Interfaces:

* An interface is a collection of abstract methods.

* It can never be instantiated.

* Doesn’t contain any constructors or instance fields. The only fields that can appear in an interface must be declared both static and final.

* Not extended by a class but it is implemented by a class.

* An interface can extend multiple interfaces.

Declaring interfaces:

Syntax:

```
interface Interface_name
{
        // any number of final and static fields
        // any number of abstract method declarations
}
```

Interfaces have the following properties

* An interface is implicitly abstract.

* Methods in an interface are implicitly public

Extending Interfaces:

* Interfaces can also be extended similar to class

* The “extends” keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

```
(e.g.) interface itemconst
       {
               int code=1001;
               String name="xyz";
       }

       interface itemmethods
       {
               void display();
        }
        interface item extends itemconst,itemmethods
        {
               ……………..
               ……………..
         }
```

Difference between abstract classes and Interfaces:

* Main difference is methods of a Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implements a default behaviour.

*Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.

*Members of a Java interface are public by default. A Java abstract class can have the usual flavours of class members like private, protected, etc..

*Java interface should be implemented using keyword "implements"; A Java abstract class should be extended using keyword "extends".

*An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.

*A Java class can implement multiple interfaces but it can extend only one abstract class.

Nested Interfaces:

***** An interface can be declared a member of a class or another interface.

* A nested interface can be declared as public, private or protected.

* When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

(e.g.)
```
class A
{
        public interface NestedIF
        {       boolean isNotNegative(int x);                }
}
class B implements A.NestedIF
{
        public boolean isNotNegative(int x)
        {       return x < 0 ? false : true                  }
 }
```

* Interfaces can be used to import constants into multiple classes by declaring an interface that contains variables that are initialized to the desired values.

(e.g.)
```
interface SharedConstants
{
        int NO=0;
        int YES=1;
        int MAYBE=2;
        int LATER=3;
        int SOON=4;
        int NEVER=5;
}
```

Encapsulation:

* Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods.

* If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

* Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

* The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

```
public class emp
{
        private String name;
        private int age;
        public void setAge(int newAge)
        {
                age=newAge;
        }
        public int getAge()
        {
                return age;
        }
        public void setName(String newName)
        {
                name=newName;
        }
        public int getName()
        {
                return name;
        }
```

Packages:

* Grouping of classes/interfaces together.

* The grouping of classes/interfaces is usually done according to functionality. Packages act as “containers” for classes.

Benefits of Packages:

* The classes contained in the package of other programs can be easily reused.

* Two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the class name.

* Packages provide a way for separating “design” from “coding”.

* Packages provide a way to “hide” classes thus preventing other programs or packages from accessing classes that are meant for internal use only.

Java packages are classified into two types:

* Java API packages
* User Defined packages

Java API Packages:

* Java API provides a large number of classes grouped into different packages according to functionality.

java.lang	-	Language support classes. They include classes for primitive types, strings, math functions, threads and exceptions.
java.util	-	Language utility classes such as vectors, hash tables, random numbers, date, time, etc…
java.io	-	Input/Output support classes. They provide facilities for the input and output of data.
java.awt	-	Set of classes for implementing GUI. They include classes for windows, buttons, lists, menus and so on.
Java.net	-	Classes for networking.
Java.applet	-	Classes for creating and implementing applets.

Using System Packages:

There are two ways of accessing the classes stored in a package.

* Use fully qualified class name.
  (e.g.) java.awt.Color

* use import statements.
  (e.g.) import java.awt.Color;
  import java.awt.*;

* Packages begin with lowercase letters.
  (e.g.) double y=java.lang.Math.sqrt(x);

* Package names should be unique.

Defining Packages:

* When no package has been explicitly specified, the default (or) global package is used.

* The default package has no name, which makes the default package transparent.

* Java uses the file system to manage packages, with each package stored in its own directory.

* Declare the name of the package using the package keyword followed by a package name.

```
package firstpackage;            // Package declaration
public class firstclass          // Class definition
{        ……….
         ……….
}
```

* The .class files must be located in a directory that has the same name as the package.

-

Steps for creating package:

* Declare the package at the beginning of a file.

* Define the class that is to be put in the package and declare it public.

* Create a subdirectory under the main source files are stored.

* Store the listing as the classname.java file in the subdirectory created.

* Compile the file. This creates .class file in the subdirectory.

```
package firstpackage.secondpackage
```

* The general form of import statement for searching a class is:

```
import package1 [.package2][.package3].classname;

(or)

import packagename.*;
```

// Here, packagename may denote a single package or hierarchy of
// packages.
// The * indicates that the compiler should search this entire package
// hierarchy when it encounters a class name.

* A Java package file can have more than one class definitions. Only one of the classes may be declared public and that class name with .java extension is the source file name.

Finding Packages and Classpath:

* First by default, the run-time system uses the current working directory as its starting point.

* Specify a directory path or paths by setting the CLASSPATH environment variable.

* Use the –classpath option with java and javac to specify the path to to your classes.

Static import

- In general, any class from same package can be called without importing it.
- In case, if the class is not part of the same package, we need to provide the import statement to access the class.
- We can access any static fields or methods with reference to the class name. Here comes the use of static imports.
- Static imports allow us to import all static fields and methods into a class and you can access them without class name reference.

- The syntax for static imports are

1. To access all static members of a class

```
import static package-name.class-name.*;
```

2. To access specific static variable of a class

```
import static package-name.class-name.static-variable;
```

3. To access specific static method of a class

```
import static package-name.class-name.static-method;
```

To access all static members of a class

NewClass1.java

```
package pak1;
public class NewClass1 {
  public  static int a=120;
   public static void m(){
        System.out.println("Java");
    }
}
```

NewClass.java

```
package newpackage;
 import static pak1.NewClass1.*;
public class NewClass {
    public static void main(String []
arg){
        m();
        System.out.println(a);
    }
}
```

To access specific static variable of a class

NewClass1.java

```
package pak1;
public class NewClass1 {
  public  static int a=120;
   public static void m(){
        System.out.println("Java");
    }
}
```

NewClass.java

```
package newpackage;
 import static pak1.NewClass1.a;
public class NewClass {
    public static void main(String []
arg){

        System.out.println(a);
    }
}
```

To access specific static method of a class

NewClass1.java

```
package pak1;
public class NewClass1 {
  public  static int a=120;
   public static void m(){
        System.out.println("Java");
    }
}
```

NewClass.java

```
package newpackage;
 import static pak1.NewClass1.m;
public class NewClass {
    public static void main(String []
arg){
        m();

    }
}
```