**JAVA INTERFACES**

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. 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 bytecode of an interface appears in a **.class** file.
* Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including:

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

Declaring Interfaces:

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

Example:

Let us look at an example that depicts encapsulation:

/\* 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\

}

Interfaces have the following properties:

* An interface is implicitly abstract. You do not need to use the **abstract** keyword when declaring an interface.
* Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
* Methods in an interface are implicitly public.

Example:

/\* File name : Animal.java \*/

interface Animal {

public void eat();

public void travel();

}

Implementing Interfaces:

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

/\* File name : MammalInt.java \*/

public class MammalInt implements Animal{

public void eat(){

System.out.println("Mammal eats");

}

public void travel(){

System.out.println("Mammal travels");

}

public int noOfLegs(){

return 0;

}

public static void main(String args[]){

MammalInt m = new MammalInt();

m.eat();

m.travel();

}

}

This would produce the following result:

Mammal eats

Mammal travels

When overriding methods defined in interfaces there are several rules to be followed:

* Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
* The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
* An implementation class itself can be abstract and if so interface methods need not be implemented.

When implementation interfaces there are several rules:

* A class can implement more than one interface at a time.
* A class can extend only one class, but implement many interfaces.
* An interface can extend another interface, similarly to the way that a class can extend another class.

Extending Interfaces:

An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

//Filename: Sports.java

public interface Sports

{

public void setHomeTeam(String name);

public void setVisitingTeam(String name);

}

//Filename: Football.java

public interface Football extends Sports

{

public void homeTeamScored(int points);

public void visitingTeamScored(int points);

public void endOfQuarter(int quarter);

}

//Filename: Hockey.java

public interface Hockey extends Sports

{

public void homeGoalScored();

public void visitingGoalScored();

public void endOfPeriod(int period);

public void overtimePeriod(int ot);

}

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces:

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as:

public interface Hockey extends Sports, Event

**JAVA PACKAGES**

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations ) providing access protection and name space management.

Some of the existing packages in Java are::

* **java.lang** - bundles the fundamental classes
* **java.io** - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

## Defining a Package:

## To create a package is quite easy: simply 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 no name.

## While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. This is the general form of the package statement:

## package pkg;

## Here, pkg is the name of the package. For example, the following statement creates a package called MyPackage.

## package MyPackage;

## 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. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

## package pkg1[.pkg2[.pkg3]];

## A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

## package java.awt.image;

## needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

## A Short Package Example

## Keeping the preceding discussion in mind, you can try this simple package:

## // A simple package

## package MyPack;

## class Balance {

## String name;

## double bal;

## Balance(String n, double b) {

## name = n;

## bal = b;

## }

## void show() {

## if(bal<0)

## System.out.print("--> ");

## System.out.println(name + ": $" + bal);

## }

## }

## class AccountBalance {

## public static void main(String args[]) {

## Balance current[] = new Balance[3];

## current[0] = new Balance("K. J. Fielding", 123.23);

## current[1] = new Balance("Will Tell", 157.02);

## current[2] = new Balance("Tom Jackson", -12.33);

## for(int i=0; i<3; i++) current[i].show();

## }

## }

## 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

## Remember, you will need to be in the directory above MyPack when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path MyPack.) As explained, AccountBalance is now part of the package MyPack. This means that it cannot be executed by itself. That is, you cannot use this command line:

## java AccountBalance

## AccountBalance must be qualified with its package name.

**JAVA EXCEPTIONS**

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

* A user has entered invalid data.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

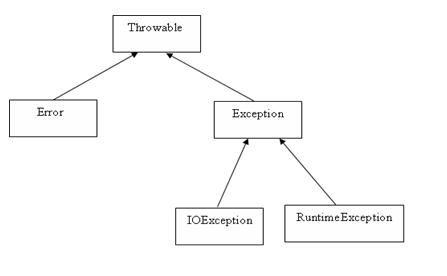
* **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
* **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
* **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

try

{

//Protected code

}catch(ExceptionName e1)

{

//Catch block

}

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

// File Name : ExcepTest.java

import java.io.\*;

public class ExcepTest{

public static void main(String args[]){

try{

int a[] = new int[2];

System.out.println("Access element three :" + a[3]);

}catch(ArrayIndexOutOfBoundsException e){

System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}

This would produce the following result:

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

Out of the block

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

try

{

//Protected code

}catch(ExceptionType1 e1)

{

//Catch block

}catch(ExceptionType2 e2)

{

//Catch block

}catch(ExceptionType3 e3)

{

//Catch block

}

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example:

Here is code segment showing how to use multiple try/catch statements.

try

{

file = new FileInputStream(fileName);

x = (byte) file.read();

}catch(IOException i)

{

i.printStackTrace();

return -1;

}catch(FileNotFoundException f) //Not valid!

{

f.printStackTrace();

return -1;

}

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws**keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

The following method declares that it throws a RemoteException:

import java.io.\*;

public class className

{

public void deposit(double amount) throws RemoteException

{

// Method implementation

throw new RemoteException();

}

//Remainder of class definition

}

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

import java.io.\*;

public class className

{

public void withdraw(double amount) throws RemoteException,

InsufficientFundsException

{

// Method implementation

}

//Remainder of class definition

}

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

try

{

//Protected code

}catch(ExceptionType1 e1)

{

//Catch block

}catch(ExceptionType2 e2)

{

//Catch block

}catch(ExceptionType3 e3)

{

//Catch block

}finally

{

//The finally block always executes.

}

Example:

public class ExcepTest{

public static void main(String args[]){

int a[] = new int[2];

try{

System.out.println("Access element three :" + a[3]);

}catch(ArrayIndexOutOfBoundsException e){

System.out.println("Exception thrown :" + e);

}

finally{

a[0] = 6;

System.out.println("First element value: " +a[0]);

System.out.println("The finally statement is executed");

}

}

}

This would produce the following result:

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

First element value: 6

The finally statement is executed

Note the following:

* A catch clause cannot exist without a try statement.
* It is not compulsory to have finally clauses when ever a try/catch block is present.
* The try block cannot be present without either catch clause or finally clause.
* Any code cannot be present in between the try, catch, finally blocks.