Interfaces, Packages and ExceptionHandling

Module 3

Interface Fundamentals:

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types

Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways -

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension,with the name of the interface matching the name of the file.
- The byte code of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in directory structure that matches the package name.

Declaring Interfaces

The interface keyword is used to declare an interface. Here is a simple example to declare an interface -

Example

Following is an example of an interface

```
/* File name: NameOfinterface.java */
import java.lang.";
// Any number of import statements
public interface NameOfInterface {
// Any number of final, static fields
// Any number of abstract method declarations
}
```

Creating an Interface:

```
interface <interface_name>
{
//declare constant fields
//declare methods that abstract
// by default.
}
```

To declare an interface, use interface keyword. It is used to provide total abstraction.

That means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default.

A class that implement interface must implement all the methods declared in the interface. To implement interface use implements keyword.

Why do we use interface ?

It is used to achieve total abstraction.

Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance

It is also used to achieve loose coupling.

Interfaces are used to implement abstraction. So the question arises why us interfaces when we have abstract classes?

Difference between Class and Interface

Class

- In class, you can instantiate variable and create an object
- Class can contain concrete (with implementation) methods
- The access specifiers used with classes are private. protected and public.

Interface

- In an interface, you can't instantiate variable & create an object.
- The interface cannot contain concrete (with implementation) methods
- In Interface only one specifier is used- Public

Implementing an Interface:

This is how a class implements an interface. It has to provide the body of all the methods that are declared in interface or in other words you can say that class has to implement all the methods of interface.

```
interface MyInterface
{
/* compiler will treat them as:
* public abstract void methodl();
* public abstract void method2();
"/
public void methodl();
public void method2();
}

class Demo implements MyInterface
{
/* This class must have to implement both the
     abstract methods
* else you will get compilation error
*/
public void methodl()
{
System.out.println("implementation of
     method!");
}
public void method2() {
System.out.println("implementation of
     method2");
}
public static void main(String args[])
{
MyInterface obj = new Demo();
obj.methodl();
}}
```

Output:

Implementation of method1

Using Interface References:

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.

```
interface Speed {
void topSpeed();
}

class Audi implements Speed
{
public void topSpeed(){
System.out.println("Audi 250
    km/h);
}}

class Ford implements Speed
{
public void topSpeed(){
System.out.println("Ford 220 km/h)
}}

class BMW implements Speed
 {
public void topSpeed(){
System.out.println("Bit 300 km/h”);
}}

class Javaapp
{
public static void main(String[] args)
{
Speed spd = new Audi();
spd.topSpeed();
spd=new Ford();
spd.topSpeed();
spd=new BW();
spd.topSpeed();
}}
```

Implementing Multiple Interfaces:

A Java class can only extend one parent class. Multiple inheritance (extends) is not allowed. Interfaces are not classes, however, and a class can implement more than one interface. The parent interfaces are declared in a comma-separated list, after the implements keyword.

Here is an example that shows how to implement multiple interfaces in java.Source: (MultipleInterfaces.java)

```
interface I1
{
abstract String methodl();
}
Interface I2
{
abstract String method2();
}
class MultipleInterfaces implements I1, I2
{   //@Override
public String methodl()
{
return "Hello from method";
// @Override
}
public String method2()
{
return "Hello form method2";
}
public static void main(String[] args)
{
MultipleInterfaces mi = new
    Multipleinterfaces();
System.out.println(mi.methodl());
System.out.println(mi.method2(1));
}}
```

Output:

```
Hello from method1
Hello from method2
```

Overloaded Constructor
This java example has a class with two constructors.

Source: (Overloaded Constructor.java)

```
class OverloadedConstructor
{
public static void main(String[] args)
{
class Dog
{
String name;
Dog(String name)
{
this.name = name;
}
Dog()
{this(“NoName");}
Dog d1=new Dog("Spot");
Dog d2=new Dog();
System.out.println("d1 name :” +
    d1.name);
System.out.println("d2 name :” +
    d2.name);
}}
```

Output:

```
d1 name: Spot
d2 name: NoName
```

Constants in Interfaces:

In the interface describes the use of an deda solely to define constants, and having implement that interface in order to achieve convenient syntactic access to those constants. However, since constants are very often merely an implementation detail, and the interfaces implemented by a class are part of its exported API 197 the constant

```
interface Constants {
public static final int CONSTANT = 1;
}

class Class1 implements Constants {
public static final int CONSTANT = 2;

public static void main(String args[]) throws Exception {
            System.out.println(CONSTANT);
   }
}
```

Output

```
2
```

Now let us consider a program that demonstrates how interface extends another interface

```
interface Interfacel
{    public void f1();    }

 //Interfacel extending Interfacel
interface Interface2 extends Interfacel
{    public void f2();    }

class x implements Interface2
{    //definition of method declared in interfacet
     public void f1()
       {
       System.out.println("Contents of Method ) in
           Interface!");
       }
       public void f2()
       {
       System.out.println("Contents of Method #2() in
           Interface");
       }
  }
```

```
class ExtendingInterface
{
public static void main(String[] args)
     {
     Interface2 v2; //Reference variable of Interface
     v2 = new x(); //assign object of class x
     v2.f1();
     v2.f2();
     x x1=new x();
     x1.f3();
 } }
```

Output:

```
Contents of Method f1() in Interface1
Contents of Method f2() in Interface2
Contents of Method f3() of Class X
```

Interfaces can be extended:

When one interface inherits from another interface, that sub-interface inherits all the methods and constants that its super interface declared. In addition, it can also declare new abstract methods and constants. To extend an interface, you use the extends keyword just as you do in the class definition. Unlike a subclass which can directly extend only one subclass, an interface can directly extend multiple interfaces.

Nested Interfaces

We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface

Packages, Packages and Member Access, Importing Packages

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-inpackages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Package Fundamentals:

A Package is a collection of related classes.

It helps organize your classes into a folder structure and make it easy to locate and use them.

More importantly, it helps improve re-usability. Each package in Java has its unique name and organizes its classes and interfaces into a separate namespace, or name group

Built-in Packages

These packages consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:

1) java.lang: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) java.io: Contains classed for supporting input/output operations.
3) java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support; for Date/Time operations.

4) java.applet: Contains classes for creating Applets.

5) java.awt: Contain classes for implementing the components for graphical user interfaces (like button, ;menus etc).

6) java.net: Contain classes for supporting networking operations.

User-defined packages

These are the packages that are defined by the user. First we create a directory myPackage (name should be same as the name of the package). Then create the MyClass inside the directory with the first statement being the package names.

Static Import

Java Static Import

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Advantage of static import:

Less coding is required if you have access any static member of a class oftenly.

Disadvantage of static import:

If you overuse the static import feature, it makes the program unreadable and unmaintainable.

```
import static java.lang.System.*;
class StaticImportExample
{
   public static void main(String args[])
    {
       out.println("Hello");//Now no need of System.out
       out.println("Java");
    }
}
```

Exception Handling

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually-typically through the use of error codes. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.

Abstractinheritance2-Notepad File Edit Format View Help

```
abstract class B
{
abstract void test():
}
class C extends B
{
void test()
{
System.out.println("test() overrided in
    class C”);
}
}
```

```
public class Abstractinheritance2
{
public static void main(String[] args)
{
System out println(“Program starts”);
C c1=new C();c1.test();
System out println(“Program Ends”);
}
}
//Program starts
//test overrided in class C
//Program Ends
```

Exception Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and ***thrown*** in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is ***caught*** and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed using five keywords:

try: A suspected code segment is kept inside try block.

catch: The remedy is written within catch block.

throw: Whenever run-time error occurs, the code must throw an exception.

throws: If a method cannot handle any exception by its own and some subsequent methods needs to handle them, then a method can be specified with throws keyword with its declaration.

finally: block should contain the code to be executed after finishing try-block.

The general form of exception handling is try

```
try
{
// block of code to monitor errors
}
catch (ExceptionType1 exOb)
{
// exception handler for Exception Type1
}
catch (ExceptionType2 exOb)
{
// exception handler for Exception Type2
}
.....
finally
{
// block of code to be executed after try block ends
}
```

Exception Types

All the exceptions are the derived classes of built- in class viz. ***Throwable***.

It has two subclasses viz.

Exception and Error

Exception class is used for exceptional conditions that user programs should catch. We can inherit from this class to create our own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

Error class defines exceptions that are not expected to be caught under normal circumstances by our program. Exceptions of type ***Error*** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Uncaught Exceptions

Let us see, what happens if we do not handle exceptions.

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

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of Exco to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. Since, in the above program, we have not supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

Any un-caught exception is handled by 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 above example is executed:

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

The stack trace displays *class name*, *method name*, *file name* and *line number* causing the exception. Also, the type of exception thrown viz.

ArithmeticException which is the subclass of Exception is displayed. The type of exception gives more information about what type of error has occurred. The stack trace will always show the sequence of method invocations that led up to the error.

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

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine (Exc1.java:6)
at Exc1.main(Exc1.java:10)
```

Using try and catch

Handling the exception by our own is very much essential as

We can display appropriate error message instead of allowing Java run-time to display stack- trace.

It prevents the program from automatic (or abnormal) termination.

To handle run-time error, we need to enclose the suspected code within try block.

```
class Exc2
{
public static void main(String args[])
{   int d, a;
try
{
d=0;
a = 42/d;
System.out.println("This will not be
    printed.“);
}
catch (ArithmeticException e)
{
System.out.println("Division by zero.");
System.out.println("After catch
    statement");
}
```

Output:

```
Division by zero.
After catch 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.

The output of above program is not predictable exactly, as we are generating random numbers. But, the loop will execute 10 times. In each iteration, two random numbers (b and c) will be generated. When their division results in zero, then exception will be caught. Even after exception, loop will continue to execute.

Displaying a Description of an Exception

We can display this description in a println() statement by simply passing the exception as an argument.

This is possible because Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.

```
catch (ArithmeticException e)
{
    System.out.println("Exception: " + e);
    a = 0;
}
```

Now, whenever exception occurs, the output will be Exception java lang ArithmeticException. /by zero.

Multiple Catch Clause

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.

While using multiple *catch* blocks, we should give the exception types in a hierarchy of subclass to superclass. Because, *catch* statement that uses a superclass will catch all exceptions of its own type plus all that of its subclasses. Hence, the subclass exception given after superclass exception is never caught and is a *unreachable* code, that is an error in Java

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 the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

throw

Till now, we have seen catching the exceptions that are thrown by the Java run-time system. It is possible for your program to throw an exception explicitly, using the throw statement. The general form of **throw** is shown here: *throw ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as int or char, as well as **non-Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.

There are two ways you can obtain a Throwable object:

- using a parameter in a catch clause, or
- creating one with the new operator.

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

```
public static void main(String args[])
{
demoproc();
}
```

Output:

```
Caught inside
demoproc:
java.lang.NullPointer
Exception: demo
```

Here, new is used to construct an instance of **NullPointerException.** Many of Java's built-in run-time exceptions have at least two constructors:

- one with no parameter and
- one that takes a string parameter

When the second form is used, the argument specifies a string that describes the exception.

This string is displayed when the object is used as an argument to **print**() or **println**(). It can also be obtained by a call to **getMessage**(), which is defined by **Throwable**.

throws

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

The general form of a method declaration that includes a **throws** clause:

type *method-name*(*parameter-list*) throws exception-list

{

// body of method

}

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

```
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.Illegal
AccessException: demo
```

finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Sometimes 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 exception-handling mechanism. The **finally** keyword is designed to address such situations

The **finally** clause creates a block of code that will be executed after a **try/catch** block has completed and before the next code of **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.

```
class FinallyDemo
{
static void procA()
{
try
{
System.out.println("inside procA");
throw new RuntimeException ("demo");
} finally{
System.out.println("procA's finally");
}}
static void procB()
{ try {
System.out.println("inside procB");
return;
} finally {
System.out.println("proceB’s finally");
}}

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

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. Such exceptions are called as unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions.

Java.lang defines few checked exceptions which needs to be listed out by a method using throws list if that method generate one of these exceptions and does not handle it itself. Java defines several other types of exceptions that relate to its various class libraries.

Table: Java's Unchecked Exceptions

Exception

Creating your own Exception Subclasses

Although Java's built-in exceptions handle most common errors, sometimes we may want to create our own exception types to handle situations specific to our applications. This is achieved by defining a subclass of Exception class. Your subclasses don't need to actually implement anything-it is their existence in the type system that allows you to use them as exceptions. The Exception class does not define any methods of its own. It inherits those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.

We may wish to override one or more of these methods in exception classes that we create. Two of the constructors of Exception are:

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

Though specifying a description when an exception is created is often useful, sometimes it is better to override **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 your description. By overriding **toString**(), you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

Chained Exceptions

The concept of chained exception allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an ArithmeticException because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

To allow chained exceptions, two constructors and two methods were added to Throwable. The constructors are shown here:

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

In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the Error, Exception, and RuntimeException classes.

Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program's logic. Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

Note that Java's exception-handling statements should not be considered a general mechanism for nonlocal branching.

If you do so, it will only confuse your code and make it hard to maintain