CHp-25
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.

CHAPTER

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
p
ublic 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.

Aclass 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");
}
p
ublic void travel(){
System.out.println("Mammal travels");
}
p
ublic int noOfLegs(){
return0;
}
```

```
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
```

Tagging Interfaces:

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

```
package java.util;
public interface EventListener
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces:

Creates a common parent: As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class: This situation is where the term tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

CHP-26
Java Packages

Packages are used in Java inorder to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerationsss 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 classed.

Creating a package:

When creating a package, you should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.
The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.
If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

Example:

Let us look at an example that creates a package called **animals**. It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.
Put an interface in the package *animals*:

```
/* File name : Animal.java */
package animals;
interface Animal{
public void eat();
```

CHAPTER

```
public void travel();
}
```

Now, put an implementation in the same package *animals*:

```
package animals;
```

```
/* File name : MammalInt.java */
public class MammalInt implements Animal{
public void eat(){
System.out.println("Mammal eats");
}
p
ublic void travel(){
System.out.println("Mammal travels");
}
p
ublic int noOfLegs(){
return0;
}
public static void main(String args[]){
MammalInt m =new MammalInt();
m.eat();
m.travel();
}
}
```

Now, you compile these two files and put them in a sub-directory called **animals** and try to run as follows:

```
$ mkdir animals
$ cp Animal.classMammalInt.class animals
$ java animals/MammalInt
Mammal eats
Mammal travels
```

The import Keyword:

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

Example:

Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;
public class Boss
{
publicvoid payEmployee(Employee e)
{
e.mailCheck();
}
}
```

What happens if Boss is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

TUTORIALS POINT Simply Easy Learning

- The fully qualified name of the class can be used. For example:

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example:

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example:

```
import payroll.Employee;
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

The Directory Structure of Packages:

Two major results occur when a class is placed in a package:

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in Java:

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**. For example:

```
// File Name : Car.java
package vehicle;
public class Car{
// Class implementation.
}
```
Now, put the source file in a directory whose name reflects the name of the package to which the class belongs:
```
....\vehicle\Car.java
```
Now, the qualified class name and pathname would be as below:

 Class name -> vehicle.Car

 Path name -> vehicle\Car.java (in windows)

In general, a company uses its reversed Internet domain name for its package names. Example: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example: The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this:
```
....\com\apple\computers\Dell.java
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is**.class**

For example:
```
// File Name: Dell.java
package com.apple.computers;
public class Dell{
}
classUps{
}
```
Now, compile this file as follows using -d option:
```
$javac -d .Dell.java
```
This would put compiled files as follows:
```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```
You can import all the classes or interfaces defined in *\com\apple\computers\* as follows:
```
import com.apple.computers.*;
```
Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as:
```
<path-one>\sources\com\apple\computers\Dell.java
<path-two>\classes\com\apple\computers\Dell.class
```
By doing this, it is possible to give the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\compters.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (UNIX). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Set CLASSPATH System Variable:

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell):

 In Windows -> C:\> set CLASSPATH

 In UNIX -> % echo $CLASSPATH

To delete the current contents of the CLASSPATH variable, use:

 In Windows -> C:\> set CLASSPATH=

 In UNIX -> % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable:

 In Windows -> set CLASSPATH=C:\users\jack\java\classes

 In UNIX -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH