Object Oriented Programming with JAVA

Module 4

Packages and Exceptions

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.

Packages:

Generally, a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.

Packages are containers for classes.

They are used to keep the class name space compartmentalized.

Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

The package is both a naming and a visibility control mechanism.

You can define classes inside a package that are not accessible by code outside that package.

You can also define class members that are exposed only to other members of the same package.

This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package:

To create a package , simply include a package command as the first statement in a Java source file.

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;

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. Typically, 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.

More than one file can include the same package statement.

You cannot rename a package without renaming the directory in which the classes are stored.

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 a.b.c; needs to be stored in a\b\c in a Windows environment.

An Example Java program that uses 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();
  }
}
```

Finding Packages and CLASSPATH

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.

Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.

Third, you can use the **-classpath** option with java and javac to specify the path to your classes.

For example, consider the following package specification:

package mypack;

In order for a program to find **mypack**, the program can be executed from a directory immediately above **mypack**, or the **CLASSPATH** must be set to include the path to **mypack**,

or the **-classpath** option must specify the path to **mypack** when the program is run via java.

When the second two options are used, the class path must not include **mypack**, itself.

It must simply specify the path to **mypack**.

For example, in a Windows environment, if the path to **mypack** is **C:\MyPrograms\Java\mypack** then the class path to **mypack** is **C:\MyPrograms\Java**

Packages and Member Access:

Packages add another dimension to access control. 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.

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 in subclasses

	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

Anything declared **public** can be accessed from different classes and different packages.

Anything declared **private** cannot be seen outside of its class.

When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.

If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

An Access Example

Protection.java

```
package p1;

public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;

  public Protection() {
    System.out.println("base constructor");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file Derived.java:

```
class Derived extends Protection {
  Derived() {
    System.out.println("derived constructor");
    System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file SamePackage.java:

```
class SamePackage {
  SamePackage() {
    Protection p = new Protection();
    System.out.println("same package constructor");
    System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);

    System.out.println("n_pro = " + p.n_pro);
```

```
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

This is file Protection2.java:

```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("derived other package constructor");

//  class or package only
//  System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file OtherPackage.java:

```
class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");

//  class or package only
//  System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);

//  class, subclass or package only
//  System.out.println("n_pro = " + p.n_pro);

    System.out.println("n_pub = " + p.n_pub);
  }
}
```

If you want to try these two packages, here are two test files you can use. The one for **package p1** is shown here:

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
  public static void main(String[] args) {
    Protection ob1 = new Protection();
    Derived ob2 = new Derived();
    SamePackage ob3 = new SamePackage();
```

```
  }
}
```

The test file for p2 is shown next:

```
// Demo package p2.
package p2;

// Instantiate the various classes in p2.
public class Demo {
  public static void main(String[] args) {
    Protection2 ob1 = new Protection2();
    OtherPackage ob2 = new OtherPackage();
  }
}
```

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.

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:

import pkg1 [.pkg2].(classname | *);

import java.util.Date;

import java.io.*;

import java.lang.*;

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 {
}
```

When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.

For example, if you want the **Balance** class of the **package mypack** shown earlier to be available as a stand-alone class for general use outside of **mypack**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package mypack;

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()
  }
}
```

Exception Handling

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.

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.

That method may choose to handle the exception itself, or pass it on.

Either way, at some point, the exception is caught and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

Manually generated exceptions are typically used to report some error condition to the caller of a method.

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
}
```

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.

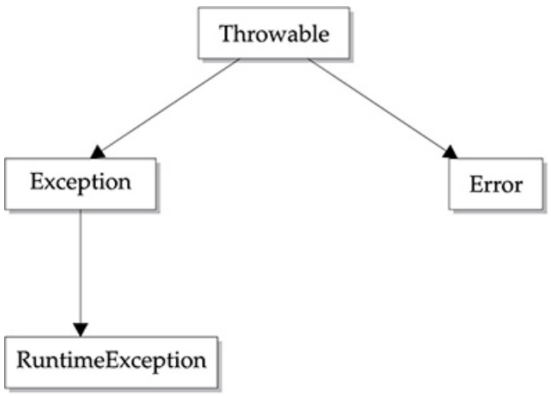
To manually throw an exception, use the keyword **throw**.

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.

ExceptionType is the type of exception that has occurred.

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.

Uncaught Exceptions:

If we do not supplied any exception handlers of our own, the exception is caught by the default handler provided by the Java run-time system.

Consider the following program for example:

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

Here is the exception generated when this example is executed:

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

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.

This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

In this example, since 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.

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.

To 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.

Consider the following example:

```
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.
```

Once an exception is thrown, program control transfers out of the **try** block into the **catch** block.

catch is not "called," so execution never "returns" to the **try** block from a catch.

Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try / catch** mechanism.

A **try** and its **catch** statement form a unit.

The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.

A **catch** statement cannot **catch** an exception thrown by another **try** statement.

The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.)

You cannot use **try** on a single statement.

The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
  public static void main(String[] args) {
    int a=0, b=0, c=0;
    Random r = new Random();

    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    }
  }
}
```

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,

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

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

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.**

```
// 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.");
  }
}
```

```
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:
Index 42 out of bounds for length 1
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.

```
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);
    }
  }
}
```

throw:

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.

```
// 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);
    }
  }
}
```

Output:

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

The keyword **new** is used to construct an instance of **an Exception** .

Example:

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

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( ).

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.

Example:

```
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.

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

```
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");
    }
  }

  // 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();
  }
}
```

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:

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples.

The most general of these exceptions are subclasses of the standard type **RuntimeException**. These exceptions need not be included in any method's throws list. 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 the following Table.

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.
IllegalCallerException	A method cannot be legally executed by the calling code.
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.
LayerInstantiationException	A module layer cannot be created.
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.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

The Table shown below 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
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.

Creating Your Own Exception Subclasses:

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.

To create your own exception: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

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.

Exception defines four public constructors:

- Exception( )
- Exception(String msg)
- Throwable(Throwable causeExc)
- Throwable(String msg, Throwable causeExc)

The first form creates an exception that has no description.

The second form lets you specify a description of the exception.

The Thord form, **causeExc** is the exception that causes the current exception.That is, **causeExc** is the underlying reason that an exception occurred.

The fourth form allows you to specify a description at the same time that you specify a cause exception.

Although specifying a description when an exception is created is often useful, sometimes it is better to override **toString**( ). Here's why: The version of **toString**( ) defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString**( ), you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

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:

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 two constructors are :

- **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**( ).

- **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. This means that 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:

```
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.

MODULE 4 QUESTION BANK

Packages:

1. What is Package in Java?
2. Illustrate the general form of the package statement in Java with a suitable programming example.
3. Illustrate Finding Packages and CLASSPATH.
4. Explain how packages can be used for controlling member access in Java.
5. Write a Java program to demonstrate the access control using package.
6. Explain the general form of the import statement in Java.
7. Write a Java program to demonstrate importing of packages.

Exception Handling:

1. Define Exception. Why is exception handling is required?
2. Explain the general form of exception-handling block in Java.
3. Illustrate the Java Exception types.
4. Explain Uncaught Exceptions in Java with a suitable programming example.
5. Illustrate how try and catch block is used to handle exception in Java with a suitable programming example.
6. Explain how to Display the Description of an Exception in Java.
7. Explain how can Multiple catch Clauses be used in Java with a suitable programming example.
8. Illustrate Nested try Statements in Java with a suitable programming example.
9. Illustrate throw in Java with a suitable programming example.
10. Illustrate throws in Java with a suitable programming example.
11. Illustrate finally in Java with a suitable programming example.
12. Explain Java's Built-in Exceptions.
13. Explain how can we create our own Exception Subclasses in Java with a suitable programming example.
14. Illustrate Chained Exceptions in Java with a suitable programming example.