown here:

```
Exception( )
Exception(String msg)
```

- The first form creates an exception that has no description.
- The second form lets you specify a description of the exception.

Method	Description
final void addSuppressed(Throwable *exc*)	Adds *exc* to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the **try**-with-resources statement.
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.
final Throwable[ ] getSuppressed( )	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the **try**-with-resources statement.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println**( ) when outputting a **Throwable** object.

Table 10-3 The Methods Defined by **Throwable**

The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString( )** method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}

class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Normal exit");
  }

  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

This example defines a subclass of **Exception** called **MyException**. This subclass is quite simple: It has only a constructor plus an overridden **toString( )** method that displays the value of the exception. The **ExceptionDemo** class defines a method named **compute( )** that throws a **MyException** object. The exception is thrown when **compute( )**'s integer parameter is greater than 10. The **main( )** method sets up an exception handler for **MyException**, then calls **compute( )** with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Chained Exceptions

- **Beginning with JDK 1.4, a feature was incorporated into the exception subsystem: chained exceptions.**
- **The chained exception feature allows you to associate another exception with an exception.**
- **This second exception describes the cause of the first exception**.
- **For example, imagine a situation in which a method throws an ArithmeticException because of an attempt to divide by zero**.
- However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly.
- **Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error.**
- **Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.**
- To allow chained exceptions, two constructors and two methods were added to Throwable. The constructors are shown here:

Throwable(Throwable *causeExc*)
Throwable(String *msg*, Throwable *causeExc*)

- In the first form, causeExc is the exception that causes **the current exception**.
- That is, **causeExc is the underlying reason that an exception occurred**.
- **The second form allows you to specify a description at the same time that you specify a cause exception**.
- **These two constructors have also been added to the Error, Exception, and RuntimeException classes**.
- **The chained exception methods supported by Throwable are getCause( ) and initCause( ).**
- These methods are shown in Table 10-3 and are repeated here for the sake of discussion.

Throwable getCause( )
Throwable initCause(Throwable *causeExc*)

- The **getCause( )** method returns the exception that underlies the current exception.
- If there is no underlying exception, null is returned.
- The **initCause( )** method **associates causeExc** with the invoking exception and returns a reference to the exception.
- Thus, you can associate a cause with an exception after the exception has been created.
- However, the cause exception can be set only once.
- Thus, you can call **initCause( )** only once for each exception object.
- Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause( ).**
- In general**, initCause( ) is used to set a cause for legacy exception classes that don't support the two additional constructors described earlier.**
- Here is an example that illustrates the mechanics of handling chained exceptions:

```
// Demonstrate exception chaining.
class ChainExcDemo {
  static void demoproc() {

    // create an exception
    NullPointerException e =
      new NullPointerException("top layer");

    // add a cause
    e.initCause(new ArithmeticException("cause"));

    throw e;
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      // display top level exception
      System.out.println("Caught: " + e);

      // display cause exception
      System.out.println("Original cause: " +
                         e.getCause());
    }
  }
}
```

The output from the program is shown here:

```
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause
```

In this example, the top-level exception is **NullPointerException**. To it is added a cause exception, **ArithmeticException**. When the exception is thrown out of **demoproc( )**, it is caught by **main( )**. There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling **getCause( )**.

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design.

Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
  // Throw an exception out of the method.
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("procA's finally");
    }
  }

  // Return from within a try block.
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }
```