PACKAGES

Package is the name given to group of classes. It acts as a container for classes. A package consists of group of classes putting together at one place and whole group is defined by a unique name.

Consider a situation of two families living near by, both families have two persons with same name. So if we want to call them, then as they belong to different families we can refer those persons by considering their families. Although they have same name but there will no name collision as both of them are referred in context with their families. If we would have not considered their families then it would have created problem regarding which person is being referred.

Same goes for classes. If two classes have same then name it will create a problem for compiler as it will create confusion that which class user is referring to. So we define classes under different packages. classes under different packages can have same name, but inside same package classes will have unique names. So if a class is defined under package, then it is referred in reference to its package. There are two types of packages:

→ **System packages** – Packages already defined in Java

→ **User defined packages** – Packages defined by the user.

System Packages

As there are inbuilt methods, Java also provides inbuilt packages which contain lots of classes and interfaces. The classes inside these packages are already defined and you can use them by importing relevant package in your program.

Some of the commonly used Java packages :

→ lang

→ util

→ io

→ awt

→ net

→ applet

➢ **lang** package includes classes for primitive types, strings, math functions, thread and exception. This package is imported by default. These classes are used by Java compiler.

Some of the classes that lang package includes, are :

- Boolean – wrapper class for boolean data type.
- Byte – wrapper class for byte data type.
- Double – wrapper class for double data type.
- Long– wrapper class for long data type.
- Float – wrapper class for float data type.

- Integer – wrapper class for integer data type.
- Math - class for mathematical functions.
- String – class for string functions.
- Thread – class for managing thread.

➢ **util** package includes classes for utility functions like vectors, hash tables, random numbers, date etc.

- Date – class for various date functions.
- Random – class for generating random numbers.
- Vector – class for creating vectors.
- Hashtable – class for hash tables.

➢ **io** package includes classes that are used to support input/output operations.

- DataInputStream – class used for input operation.
- DataOutputStream – class used for output operation.
- BufferedInputStream – class for buffered input stream.
- BufferedOutputStream – class for buffered output stream.
- InputStream – parent of all input stream of io package.
- PrintStream – output stream for displaying output.

➢ **awt** includes classes that are used for implementing graphical user interface.

- Button – class for pushbutton.
- Color – class for color types.
- Font – class for font types.
- Frame – for creating frame.
- Graphics – class for graphics.
- Label – class for labels.
- TextField – class for text field.

➢ **net** includes classes for networking.

- DatagramPacket – class for datagram packet.
- DatagramSocket – class for datagram socket.
- Socket – class for socket.
- URL – class for url.

applet includes **Applet** class for creating and implementing applets.

These packages are referred to as Java API packages.

USING SYSTEM PACKAGES

We can use packages in our program by importing them. A package is imported by using import keyword; as :

import package;

Now there are two ways to access classes of packages :

First way : We can import only that particular class that we want to use in our program.

General form :

import pkg.classname;

Example : We want to use button class of awt package then we will write as :

import java.awt.Button;

Second way : Instead of specifying a name, we write '*', *, it means that Java compiler should import all the packages.

General form : import pkg1.pkg2. *;

eg. :

import java.util. *;

import java.lang *;

The following program imports util package of Java and uses its Date class to display current date.

Program : Displaying date using util package.

```
import java.util.*;
class Abc
{
    public static void main(String args[])
    {
        Date d=new Date();
        System.out.print("Current date:");
        System.out.println(d);
    }
}
```

OUTPUT

Current date:Thu Jun 04 04:52:21 IST 2009

Here, in this program object of Date class is created. This will display current date in day:month:dd format.

Program : Displaying the sum using io package

```
import java.io.*;
class Display
{
public static void main(String args[]) throws IOException
    {

        DataInputStream in = new DataInputStream(System.in);
            String str;
        System.out.println("Enter a string");
          str=in.readLine();
        System.out.println("String is "+str);
        }
}
```

OUTPUT

Enter a string
surbhi
String is surbhi

USER DEFINED PACKAGES

As stated earlier, you can also create your own packages. To declare a package, you just need to include package keyword with a name; as:

package pckgname;

This is included as first statement. All classes that are defined inside this statement belong to the specified package. Here pckgname is any valid name given to package. If you include package statement then all classes defined below this statement belong to that very package, otherwise if the package statement is not included then all classes are put into default package without any name.

Packages are stored with the use of file system directories. A package statement only tells that what all classes belong to this packages, so any number of classes can include same package statement.

Example : package abc;

Here abc is the package name .class files for all classes defined under this package must be stored in a directory called abc.

You can even create sub packages means packages under packages then classes are referred in accordance with package hierarchy.

General form : package pkg1 [pkg2 [.pkg3]];

It means pkg3 is subpackage of pkg 2 and pkg2 is subpackage for pkg1.

Remember, once you had named a package and stored classes in it, you cannot rename it without renaming the directory in which the classes are stored.

Adding Classes to a Package

General form :

```
Package pckgname;
public class classname
{
    Body of class
}
```

Procedure :

1. Add package statement.
2. Include class definition that you want to put in package and declare it public.
3. Store this classname file into directory named as packagename.
4. Compile classname file. This will create a class file in the current directory. (Remember to keep your package and subpackage in current directory because by default Java uses current working directory as its starting point. Put the class file into the package directory and then execute it by writing :

   java packagename.class name

Similarly, you can add a non – Public class to the package.

You can create only one public class in one .java file. So if you want to create package with more than one public class then you should follow these steps:

1. Create a folder with name same as that of package.
2. Create classes separately i.e. one file should contain one public class. Each file should include package statement at the top. As :

   package packagename;

3. Compile each file.

Program : Adding one class to a package

```
package abc;
 class Name
{
    public void show( )
    {
        System.out.println ("Your name is Sakshi");
    }
}
class Display
{
    public static void main (String args[ ])
    {
        Name ob = new Name( );
        ob.show( );
    }
}
```

OUTPUT

Your name is Sakshi

Program : Adding two classes to a package

```
package abc;
 class Name
    {
        public void show( )
        {
            System.out.println ("Your name is Sakshi");
        }
    }
class Name1
    {
        public void show1( )
        {
            System.out.println ("Your name is Surbhi");
        }
    }

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

```
        {
            Name ob = new Name( );
                                Name1 ob1 = new Name1( );
            ob.show( );
            ob1.show1( );
        }
    }
```

OUTPUT

Your name is Sakshi

Your name is Surbhi

Accessing and using package

As stated earlier, you can access a package using import statement.

Program : Showing how to access a user defined package

```
package xyz;
    public class Value
    {
        public void show( )
        {
            System.out.println ("Value of a = 8");
        }
    }
```

```
import xyz.Value;                //accessing user defined package xyz
class Display
    {
        public static void main (String args[ ])
        {
            Value ob = new Value( );
            ob.show( );
              }
        }
```

OUTPUT

Value of a = 8

Program : Importing two packages.

Step 1 : Creating a package xyz

```
package xyz;
    public class Value
    {
        public void show( )
```

```
    {
            int a = 10;
        System.out.println ("Value of a = "+a);
    }
}
```

Step 2 : Creating a package xyz1

```
package xyz1;
    public class Value1
    {
        public void show1( )
        {
                int b = 50;
            System.out.println ("Value of b = "+b);
        }
            public void sum1( )
              {
                int c = 46, n = 30;
                int sum=c+n;
            System.out.println("Sum = "+sum);
            }
}
```

Step 3 : Creating class that will import package xyz and xyz1

```
import xyz.Value;
import xyz1.*;
class Display
    {
        public static void main (String args[ ])
        {
            Value ob = new Value( );
                Value1 ob1 = new Value1( );
    ob.show( );
                ob1.show1( );
                ob1.sum1();
            }
        }
```

OUTPUT

Value of a = 10

Value of b= 50

Sum = 76

While importing multiple packages, it can be possible that a class that imports two packages contain classes with same name. So in order to import such type of packages you should explicitly define which class belong to which package. Let's write a program to understand this concept more clearly:

Program : Creating class importing two packages which contain classes with same name.

Step 1 : Creating a package abc

```
package abc;
public class Name
  {
    public void show( )
      {
      System.out.println("Name is ; Atul and Ishan");
      }
  }
```

Step 2 : Creating a package abc1

```
package abc1;
public class Name
  {
    public void show1( )
      {
      System.out.println("Name is : Purna");
      }
  }
```

Step 3 : Creating a class that import abc and abc1

```
import abc.*;
import abc1.*;
class Display
  {
   public static void main (String args[])
        {
              abc.Name n1 = new abc.Name( );
              abc1.Name n2 = new abc1.Name( );
              n1.show( );
              n2.show1( );
        }
  }
```

OUTPUT

Name is : Atul and Ishan

Name is : Purna

Program : Showing that you can extend an imported class

Step 1 : Creating a package xyz

```
package xyz;
    public class Value
    {
        public void show( )
        {
                int a = 10;
            int b = 20;
        }
    }
```

Step 2 : Creating a class that can extend Value

```
import xyz.*;
class Value1 extends Value
    {
        public void sum1( )
            {
                int sum = a +b;
            System.out.println ("Sum = " +sum);
            }
}

class Display
    {
    public static void main (String args[])
        {
            Value1 ob = new Value1();
            ob.sum1();
        }
    }
```

OUTPUT

Sum = 30

Program : Creating a package with multiple public classes:

Step 1 : Creating first public class

```
package abc;
public class First
    {
    int a = 20;
```

```
        int b = 30;
        public void product( )
            {
                int c = a * b;
                System.out.println("Product = "+c);
            }
    }
```

Step 2 : Creating second public class

```
package abc;
public class Second
    {
        int n = 15;
        int m = 30;
        public void divide( )
            {
                int i = m / n;
                System.out.println("Quotient = "+i);
            }
    }
```

Step 3 : Creating public class with main method

```
package abc;
public class Display
    {
public static void main ( String args[])
        {
            First ob = new First();
            Second ob1 = new Second();
            ob.product();
            ob1.divide();
        }
}
```

OUTPUT

Product = 600

Quotient = 2

ACCESS PROTECTION

As you know, Java provides four categories of visibility control for its class members :

→ Subclasses in the same package

CHAPTER

10

EXCEPTION HANDLING

Chapter Includes :

- INTRODUCTION
- EXCEPTIONS
- USING TRY & CATCH
- MULTIPLE CATCH CLAUSES
- FINALLY
- THROW
- THROWS

INTRODUCTION

When we write a program, then it is very common that we make mistakes. These mistakes cause errors in a program which lead to unexpected result. Errors are very harmful for a program. They can produce incorrect output, cause the program to terminate abruptly, or even sometimes they are so dangerous that they can cause a system to crash.

Errors are classified into 3 types:

1. Syntax error
2. Logical error
3. Run-time error

Syntax Errors

Every language has its own syntax for writing a program. So if we make any mistake in writing syntax, then errors are termed as syntax errors. All errors that we make in syntax of a program are detected at compile time. When we compile a file then compiler detects all errors and do not create .class file. .class file is created only when we fix all errors of a program. When we compile a program in Java then Java compiler displays a list that contains error description and line number where error has occured. So it provides us detail of errors in program. As it tells us about line also, so we can go to appropriate line, correct it and recompile the program again. It goes on giving us errors until we fix all the errors. When we have successfully removed all errors, only then compiler will create .class file of program. Most common syntax errors that we make are missing semicolons, use of undeclared variables, missing brackets between methods and classe, missing double quotes in string.

Logical Errors

Logical errors are the errors that occur due to use of incorrect logic in program. These errors are not detected by any one, but they produce incorrect output.

For example : We make a program that will be used to add two numbers. But instead of using + operator, we have used * operator. So when we will compile this program, it will be compiled, and will be executed also. But it will give us product of two numbers instead of sum. This is because we have used * operator while writing the program. Here it's our mistake; we have used incorrect logic due to which we are getting incorrect output.

So these types of errors are termed as logical errors.

Runtime Errors

Runtime errors are the errors that arise at runtime. They may produce wrong results due to wrong logic like division by zero, or may cause a program to terminate due to stack overflow. Most common run-time errors are division by zero, accessing an element that is out of the boundary of

EXCEPTIONS

An exception is a run-time error that has occurred in a piece of code. You can also define exception as an object that describes an error condition that arises in a code at run-time. When runtime error occurs, an object is created in order to represent that error (exception). After creation of exception object, it is thrown in the method that caused the error. In this way, exception is caught by the method and it displays an appropriate message for taking corrective actions.

A method may choose to handle the exception itself or pass it on. So if a method is not handling the exception then at some point exception is caught by some other way. This process is known as exception handling.

Exceptions can be generated by Java run-time system or they can be manually generated by our code.

Java run-time system generates exceptions when any error occurs that violates the rules of the Java language.

Manually generated exceptions are generated in order to tell caller of method about some error condition.

Exception handling is done in order to detect and report an error so that appropriate action can be taken.

So the process of exception handling can be breakup into these tasks:

1. Finding out the error.
2. To inform that an error has occurred
3. Receiving the error information.
4. Taking appropriate action.

This process is accomplished by using these five keywords:

- try
- catch
- throw
- throws
- finally

- **try** block contains section or segment of a program that we want should be monitored (checked) for any error.
- **catch** is used to catch exception that is thrown by try block in case of occurrence of error.
- **throw** is used when you want to throw an exception manually.
- **throws** clause is used to specify an exception that must be thrown out of a method.
- **finally** block contain code that must be executed before a method returns.

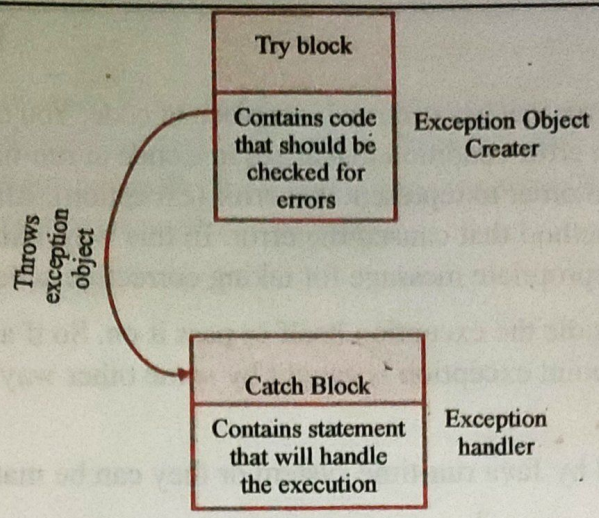
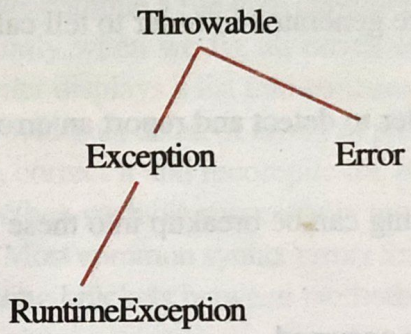


Figure : Exception handling mechanism

Let's have an overlook of classes used for exceptional handling.



All classes that are used in exceptions are sub-classes of throwable. Throwable is divided into subclasses Exception and error. Exception class is used for exceptional conditions that a user program should catch. Exception is also used when we create our own custom exception. Exception class has an important subclass named as RuntimeException. This class includes exceptions that are defined automatically for a program.

Then comes another branch of Throwable, i.e. Error class. This class defines exceptions that are not caught by program under normal conditions. These type of exceptions are used by Java run-time system.

Java's Built-in Exceptions

ArithmeticException	Arithmetic error, division by zero
ArrayIndexOutOfBoundsException	Array index is out of bounds
ClassCastException	Invalid cast
NumberFormatException	Invalid conversion of a string to a numeric format.

USING TRY & CATCH

Let's begin with a simple program that does not handle exceptions explicitly.

ram that does not handle exceptions explicitly.

```
lass Divide
  {
    public static void main (String args [])
      {
        int x, y, z;
        x = Integer.parseInt (args [0]);
        y = Integer.parseInt (args[1]);
        z = x / y;
        System.out.println ("Result =" +z);
      }
  }
```

ew after compiling, you will run this program. Suppose, you give command as :

Java Divide 40 0

Here we passed two arguments to the class Divide, 40 get stored in args[0] & 0 get stored in rgs[1].

It will create expression z = x/y as z = 40/0. So we are dividing 40 by 0. As soon as Java run-time system detects that this program is dividing a number by zero, it will create an exception and causes program to terminate. As you know once an exception has been thrown, it must be caught by an exception handler. But we are not using any exception handler in this program.

In cases like our program, Java run-time system provides default exception handler that catches the exception thrown by these types of programs. After catching the exception, default handler will display a string that describes the exception and prints a stack trace from the point at which the exception occurred. After displaying string it terminates the program.

So when above program will be executed, default handler will display a string like:

java.lang.ArithmeticException : / by zero
at Divide.main (Divide.java : 6)

So, this string is giving you detailed output explaining in which file, at which line and what error occurred.

Here error is /by zero, main is method name, Divide java is file name and line number is 6.

In this case, it is giving arithmetic exception so let's write a code so that our program will be able to handle the exception itself and program will not be terminated.

Using your own exception handler provides you with two benefits:

1. Error in the program can be fixed.
2. It prevents a program from automatically terminating.

So, you can simply put the code that you want to monitor for errors in try block. Then immediately after try block, include catch statements in order to catch an exception.

General form of try & catch

```
try
    {
        code to be monitored.
    }
    catch (Exception e) { }
```

Let's write previous program using try and catch in order to handle the exception

Program that shows use of try and catch.

```
class Divide
    {
        public static void main (String args[ ])
        { int x, y, z;
          x = Integer.parseInt (args [0]);
          y = Integer.parseInt (args [1]);
          z = 0;
          try
            {
                z = x/y;
                System.out.println ("Result =" +z);
            }
            catch (ArithmeticException a)
            {
                System.out.println ("Divisor cannot be zero");
            }
        }
    }
```

Now if we give Divide 40 0, then it will produce output as :

Divisior cannot be zero.

If we give input as Divide 40 10

then it will produce output as :

Result = 4

As you can see, when we are giving value of y as zero then statement written inside the try block is not displayed. It is because when an exception is thrown, control comes out of try block and statement written in catch clause gets displayed. Once statement written in catch clause gets executed, program control continues with the next line written after catch clause.

gram to that shows use of try and catch.

```
class Divide
    {
        public static void main (String args[ ])
        {
            int x, y, z;
        try
            {
                x = 4;
                y = 0;
                z = x/y;
            System.out.println ("Value of z ="+z);
        }
        catch (ArithmeticException e)
        {
            System.out.println ("Division by zero");
        }
    System.out.println ("You should not assign zero to y");
    }
}
```

OUTPUT

Division by zero

You should not assign zero to y.

Here after execution of statement written inside catch clause, statement written in the next line of program gets executed.

MULTIPLE CATCH CLAUSES

Consider the program again; here we are passing two arguments to main method. Suppose, by mistake we forget to pass second argument, i.e. instead of writing

Divide 40 20

We write

Divide 40

then what will happen ? Again program will terminate stating an exception known as ArrayOutOfBoundException. Surprised, how it can happen. You have already used exception handler in your program then what led to termination of program. Reason is, the exception handler that we used in our program can catch only arithmetic exception like division by zero but this time it gives another type of exception. So to handle this type of exception we need to create another exception handler that can handle this type of exception otherwise program will terminate.

So in order to catch this type of exception let's write this program again

Program to show use of multiple catch clause.

```
class Divide
  {
    public static void main (String args[])
      {
        int x, y, z;
        try
          {
            x = Integer.parseInt (args[0]);
            y = Integer.parseInt (args[1]);
            z = x/y;
            System.out.println ("Result=" +z);
          }
            catch(ArithmeticException a)
          {
          System.out.println ("Divisor cannot be zero");
          }
          catch(ArrayIndexOutOfBoundsException b)
          {
          System.out.println ("Two numbers are required");
          }
        }
      }
```

Input : Divide 40

Output : Two numbers are required.

So if we pass one argument, then control will come out of try block and check each catch clause in order to check which clause matches the exception type and will execute that catch clause. Other catch clauses will be bypassed.

Now program will not be terminated but produces output as:

Two numbers are required

One more situation to be considered is, if user gives input as :

Divide 40 x

Now again program will be terminated and gives an exception as NumberFormatException.

So we will write this program again as:

Program that can handle three types of exceptions.

```
class Divide
  {
  public static void main (String args[ ])
    {
      int x, y, z;
      try
```

```
        {
            x = Integer.parseInt(args[0]);
            y = Integer.parseInt(args[1]);
            z = x/y;
        System.out.println("Result=" +z);
        }
        catch (ArithmeticException a)
        {
            System.out.println ("Division by zero");
        }
        catch (ArrayIndexOutOfBoundsException a)
            {
        System.out.println ("Two no's are required");
            }
        catch (NumberFormatException a)
        {
            System.out.println ("Valid number should be entered");
        }
    }
}
```

Now on giving input as:

Divide 20 0

Output: Division by zero

Input – Divide 20

Two no's are required.

Input – Divide 20 x

Valid number should be entered.

Now if we want to catch all type of exceptions using one clause only, then we can write as:

```
catch (Exception e)
{
    System.out.println ("Unknown Error");
}
```

As Exception is parent class, so a reference of parent class can be used to point to child class. So this clause can handle all types of exceptions.

Program that can handle all types of exceptions.

```
class Divide
    {
        public static void main (String args[ ])
        {
            int x, y, z;
            try
```

```
        {
            x = Integer.parseInt (args[0]);
            y = Integer.parseInt (args[1]);
            z = x/y;
            System.out.println ("Result =" +z);
        }
        catch (Exception e)
        {
            System.out.println ("Unknown error");
        }
    }
}
```

Input Divide 40 0
Output Unknown Error
Input Divide 40 x
Output Unknown Error

So this program can handle all types of exceptions. It will produce output unknown for all types of exceptions.

Now if you want to print name of exception, then you can write as :

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

So this code will give name of exception.

Program that gives you name of exception.

```
class Divide
    {
        public static void main (String args[ ])
        {
            int x, y, z;
            try
            {
                x = Integer.parseInt (args[0]);
                y = Integer.parseInt (args[1]);
                z = x/y;
                System.out.println ("Result =" +z);
            }
            catch (Exception e)
            {
                System.out.println (e);
            }
        }
    }
```

Output
java.lang.ArithmeticException: / by zero

FINALLY

Exceptions can also be handled by using finally statement. Code written inside finally will be executed in both cases, in case exception is caught by catch statement, and in the case exception is not caught by catch clause. Finally block is executed after try/catch block but before any statement that followed try/catch statement.

When we define a code inside finally, then this code will be executed, regardless of the fact that exception has been thrown or not.

This clause is optional, but try block must contain a single catch or finally clause. If finally block is associated with try then its statement is executed based upon the conclusion of try block.

General form:

```
try
    {
    }
    catch (Exception e)
        {    }
    finally
    {
    }
                    OR
try
    {
    }
    finally
    {
    }
```

Program that includes finally only

```
class Divide
  {
    public static void main (String args[ ])
    {
        int x, y, z;
        try
        {
            x = Integer.parseInt (args[0]);
            y = Integer.parseInt (args[1]);
            z = x/y;
            System.out.println ("Result =" +z);
        }
```

```
            finally
              {
              System.out.println (" Input valid numbers");
              }
        }
    }
```

Output

Input : java Divide 20 0

Output

Input valid numbers

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Divide.main(Divide.java:10)

Input : java Divide 20 10

Output

Result =2

Input valid numbers

Program that includes catch and finally

```
class Divide
  {
      public static void main (String args[ ])
      {
          int x, y, z;
          try
          {
              x = Integer.parseInt (args[0]);
              y = Integer.parseInt (args[1]);
              z = x/y;
              System.out.println ("Result =" +z);
          }
          catch (Exception e)
          {
              System.out.println (e);
          }
              finally
                {
                System.out.println (" Input valid numbers");
                }
      }
  }
```

Output

Input: java Divide 20 0

Output

java.lang.ArithmeticException: / by zero

Input valid numbers

THROW

This is used to explicitly raise an exception.

Program that shows use of throw.

```
class Extest
{
    public static void main (String args[ ])
    {
        int x, y, z;
        x=10;
        y=20;
        z =x+y;
        if(x==y)
            throw (new ArithmeticException());
        System.out.println ("Result=" +z);
    }
}
```

Output

Result = 30

Creating own Exception class

You can create your own Exception class by extending Exception class.

General form:

```
class MyException extends Exception
{
    Create your own method here
}
```

THROWS

It is used to warn the caller of method, if method is capable of causing an exception that it can not handle.

General form

```
type methodname (parameter_list) throws exception list
{
    // body of method
}
```

Here exception list is a list of exception that a method can throw separated by comma.

for example

```
void abc( ) throws MyException
{
    throw (new Myexception( ));
}
```

Security mechanism of Java warns the caller about possible exceptions.