**Abstract class in Java**

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

**Abstraction in Java**

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

**Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

**Abstract class in Java**

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

**Points to Remember**

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have constructors and static methods also.

**Example of abstract class**

abstract class A{}

**Abstract Method in Java**

A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

abstract void printStatus();//no method body and abstract

**Example of Abstract class that has an abstract method**

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

abstract class Bike{

abstract void run();

}

class Honda4 extends Bike{

void run(){System.out.println("running safely");}

public static void main(String args[]){

Bike obj = new Honda4();

obj.run();

}

}

**Understanding the real scenario of Abstract class**

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the factory method.

A factory method is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

abstract class Shape{

abstract void draw();

}

//In real scenario, implementation is provided by others i.e. unknown by end user

class Rectangle extends Shape{

void draw(){System.out.println("drawing rectangle");}

}

class Circle1 extends Shape{

void draw(){System.out.println("drawing circle");}

}

//In real scenario, method is called by programmer or user

class TestAbstraction1{

public static void main(String args[]){

Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method

s.draw();

}

}

**Another example of Abstract class in java**

abstract class Bank{

abstract int getRateOfInterest();

}

class SBI extends Bank{

int getRateOfInterest(){return 7;}

}

class PNB extends Bank{

int getRateOfInterest(){return 8;}

}

class TestBank{

public static void main(String args[]){

Bank b;

b=new SBI();

System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

b=new PNB();

System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

}}

**Interface in Java**

An interface in Java is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

**Why use Java interface?**

There are mainly two reasons to use interface. They are given below.

1. It is used to achieve abstraction.
2. By interface, we can support the functionality of multiple inheritance.

**Syntax:**

interface <interface\_name>{

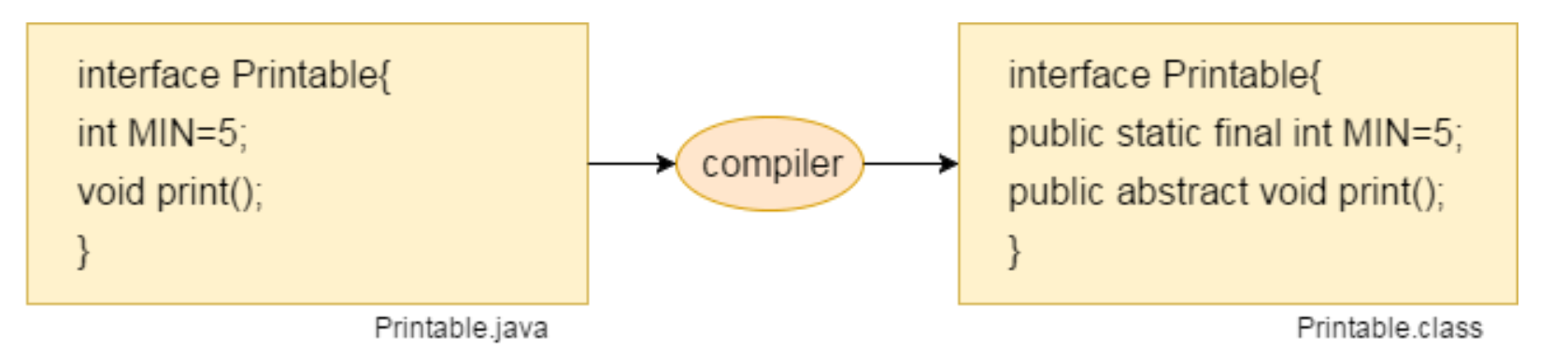
// declare constant fields

// declare methods that abstract

// by default.

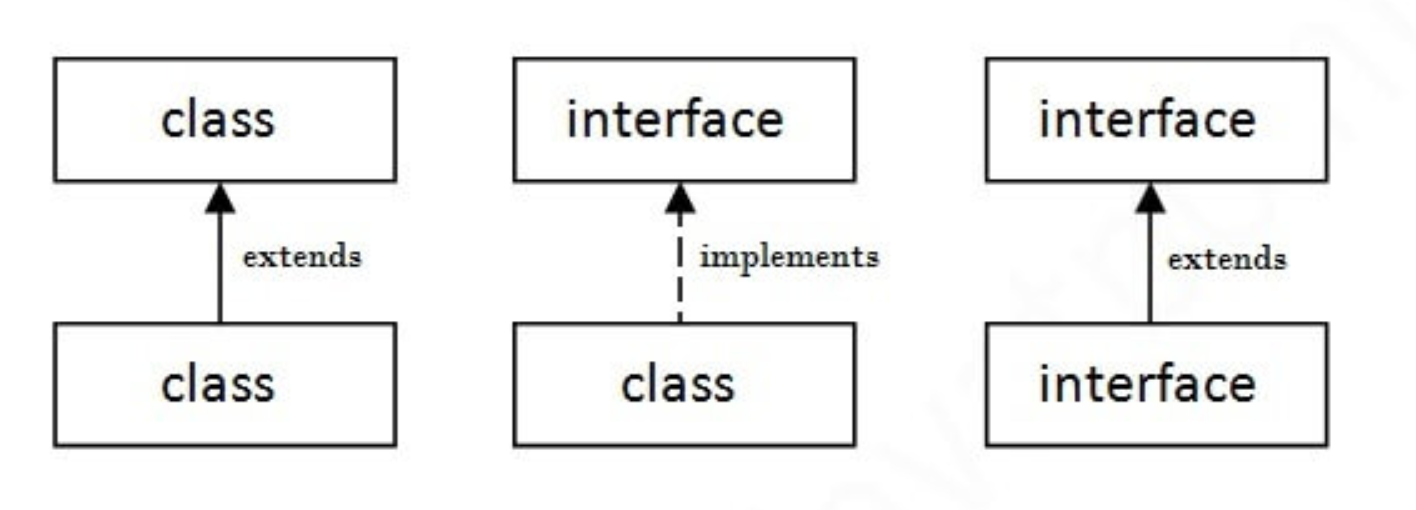
}

**Internal addition by the compiler**



**The relationship between classes and interfaces**

As shown in the figure given below, a class extends another class, an interface extends another interface, but a class implements an interface.



**Java Interface Example**

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

interface Printable{

void print();

}

class A6 implements printable{

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

public static void main(String args[]){

A6 obj = new A6();

obj.print();

}

}

**Java Interface Example: Bank**

interface Bank{

float rateOfInterest();

}

class SBI implements Bank{

public float rateOfInterest(){return 9.15f;}

}

class PNB implements Bank{

public float rateOfInterest(){return 9.7f;}

}

class TestInterface2{

public static void main(String[] args){

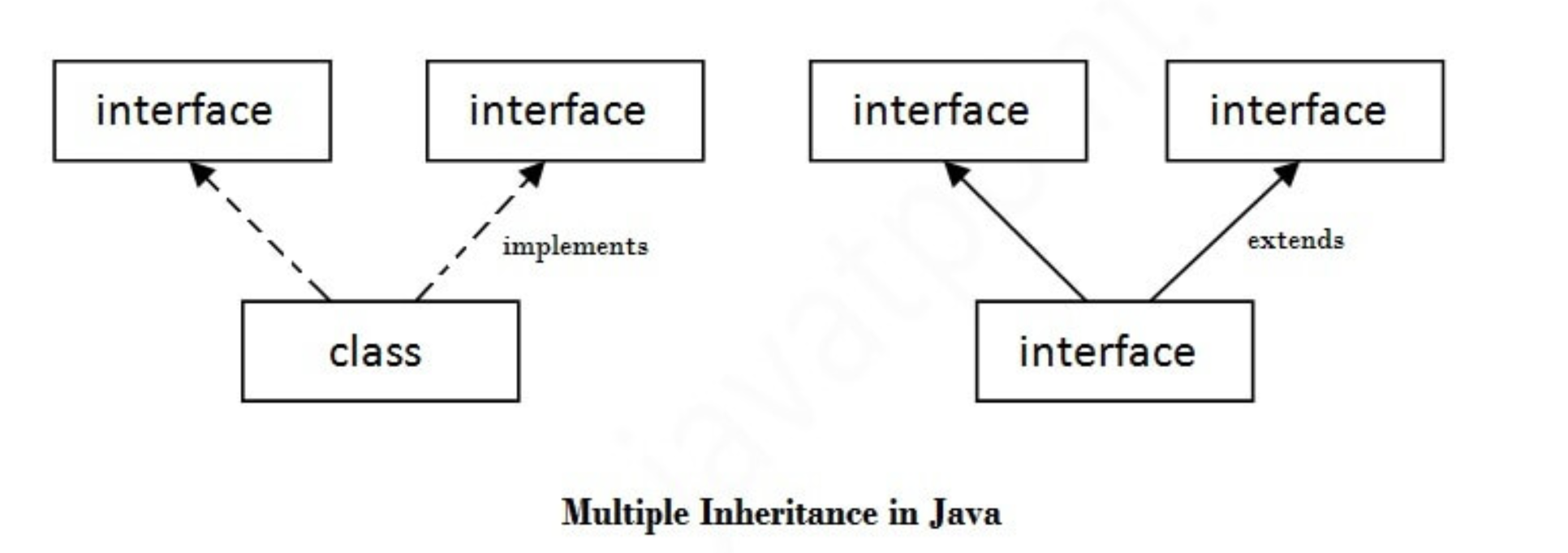
Bank b=new SBI();

System.out.println("ROI: "+b.rateOfInterest());

}}

**Multiple inheritance in Java by interface**

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



interface Printable{

void print();

}

interface Showable{

void show();

}

class A7 implements Printable,Showable{

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

public void show(){System.out.println("Welcome");}

public static void main(String args[]){

A7 obj = new A7();

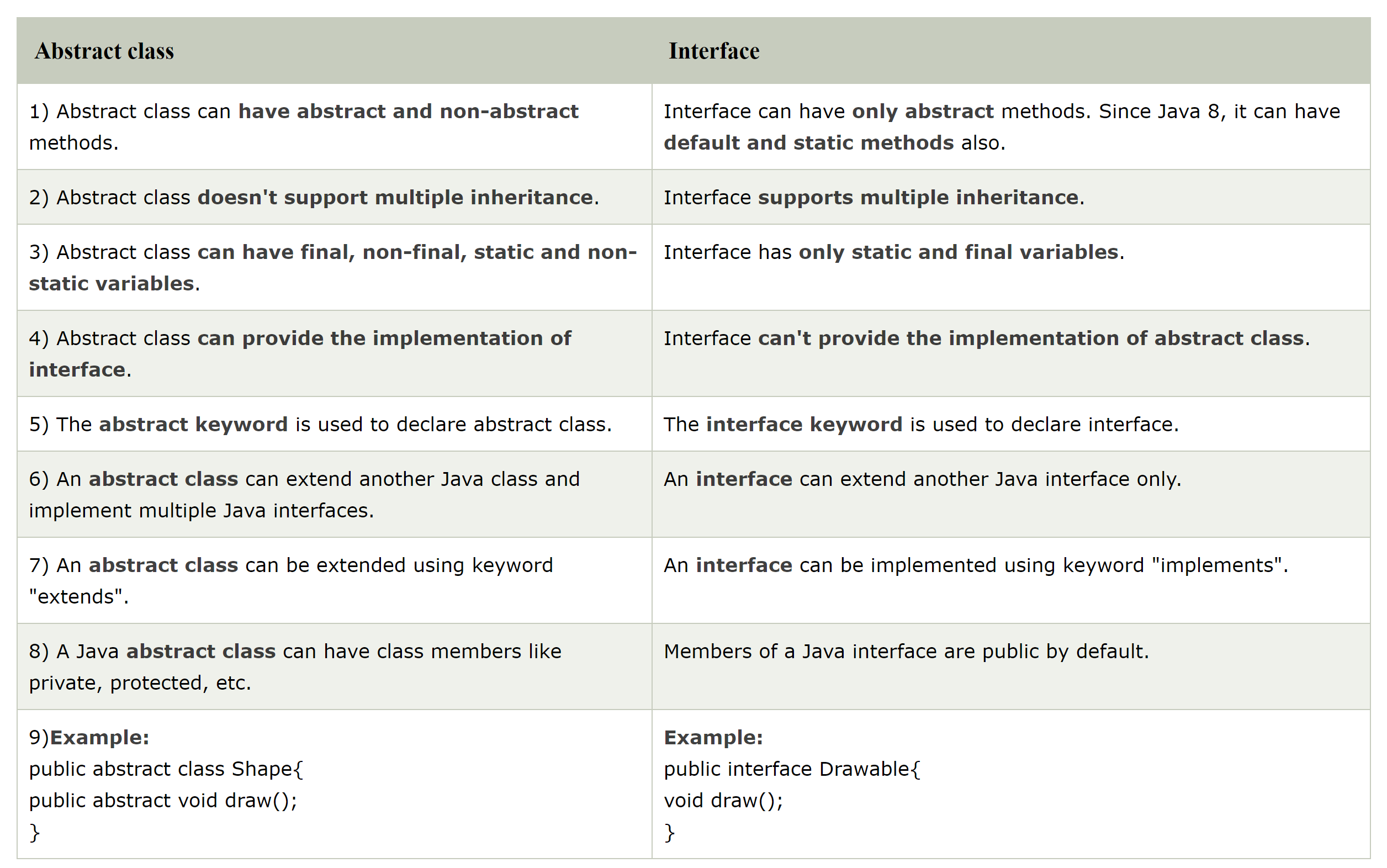
obj.print();

obj.show();

}

}

**Difference between abstract class and interface**



**Java Package**

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-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

**Advantage of Java Package**

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

**Access Modifiers in Java**

There are two types of modifiers in Java: access modifiers and non-access modifiers.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

**Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

**Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

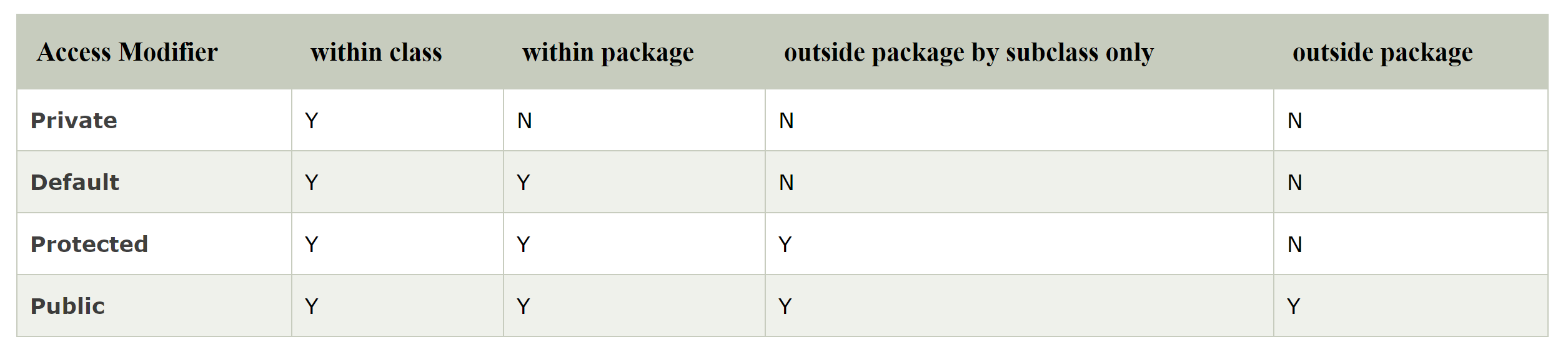
**Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

**Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.



**1) Private**

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

class A{

private int data=40;

private void msg(){System.out.println("Hello java");}

}

public class Simple{

public static void main(String args[]){

A obj=new A();

System.out.println(obj.data);//Compile Time Error

obj.msg();//Compile Time Error

}

}

**Role of Private Constructor**

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

class A{

private A(){}//private constructor

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

}

public class Simple{

public static void main(String args[]){

A obj=new A();//Compile Time Error

}

}

**2) Default**

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example of default access modifier**

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

//save by A.java

package pack;

class A{

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();//Compile Time Error

obj.msg();//Compile Time Error

}

}

**3) Protected**

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

**Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

//save by A.java

package pack;

public class A{

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

}

//save by B.java

package mypack;

import pack.\*;

class B extends A{

public static void main(String args[]){

B obj = new B();

obj.msg();

}

}

**4) Public**

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

//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();

}

}

**Encapsulation in Java**

Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class read-only or write-only. In other words, you can skip the getter or setter methods.

It provides you the control over the data. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.

The standard IDE's are providing the facility to generate the getters and setters. So, it is easy and fast to create an encapsulated class in Java.

**Simple Example of Encapsulation in Java**

public class Student{

//private data member

private String name;

//getter method for name

public String getName(){

return name;

}

//setter method for name

public void setName(String name){

this.name=name

}

}

class Test{

public static void main(String[] args){

//creating instance of the encapsulated class

Student s=new Student();

//setting value in the name member

s.setName("vijay");

//getting value of the name member

System.out.println(s.getName());

}

}

**Read-Only class**

public class Student{

//private data member

private String college="AKG";

//getter method for college

public String getCollege(){

return college;

}

}

Now, you can't change the value of the college data member which is "AKG".

s.setCollege("KITE");//will render compile time error

**Write-Only class**

//A Java class which has only setter methods.

public class Student{

//private data member

private String college;

//getter method for college

public void setCollege(String college){

this.college=college;

}

}

Now, you can't get the value of the college, you can only change the value of college data member.