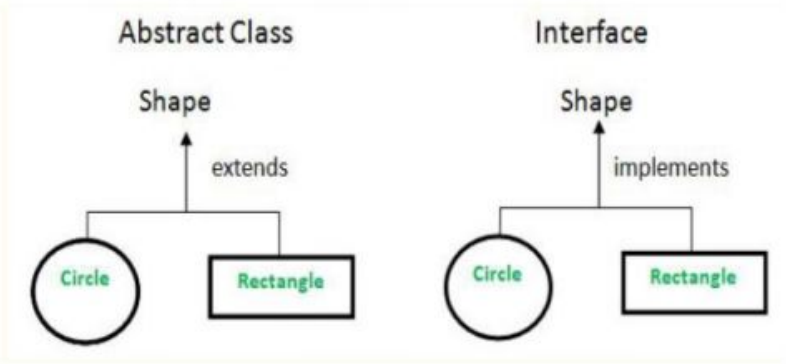
	Abstract Class		Interface
1.	Abstract class can have both an abstract as well as concrete methods.	1.	Interface can have only abstract methods.
2.	Multiple Inheritance is not supported.	2.	Interface supports Multiple Inheritance.
3.	final, non-final, static and non-static variables supported.	3.	Only static and final variables are permitted.
4.	Abstract class can provide the implementation of interface.	4.	Interface can't provide the implementation of abstract class.
5.	Abstract class declared using abstract keyword.	5.	Interface is declared using interface keyword.
6.	Abstract class can inherit another class using extends keyword and implement an interface.	6.	Interface can inherit only an interface.
7.	An abstract class can have static, final or static final variable with any access specifier.	7.	Interface can only have public static final (constant) variable.
	Syntax:		Syntax:

8.	abstract class class_name
	{
	// code
	}
	
	Eg: abstract class Xyz{
	//code
	}
		8.	interface <interface_name>
			{
			//methods
			}
			
			Eg: interface Xyz{
			//method
			}



Defining an Interface

An interface is a blueprint of a class. It is similar to class. It is a collection of abstract methods. It is used to achieve abstraction and multiple inheritance. interfaces can have abstract methods and variables. It cannot have a method body.

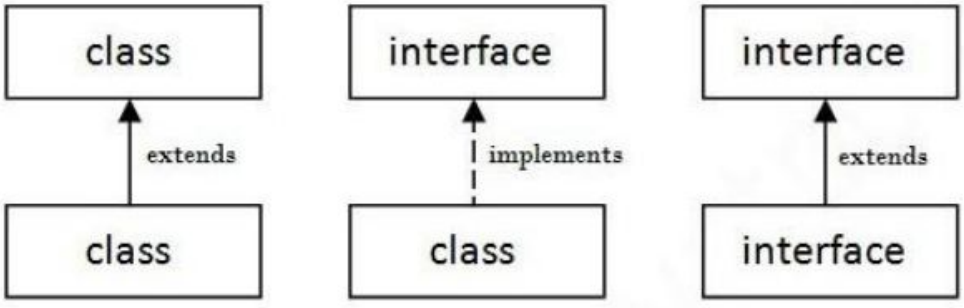
Internal addition by the compiler:

The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

An interface is different from a class in several ways:

- 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.

The relationship between classes and interfaces:



When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces.

Declaring Interfaces:

The **interface** keyword is used to declare an interface.

Syntax:

```
interface <interface_name>

{

        // declare constant fields

        // declare methods that abstract

        // by default.

}
```

Example:

```
interface MyInterface

{
   public void method1();     // interface method (does not have a body)
  public void method2();      // interface method (does not have a body)
}
```

Implementing Interfaces

A class uses the **implements** keyword to implement an interface.

Example 1.

```
interface MyInterface
{
  public void method1();
  public void method2();
}
```

```
class Demo implements MyInterface
{

  public void method1()
  {
        System.out.println("implementation of method1");
  }
  public void method2()
  {
        System.out.println("implementation of method2");
  }
  public static void main(String arg[])
  {
        MyInterface obj = new Demo();
        obj.method1();
  }
}
```

Output:

```
implementation of method1
```

//Implementing multiple inheritance through Interfaces

```
interface A
{
        int a=10;
        void show();
}
interface B
{
        int a=20;
        void show();
}
class C implements A,B
{
        int c;
        void show();
{
        c=A.a+B.a;
        System.out.println("A.a="+A.a);
        System.out.println("B.a="+B.a);
        System.out.println("C"+c);
}
class Test
{
        public static void main(String args[])
{
```

```
        C obj=new c();
        obj.show();
}
}
```

OUTPUT:

A.a=10

B.a=20

C=30

//Example program for Extending Interfaces:

```
interface A
{
void m1();
void m2();
}
interface B extends A
{
void m3();
}
class  C implements B
{
public  void m1();
{
System.out.println("interface A  m1() definition");
}
public  void m2();
{
System.out.println("interface A  m2() definition");
}
public  void m3();
```

```
{
System.out.println("interface B  m3() definition");
}
}
class Test
{
public static void main(String args[])
{
        C obj=new c();
        obj.m1();
        obj.m2();
        obj.m3();
}
}
```

OUTPUT:

```
interface A  m1() definition
interface A  m2() definition

interface B  m3() definition
```

Accessing Implementations Through Interface References

This process is similar to using a superclass reference to access a subclass object.

//Example program for Accessing Implementations Through Interface References

```
interface Callback
{
  void callback(int param);
}
class Client implements Callback
{

   public void callback(int p)
{
     System.out.println("callback called with " + p);
   }
   void aa()
{
    System.out.println("hi.");
   }
```

```
}
public class Main
{
  public static void main(String args[])
{
    Callback c = new Client();
    c.callback(42);
  }
}
```

OUTPUT:

callback called with 42

Nested or Inner interfaces:

An interface which is declared inside another interface or class is called nested interface. They are also known as inner interface. we can only call the nested interface by using outer class or outer interface name followed by dot(.), followed by the interface name.

Syntax:

```
interface interface_name

{

...

 interface nested_interface_name

{

 ...

 }  }
```

//Example 1: Nested interface declared inside another interface:

```
interface MyInterfaceA
{
   void display();
   interface MyInterfaceB
{
       void myMethod();
   }
}

class NestedInterfaceDemo1
   implements MyInterfaceA.MyInterfaceB
{
    public void myMethod()
```

```
{
        System.out.println("Nested interface method");
    }

    public static void main(String args[])
{
        MyInterfaceA.MyInterfaceB obj=
                new NestedInterfaceDemo1();
     obj.myMethod();
    }
}
```

Output:

Nested interface method

Interface inside Class:

Syntax:

```
 class class_name

{

 ...

interface nested_interface_name

{

...

 }

 //Nested interface declared inside a class


class MyClass
{
   interface MyInterfaceB
{
       void myMethod();
   }
}

class NestedInterfaceDemo2 implements MyClass.MyInterfaceB
{
    public void myMethod()
{
        System.out.println("Nested interface method");
    }

    public static void main(String args[])
```

```
{
      MyClass.MyInterfaceB obj=new NestedInterfaceDemo2();
      obj.myMethod();
    }
}
```

Output:

Nested interface method

Packages

A package is a collection of classes, interfaces and sub packages.

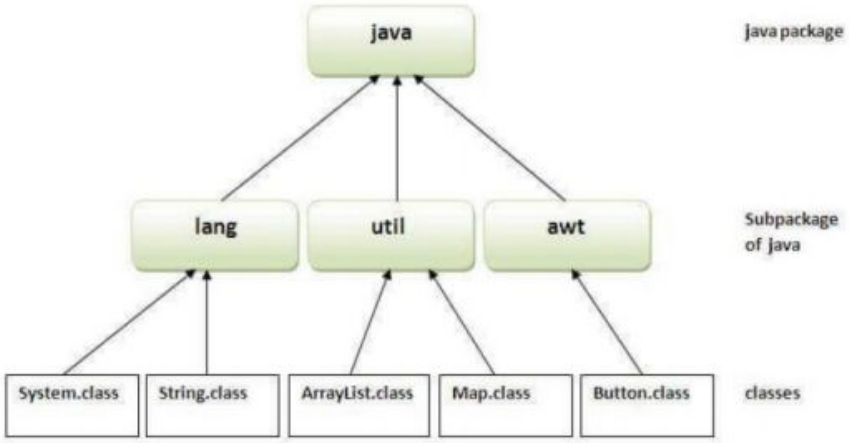
Packages are similar to folders, which are mainly used to organize classes and interfaces.

Packages are used for:

- **Preventing naming conflicts:** Packages help to resolve the naming conflict between the two classes with the same name. Assume that there are two classes with the same name Student.java.

Each class will be stored in its own packages such as stdPack1 and stdPack2 without having any conflict of names.

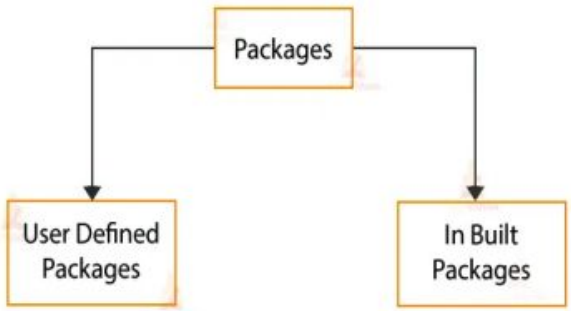
- **Access protection:** A package provides access protection. It can be used to provide visibility control. The members of the class can be defined in such a manner that they will be visible only to elements of that package.



Types of Packages in Java

They can be divided into two categories:

1. User-defined packages and
2. Predefined Packages (Built-in Packages)



User-defined Package

The package which is defined by the user is called a User-defined package. It contains user-defined classes and interfaces.

Creating package

Java supports a keyword called "package" which is used to create user-defined packages in java programming.

```
package packageName;
```

example:

```
package myPackage;
public class A
{
    // class body
}
```

//Example of Java Package

```
package packagedemo;          //package
class Example
{
public static void main(String args[])
{
System.out.println("Welcome to Techvidvan's Java Tutorial");
}
}
```

Output:

Welcome to Techvidvan's Java Tutorial

Predefined Packages in Java (Built-in Packages):

These packages consist of a large number of predefined classes, interfaces, and methods that are used by the programmer to perform any task in his programs.

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.

Some important & commonly used packages.

Accessing Packages or Classes from Another Package

to access all the classes and interfaces of an existing package then we use the **import** statement.

There are three ways to access the package from outside the package.

- import package.*;
- import package.classname;
- fully qualified name.

Using packagename.*

● If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

● The import keyword is used to make the classes and interface of another package accessible to the current package.

Syntax:

import packageName.*;

//Example for import package.*;

```
//save by A.java
package pack;
public class A
{
public void msg()
{
        System.out.println("Hello");
```

```
}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B
{
public static void main(String args[])
{
A obj = new A();
obj.msg();
}
}
```

OUTPUT:

Hello

Using packagename.classname

- If you import package.class_name then only declared class of this package will be accessible.

//Example for import package.classname;

```
//save by A.java
package pack;
public class A{
public void msg()

{

System.out.println("Hello");

}
}
```

```
//save by B.java
package mypack;
import pack.A;
class B

{
public static void main(String args[])

{

A obj = new A();
obj.msg();
```

```
}
}
```

Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

```
//Example
//save by A.java
package pack;
public class A

{
public void msg()
System.out.println("Hello");

}
}

package mypack; //save by B.java
class B

{
public static void main(String args[])

{
pack.A obj = new pack.A();//using fully qualified
name
obj.msg();

}
}
```

CLASSPATH

Classpath is an environment variable which is used by application class loader to locate and load the .class files.

The default value of class path is (.) it means current directory searched.

Class path can be set any of the following:

- Class path can be set permanently in the environment

Windows

↓

Control panel

↓

System

↓

Advanced system settings

↓

Environment variables

↓

Choose system variables [for all the users]

(or)

User variables [only the currently login user]

↓

New

↓

Enter “path” as the variable name

↓

Enter the required directories as the value

↓

OK

To check the setting of the CLASSPATH, use the following command in command prompt(cmd)

- SET CLASSPATH

➢ CLASSPATH can be set temporarily by using (cmd) command prompt.

- SET CLASSPATH=”c:\programfiles\java\JDK\bin;”