Ch 4 :Exception Handling

Introduction

- Exception is an abnormal condition that arises when executing a program.
- Java:
  1) provides syntactic mechanisms to signal, detect and handle errors
  2) ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
  3) brings run-time error management into object-oriented programming
- An exception is an object that describes an exceptional condition (error) that has occurred when executing a program.
- Exception handling involves the following:
  1) when an error occurs, an object (exception) representing this error is created and thrown in the method that caused the error
  2) that method may choose to handle the exception itself or passes it on
  3) either way, at some point, the exception is caught and processed

Exception Sources

- Exceptions can be:
  1) generated by the Java run-time system-
     Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
  2) manually generated by programmer's code

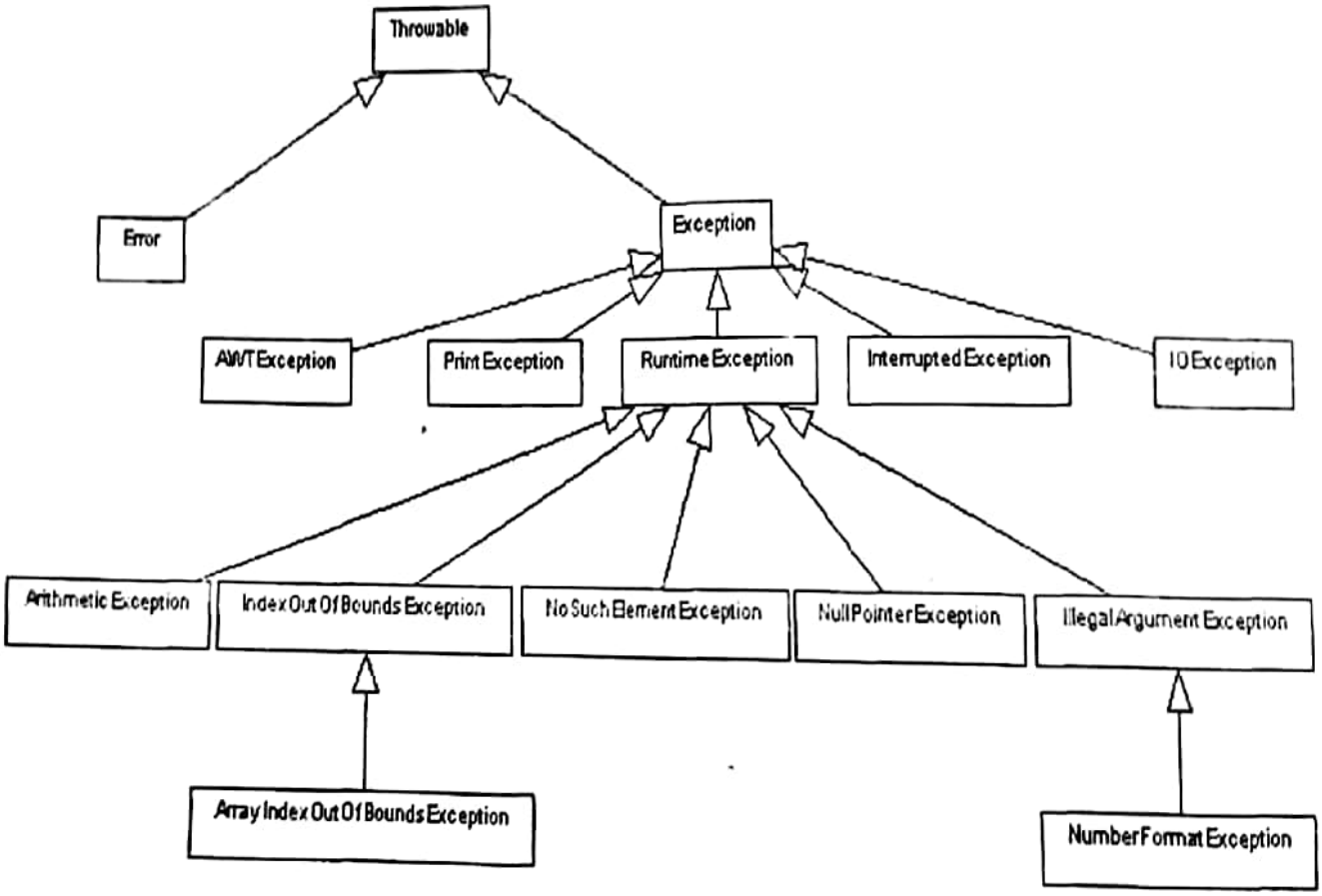
Exception Hierarchy

- All exceptions are sub-classes of the build-in class Throwable.
- Throwable contains two immediate sub-classes:
- 1) Exception – exceptional conditions that user programs should catch.
  The class includes:
  a) RuntimeException: automatically defined for user programs and include division by zero, invalid array indexing, etc.
  b) user defined exception classes

- 2) Error – defines exceptions that are not expected to be caught under normal circumstances by user program.

Exceptions of type **Error**

- Used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.



Uncaught Exception

- What happens when exceptions are not handled?

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

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and throws this object.
- This will cause the execution of Exc0 to stop – once an exception has been thrown it must be caught by an exception handler and dealt with.

Default Exception Handler

- As no exception handler provided, the exception is caught by the default handler provided by the Java run-time system.
- This default handler:

  1) displays a string describing the exception,

  2) prints the stack trace from the point where the exception occurred

  3) terminates the program

  java.lang.ArithmeticException: / by zero

  at Exc0.main(Exc0.java:4)

- Any exception not caught by the user program is ultimately processed by the default handler.

Stack Trace Display

- The stack trace displayed by the default error handler shows the sequence of method invocations that led up to the error.

Types of exceptions

- Checked exceptions –
- Inability to acquire system resources (such as insufficient memory, file does not exist)
- Java checks at compile time that some mechanism is explicitly in place to receive and process an exception object that may be created during runtime due to one of these exceptions occurring.
- Unchecked exceptions –
- exceptions that occur because of the user entering bad data, or failing to enter data at all.
- Unchecked exceptions can be avoided by writing more robust code that protects against bad input values. Java does not check at compile time to ensure that there is a mechanism in place to handle such errors.
- It is often preferred to use Java's exception handling capabilities to handle bad user input rather than trying to avoid such circumstances by providing user-input validation in the code.

Exception Handling Constructs

- Five constructs are used in exception handling:
  1) try – a block surrounding program statements to be monitored for exceptions
  2) catch – together with try, catches specific kinds of exceptions and handles them in some way
  3) finally – specifies any code that must be executed whether or not an exception occurs
  4) throw – used to throw a specific exception from the program
  5) throws – specifies which exceptions a given method can throw

Exception-Handling Block

General form:

```
try { ... }
catch(ExceptionType1 exOb1) { ... }
catch(ExceptionType2 exOb2) { ... }
...
finally { ... }
```

where:

1) try { ... } is the block of code to monitor for exceptions
2) catch(Exception ex) { ... } is exception handler for the exception ExceptionType
3) finally { ... } is the block of code to execute before the try block ends

Own Exception Handling

Default exception handling is basically useful for debugging.

Normally, we want to handle exceptions ourselves because:

1) if we detected the error, we can try to fix it
2) we prevent the program from automatically terminating

Exception handling is done through the try and catch block.

Multiple Catch Clauses

When more than one exception can be raised by a single piece of code, several catch clauses

can be used with one try block:

1) each catch catches a different kind of exception
2) when an exception is thrown, the first one whose type matches that of the exception is executed
3) after one catch executes, the other are bypassed and the execution continues after the try/catch block

Example: Multiple Catch

- Two different exception types are possible in the following code: division by zero and array index out of bound:

```
class MultiCatch {
    public static void main(String args[]) {
      try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[20] = 99;
      }
```

- Both exceptions can be caught by the following catch clauses:

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

Order of Multiple Catch Clauses

Order is important:

1) catch clauses are inspected top-down

2) a clause using a super-class will catch all sub-class exceptions

Therefore, specific exceptions should appear before more general ones.

In particular, exception sub-classes must appear before super-classes.

Throwing Exceptions

So far, we were only catching the exceptions thrown by the Java system.

In fact, a user program may throw an exception explicitly:

```
throw ThrowableInstance;
```

ThrowableInstance must be an object of type Throwable or its subclass.

Once an exception is thrown by:

throw ThrowableInstance;

1) the flow of control stops immediately

2) the nearest enclosing try statement is inspected if it has a catch statement that matches the type of exception:

   1) if one exists, control is transferred to that statement

   2) otherwise, the next enclosing try statement is examined

3) if no enclosing try statement has a matching catch clause, the default exception handler halts the program and prints the stack trace.

Creating Exceptions

- Two ways to obtain a Throwable instance:

1) creating one with the new operator-

All Java built-in exceptions have at least two constructors: one without parameters and another with one String parameter:

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

2) using a parameter of the catch clause-

```
try { ... } catch(Throwable e) { ... e ... }
```

Example: throw

- The method demoproc() throws a NullPointerException exception and it is caught in next line and it is then rethrown:

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

- The main method calls demoproc within the try block which catches and handles the NullPointerException 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
```

throws Declaration

If a method is capable of causing an exception that it does not handle, it must specify this behaviour by the throws clause in its declaration:

```
type name(parameter-list) throws exception-list {
    ...
    }
```

where exception-list is a comma-separated list of all types of exceptions that a method might throw.

All exceptions must be listed except Error and RuntimeException or any of their subclasses, otherwise a compile-time error occurs.

Example: throws

The throwOne method throws an exception that it does not catch, nor declares it within the throws clause.

```
class ThrowsDemo {
        static void throwOne() {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
          }
          public static void main(String args[]) {
                throwOne();
        }
}
```

- Therefore this program does not compile.
- Corrected program: throwOne lists exception, main catches it:

```
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 of the code-

Inside throwOne.

Caught java.lang. IllegalAccessException:demo

User-Defined Exceptions

- Although several exception classes already exist in the java.lang package, the built-in exception classes are not enough to cover all possible types of exception that may occur. Hence, it is very likelv that user wants to create his own exception.
- For creating your own exception, you have to create a class that extends the **RuntimeException** (or Exception) class. Then, it is up to you to customize the class according to the problem you are solving.
- Members and constructors may be added to your exception class.
- User-defined exceptions allow the programmer to handle errors in the application with customized responses.
- Creating these functions makes an application more easily understood and user friendly.

Instructions

1. A user-defined exception is an extension of the internal exception class. In this example, the exception is set up to test a negative age/number in the application. The following code sets up the class.

```
public class NegativeAgeException extends
        Exception {
}
```

2. Set up the constructor. The constructor takes a variable that is used to tell the user that the number is incorrect. This can be any variable in your application that causes the exception.

```
private int age;
public NegativeAgeException(int age){
        this. age = age;
}
```

3. Create the function that returns the error to the user. Usability in applications includes clear responses when there is an error. This code sets up the response to the user to let him know the number entered is incorrect. This is added to the exception class in steps 1 and 2.

```
public String toString() {
        return "Age cannot be negative" + " " +age;
    }
```

4. Test the exception class. When there is an error, the main code needs to throw an exception. In this example, if the user enters a negative number, the NegativeAgeException " exception function is called. This code sets up the test class.

```
public class TestException {
    }
```

5. Create the code to throw an exception. This example calls a function that returns a negative number that subsequently calls the " NegativeAgeException " exception.

```
public class NegativeAgeException extends Exception {
    private int age;
    public NegativeAgeException(int age)    {
        this.age = age;
      }

    public String toString()  {
        return "Age cannot be negative" + " " +age;
   }
  }
public class TestException{
  public static void main(String[] args) throws Exception{
            int age = getAge();
            if (age < 0){
                throw new NegativeAgeException(age);
            } else
```

```
        {
            System.out.println("Age entered  is " + age);
          }   }
    static int getAge(){
          return -7,
       }
}
```

Finally

- When an exception is thrown:
  1) the execution of a method is changed
  2) the method may even return prematurely.
- This may be a problem is many situations.
- For instance, if a method opens a file on entry and closes on exit; exception handling shoulu not bypass the proper closure of the file.
- The finally block is used to address this problem.
- The try/catch block requires at least one catch or finally clause, although both are optional:

```
try { ... }
catch(Exception1 ex1) { ... } ...
finally { ... }
```

- Executed after try/catch whether or not the exception is thrown.
- Any time a method is to return to a caller from inside the try/catch block via:
  1) uncaught exception or
  2) explicit return

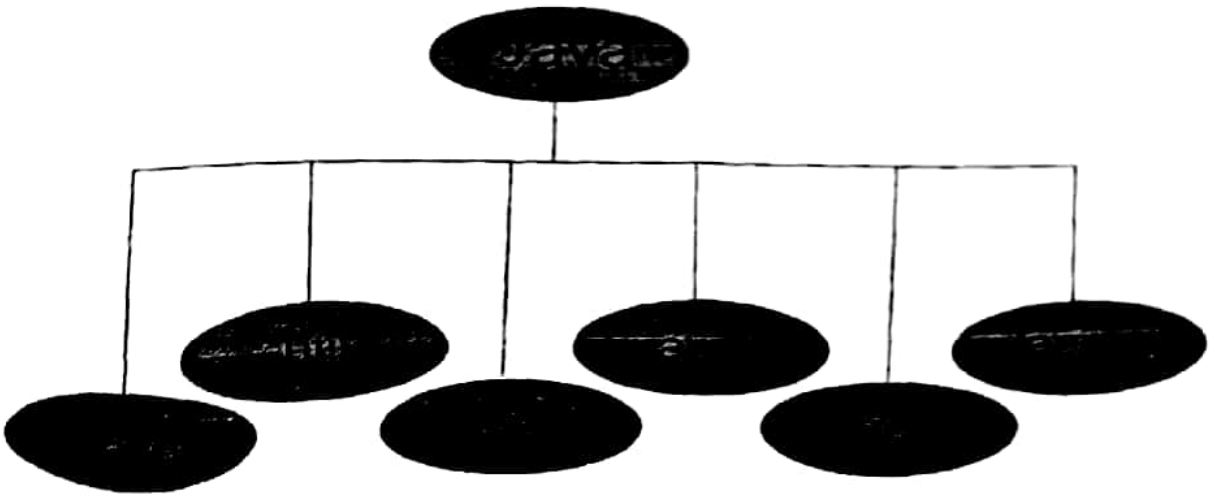
the finally clause is executed just before the method returns.

Packages

Introduction

- A package contains a group of classes, organized together under a single *namespace*. A **namespace** is an abstract container or environment created to hold a logical grouping of unique identifiers or symbols (i.e., names).
- For example, there's a utility library that's part of the standard Java distribution, organized under the namespace java.util. One of the classes in java.util is called Vector. One way to use an Vector is to specify the full name java.util.Vector.
- Reusability of code through Inheritance and Interfaces
- Reusability of code also through packages or class libraries
  - Classes from other programs can be used without copying them into program under development.
- The benefits of using Packages:
  1) The classes contained in the packages of other programs can be easily used.
  2) In Packages, classes can be unique compared with the classes in the other Packages.
  3) Packages provide the way to 'hide' the classes, thus preventing other programs or packages from accessing these classes.
- Two categories of packages-
  i) Java API packages
  ii) User defined packages

Java API packages



Package	Contents
java.lang	Language support classes, Includes classes for primitive types, strings, math functions, threads and exceptions
java.util	Language utility classes such as vectors, hash tables, Random numbers, date etc.
java.io	Input / output support classes
java.awt	Set of classes, for implementing graphical user interface, Includes classes for windows, buttons, lists etc.
java.net	Classes for networking
java.applet	Classes for creating/implementing applets

Using System Packages

- The packages are organised in a hierarchical form. There are two ways to access classes stored in a package.
    - Use of fully qualified class name

      eg java.awt.color
    - Use import statement

      import packagename.classname; or

      import packagename.*;
- Packages can be named using the standard naming rules.

Package names begin with small letters, to distinguish package names from class names.

Creating Packages

- A package statement inserted as the first line of the source file:

```
package myPackage;

class MyClass1 { ... }
```

means that all classes in this file belong to the myPackage package.

- The package statement creates a name space where such classes are stored.
- When the package statement is omitted, class names are put into the default package which has no name.
- Java uses file system directories to store packages.
- Creating package involves following steps-

1. Declare the package at the beginning of file as
   package packagename;
2. Define the class that is to be put in the package and declare it as public.
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the code as classname.java in the subdirectory created.
5. Compile the file. This creates .class file in the subdirectory.

Package Hierarchy

- To create a package hierarchy, separate each package name with a dot:

```
package Package1.myPackage2.myPackage3;
```

- You cannot rename a package without renaming its directory!

Accessing a Package

- There are two ways to access a Java system package.
  - Use of fully qualified class name
    eg java.awt.color
  - Use import statement

import packagename.classname; or

import packagename.*;

- The import statement is used when there are many references to a particular package or the package name is too long.
- The same approach can be used to access user defined packages.
- The import statement is used to search a list of packages for a particular class.

  import package1 [.package2].classname;
- Also the another approach can be used

  import packagename.*;
- The * indicates the entire package hierarchy below the packagename.
- Here it is difficult to determine from which package or class a particular member came.

Access Control

- Classes and packages are both means of encapsulating and containing the name space and scope of classes, variables and methods:
  1) packages act as a container for classes and other packages
  2) classes act as a container for data and code
- Access control is set separately for classes and class members.

Access Control : Classes

- Two levels of access:
  1) A class available in the whole program:

     public class MyClass { ... }
  2) A class available within the same package only:

     class MyClass { ... }

Access Control: Members

Four levels of access:

1) a member is available in the whole program:

   public int variable;

   public int method(...) { ... }

2) a member is only available within the same class:

```
private int variable;
private int method(...) { ... }
```

3) a member is available within the same package (default access):

```
int variable;
int method(...) { ... }
```

4) a member is available within the same package as the current class, or within its sub-classes:

```
protected int variable;
protected int method(...) { ... }
```

The sub-class may be located inside or outside the current package.

Access Control Summary

- Any member declared public can be accessed from anywhere.
- Any member declared private cannot be seen outside its class.
- When a member does not have any access specification (default access), it is visible to all classes within the same package.
- To make a member visible outside the current package, but only to subclasses of the current class, declare this member protected.

Access Control

Access Modifier **Access location**	private	default	protected	public
same class	yes	yes	yes	yes
subclass in same package	no	yes	yes	yes
other classes in same package	no	yes	yes	yes
subclass in other packages	no	no	yes	yes
non-subclass in other packages	no	no	no	yes

Using a Package

```
//Delay.java
package mypkg1;
public class Delay {
        public static void milisec(int n)  {
                long i,j,k;
                for(k=0;k<n;k++)  {
                        j=100000;
                         for(i=1; i<100000; i++)
                           j=j-1;
                  }
         }
   }

//Test2.java
import  mypkg1.Delay;
 class Test2   {
         public static void main(String args[])  {
                System.out.println("Start....");
                Delay.milisec(15000);
                System.out.println("Stop....!");
            }
     }
```

- The above example creates a package mypkg1, which contains one public class Delay. This source file is saved as Delay.java in subdirectory mypkg1. The Delay.class file is stcred in same subdirectory.
- Test2.java file imports the mypkg1 package and uses milisec() method of Delay class.
- The Test2.java and Test2.class files are stored in the parent directory of subdirectory mypkg1.

Adding a class to a Package

- Let package p1 is existing package with one class A.
- The new class can be added to p1 using following steps-
  1. Define the class and make it public
  2. Place the package statement as the first statement as package p1; before the class definition as follows

```
package p1;
public class B{
        // Body of class B
}
```

  3. Store this as B.java under directory p1.
  4. Compile B.java. Place B.class in p1.
- Now p1 contains two classes and

```
import p1.*;
```

will import both of them.

- Java source file can have only one public class because the file name should be same as public class.

Hiding Classes

- When we use

```
import p1.*;
```

all public classes are imported.

- If we define some classes as "non public" then those classes are not imported. Such classes are hidden from outside the packages.
- Such classes can be seen and used only by other classes in the same package.

```
package p1;
public class A{
        // Body of class A
}
class B{
        // Body of class B
}
```

```
import p1.*;
------------;
------------;
{
  A  obja;          // allowed
  B  objb;    //  class B not available
------------;
  ------------;
}
```

```
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 + ": Rs." + bal);
     }
  }
class AccountBalance {
   public static void main(String args[]) {
       Balance current[] = new Balance[3];
             current[0] = new Balance("K. J. Joshi", 123.23);
             current[1] = new Balance("L.G. Patil", 157.02);
             current[2] = new Balance("S.D. Kale", -12.33);
             for (int i=0; i<3; i++) current[i].show();
       }
   }
```

- Save, compile and execute:

1) call the file AccountBalance.java
2) save the file in the directory MyPack
3) compile; AccountBalance.class should be also in MyPack
4) set access to MyPack in CLASSPATH variable, or make the parent of MyPack your current directory
5) run:

```
java MyPack.AccountBalance
```

Make sure to use the package-qualified class name.