Module -3

Packages

Packages:

- Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package**.**
- 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 only exposed to other members of the same package.

Defining a Package:

- To create a package is quite easy: 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 noname.
- 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.

- Use 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 :

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;

Finding Packages and CLASSPATH:

- Packages are mirrored by directories.
- The Java run-time system know where to look for packages that user create
  - 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 specifythe path to your classes.
- For example, consider the following package specification:**package MyPack**

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

A Short Package Example:

```
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[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**.
- Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory.
- Then, try executing the **AccountBalance** class, using the following command line:

  java MyPack.AccountBalance

Packages and Member Access:

- Packages add another dimension to access control.
- Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages 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 categoriesof 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 specifiers, **private**, **public**, and **protected**, provide a variety of waysto produce the many levels of access required by these categories.
- Table 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

- 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:

- This example has two packages and five classes. The classes for the two different packages need to be stored in directories named after their respective packages in this case, p1 and p2.

- The source for the first package defines three classes: Protection, Derived, and SamePackage. The first class defines four int variables in each of the legal protection modes.
- The variable n is declared with the default protection, n_pri is private, n_pro is protected, and n_pub is public.
- The second class, Derived, is a subclass of Protection in the same package, p1. This grants Derived access to every variable in Protection except for n_pri, the private one.
- The third class, SamePackage, is not a subclass of Protection, but is in the same package and also has access to all but n_pri.

This is file **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);
   }
}
```

```
package p1;

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

// class only
// System.out.println("n_pri = "4 + n_pri);

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

This is file **SamePackage.java**:

```
package p1;

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);
   }
}
```

```
public class Demo
{
public static void main(String args[])
 {
    Protection ob1 = new Protection();
    Derived ob2 = new Derived();
    Same Package ob3 = new Same Package ();
  }
 }
```

- Following is the source code for the other package, p2. The two classes defined in p2 cover the other two conditions that are affected by access control.
- The first class, Protection2, is a subclass of p1.Protection. This grants access to all of p1.Protection's variables except for n_pri (because it is private) and n, the variable declared with the default protection.
- The default only allows access from within the class or the package, not extra-package subclasses.
- Finally, the class OtherPackage has access to only one variable, n_pub, which was declared public.

Protection2.java:

```
package p2;
class Protection2 extends pl. 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);
  }
}
```

OtherPackage.java:

```
package p2;
class Other Package {
Other Package ( )
{
    pl. Protection p = new pl. Protection();
    System.out.println("other package constructor");
    // class or package only
   // System.out.println("n"+n);
   // class only
   System.out.println("n_pub =" n_pub);
  }
}
public class Demo {
 public static void main(String args[ ])
  {
    Protection2 ob1 = new Protection2 ();
    Other Package ob2 = new OtherPackage ();
   }
```

Importing Packages:

- Given that packages exist, it is easy to see why all of the built-in Java classes are stored in packages.
- There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package.
- 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.

This is the general form of the **import** statement:

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

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 (**.**).

- There is no practical limit on the depth of a package hierarchy, except that imposed bythe file system.

- Finally, 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.*;
```

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the **java** package called **java.lang**. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

- If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes.
- In that case, you will get a compile-time error and have to explicitly name the class specifying its package.
- 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.

- Example program:

```
package mypack;
public class Balance
{
    String name;
    double bal;
    Balance(String n, double b) {
       name = n;
```

```
        bal = b;
      }
 public void show( )
 {
  System.out.println("name and salary is ",  +name" "+bal);
 }
 }
Import mypack;
class AccountBalance
{
   public static void main(String args[])
   {
     Balance cur = new Balance(current,10000);
     cur.show();
   }
}
```

Output: name is : current 10000.

Chapter-2: Exceptions

Exception-Handling Fundamentals

- 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.
- In computer languages that do not support exception handling, errors must be checked and handled manually through the use of error codes, and so on.
- Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.
- 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 user 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.

- Java exception handling is managed via **five** keywords: **try, catch, throw, throws, and finally.**
- Program statements that user want to monitor for exceptions are contained within a try block.
- If an exception occurs within the try block, it is thrown. Users 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 must be executed after a try block completion is put in a finally block.
- The general form of an exception-handling block:

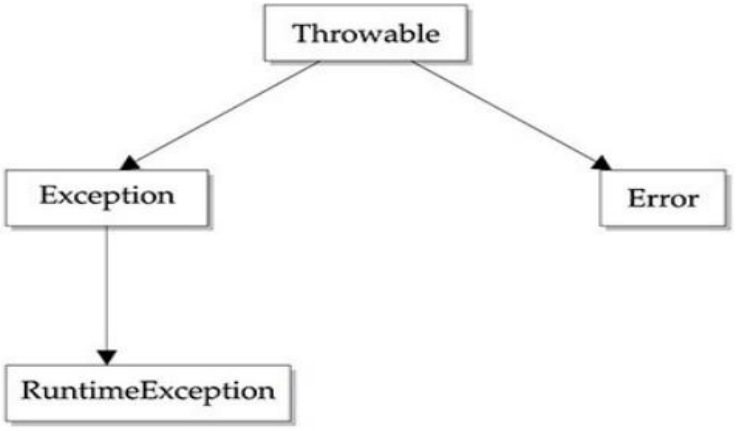
```
try {
    // block of code to monitor for errors
}

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:

- 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.
- 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. Ex: Stack overflow.
- The top-level exception hierarchy is shown here:



Uncaught Exceptions:

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

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    Exc1.subroutine();
  }
}
```

- 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, 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 Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

- Here, the class name Exc0, the method name main, the filename, Exc0.java; and the line number, 4, the bottom of the stack is main's line 7, which is the call to subroutine( ), which caused the exception at line 4. All are included in the simple stack trace.

- The type of exception thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error happened.
- The stack trace will always show the sequence of method invocations that led up to the error.

Using try and catch:

- Whenever, User want to handle an exception manually then try catch block will be used.
- 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.
- For Example, the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error:

```
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.
```

- The call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.
- Put differently, catch is not “called,” so execution never “returns” to the try block from a catch. Thus, the line "This will not be printed." is not displayed.
- 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 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.

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.
- User 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

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:

```
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 user 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.
- 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:
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.
- Here is an example that uses nested try statements:

```
class NestTry {
 public static void main(String args[ ]) {
 try
 {
        int a = args.length;
        int b = 42 / a;
        System.out.println("a = " + a);
        try
        {
            if(a==2) {
            int c[] = { 1 };
```

```
            c[42] = 99;
          }
      }
    catch(ArrayIndexOutOfBoundsException e)
    {
       System.out.println("Array index out-of-bounds: " + e);
    }
    catch(ArithmeticException e)
    {
      System.out.println("Divide by 0: " + e);
    }
    }
}
```

- This program nests one try block within another. The program works as follows.
- When you execute the program with no command line arguments, a divide-by-zero exception is generated by the outer try block.
- Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block.
- Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled.
- If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.
- Here are sample runs that illustrate each case:

  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:

Index 42 out of bounds for length 1

- Nesting of try statements can occur in less obvious ways when method calls are involved.
- For example, you can enclose a call to a method within a try block. Inside that method is another try statement.
- In this case, the try within the method is still nested inside the outer try block, which calls the method.

throw:

- It is possible for user program to throw an exception explicitly, using the throw statement.
- The general form of throw is :

  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.
- Sample program that creates and throws an exception:

```
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);
    }
  }
}
```

- The handler that catches the exception rethrows it to the outer handler.

- 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:

  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:

- 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.
- 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.
- First, you need to declare that throwOne( ) throws IllegalAccessException. Second, main( ) must define a try / catch statement that catches this exception.

```
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);
    }
  }
}
```

- Output:

  inside throwOne

  caught java.lang.IllegalAccessException: demo

finally:

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely.
- This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exceptionhandling mechanism.
- The finally keyword is designed to address this contingency.
- 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.
- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.
- Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:

```
class FinallyDemo {
 static void procA( ) {
 try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
```

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

Output:

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

- In this example, procA( ) prematurely breaks out of the try by throwing an exception.
- However, the finally block is still executed. If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

Java's Built-in Exceptions

- Inside the standard package java.lang, Java defines several exception classes.
- 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 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.
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.

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

- Although Java's built-in exceptions handle most common errors, user will probably want to create your own exception types to handle situations specific to your applications.
- This is done by defining a subclass of Exception (which is, a subclass of Throwable).
- The Exception class does not define any methods of its own. It does, inherit those methods provided by Throwable.
- Thus, all exceptions, including those that user create, have the methods defined by Throwable available to them. They are shown in Table 10-3.

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**

- Exception defines **four public constructors**:Two support chained exceptions.The other two are : Exception( ), Exception(String msg)
- The first form creates an exception that has no description. The second form lets user specify a description of the exception.
- Sometimes it is override using toString( ): 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 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.

```
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);
    }
  }
}
```

Output:

Called compute(1)

Normal exit

Called compute(20)

Caught MyException[20]

- This example defines a subclass of Exception called MyException. This subclass 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:

Chained Exceptions

- The chained exception feature allows user 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.
    - 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 user can call initCause( ) only once for each exception object.
- initCause( ) is used to set a cause for legacy exception classes that don't support the two additional constructors.
- Here is an example that illustrates the mechanics of handling chained exceptions:

```
class ChainExcDemo
 {
   Static void demoproc( ){
        NullPointerException ex = new NullPointerException ("top layer");
        ex.initCause (new ArithmeticException("cause"));
        throw ex;
    public static void main(String args[])
   {
       try{
```

```
        demoproc( );
       }
      catch (NullPointerException ex)
      {
         System.out.println("Caught: " + ex);
         System.out.println("Original cause: e.getCause());
      }
    }
}
```

Output:

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.