INHERITANCE

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing class
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined in the parent class

Inheritance

- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Reusability* is at the heart of inheritance
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

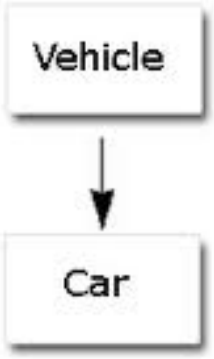
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

- The **extends** **keyword** indicates that you are making a new class that derives from an existing class

Inheritance

- Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead pointing to the parent class



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

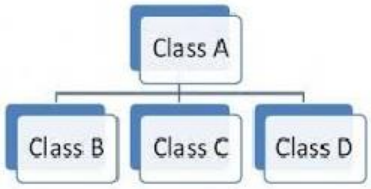
Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Car extends Vehicle
{
   // class contents
}
```

Types of inheritance

- Single inheritance – A derived class inherited from one base class
- Hierarchical inheritance- More

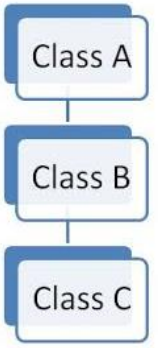


one derived classes inherited

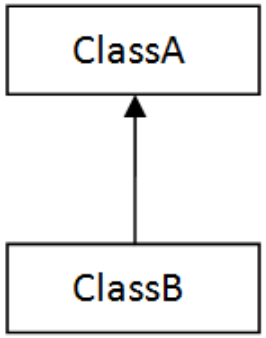
from one base class

- Multilevel inheritance- A derived class inherited

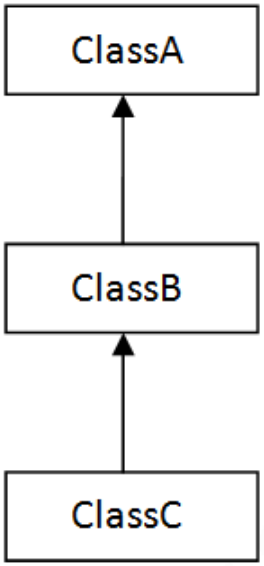
from another derived class



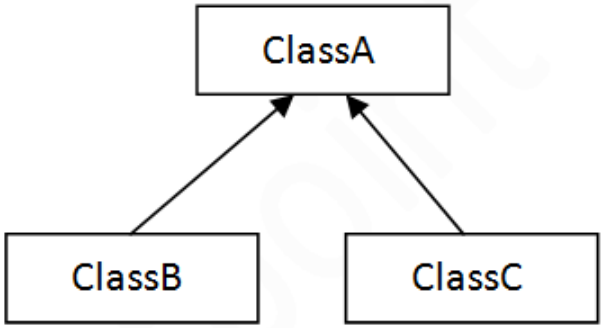
Types of Inheritances in Java



1) Single



2) Multilevel



3) Hierarchical

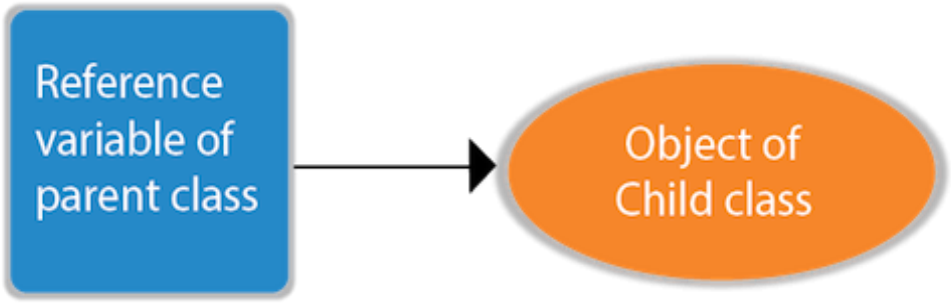
```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme
method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme
method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme
method");
}
}
```

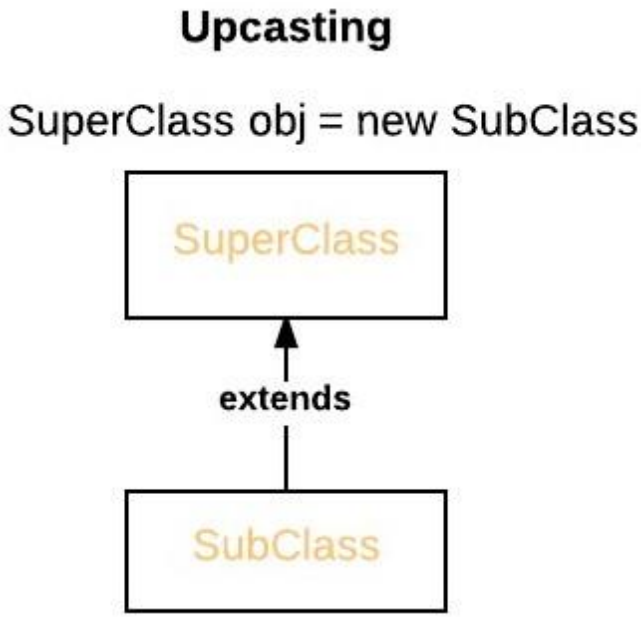
```
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r;
r = a;
r.callme();
r = b;
r.callme();
r = c;
r.callme();
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting





```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");
}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

```
class Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}
```

Output:

```
SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7
```

Abstract class

- A class that is declared with abstract keyword is known as abstract class in java. It can have abstract and non-abstract methods (method with body)
- **abstract** keyword in front of the **class** keyword
- Restricted class that cannot be used to create objects
- Cannot declare abstract constructors, or abstract static methods

```
abstract class Sample
{
    // contents
}
```

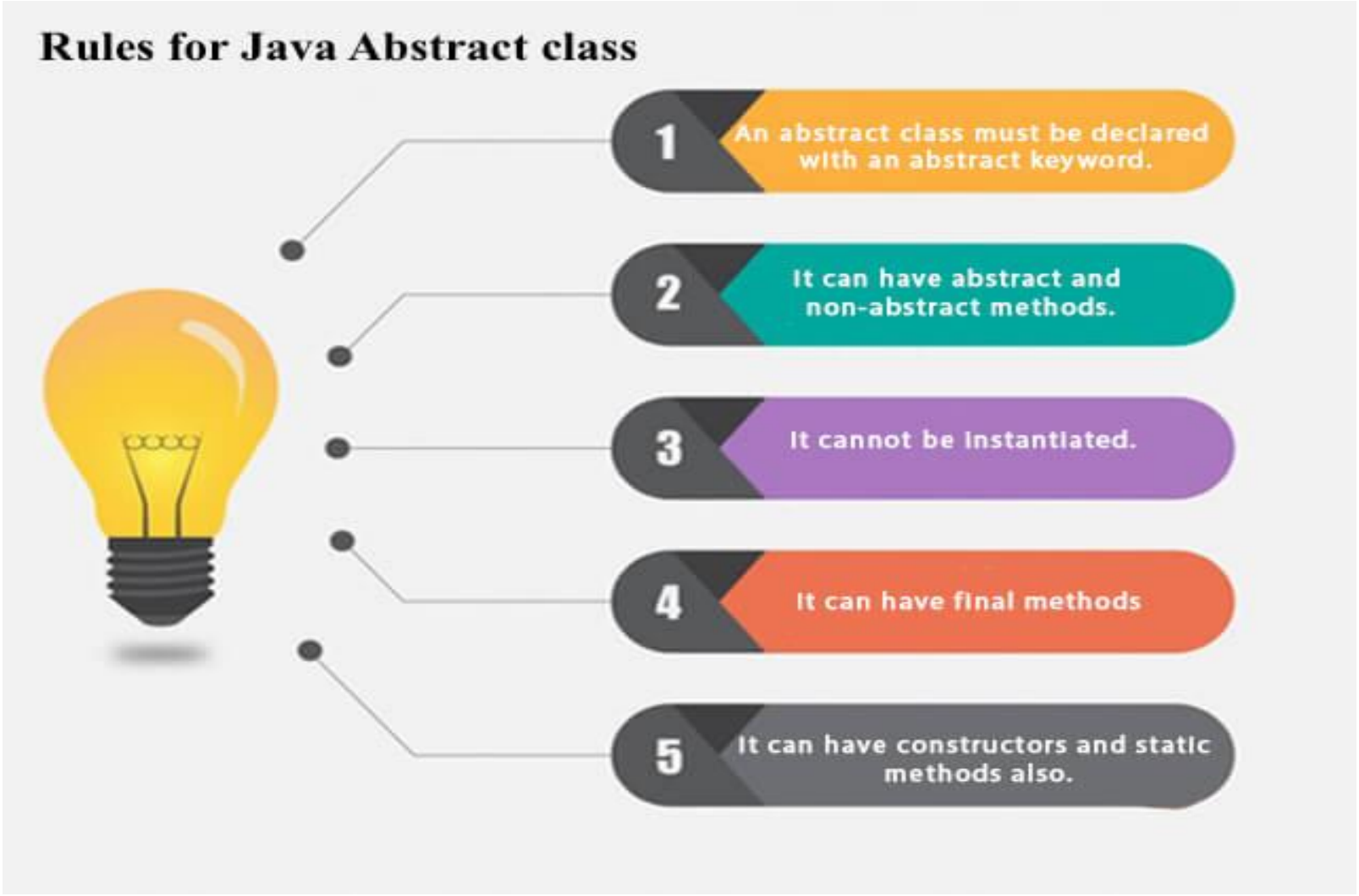
```
abstract class A {
abstract void callme();
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

```
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;    }
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
double area() {
System.out.println("Inside Area for
Rectangle.");
return dim1 * dim2;  }
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
double area() {
System.out.println("Inside Area for
Triangle.");
return dim1 * dim2 / 2;}}

class Main {
public static void main(String args[]) {
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // no object is created
figref = r;
System.out.println("Area is " +
figref.area());
figref = t;
System.out.println("Area is " +
figref.area());
}
}
```

Reference variable → Can be used to refer to an object of any class derived from super class

Abstract Classes



Final keyword

- Final Variables- can not be changed
  - final int a=10;
- Final Methods – can not be overridden
  - final void show();
- Final Class- Prevent a class from inheritance
  - can not be inherited

```
final class classname
{

}
```

Interfaces

- An interface is a way to describe what classes should do, without specifying how they should do it.
- Interfaces are similar to abstract classes
- Interfaces can contain only abstract methods and constants.
- Interfaces cannot be instantiated.
- Can use interface as a data type for variables.
- Can also use an interface as the result of a cast operation.

Declaration- syntax

- An interface is created with the following syntax:

```
Access modifier interface interfaceID
{
  //constants/method signatures
}
```

- Extending interfaces

modifier interface interface_name extends base _interface

{

}

Interface member – constants

- An interface can define named constants, which are public, static and final (these modifiers are omitted by convention) automatically. Interfaces never contain instant fields.
- All the named constants MUST be initialized.

```
interface SharedConstants {
        int SUN = 0;
        int MON = 1;
        int TUE = 2;
        int WED = 3;
        int THU = 4;
        int FRI = 5;
        int SAT = 6;
        }
```

```
import java.util.Scanner;
interface SharedConstants {
int SUN = 0; int MON = 1; int TUE = 2;
int WED = 3;  int THU = 4; int FRI = 5;
int SAT = 6;    }
class Days implements SharedConstants  {
void answer(int x)  {
switch(x) {
case SUN:
System.out.println("SUNDAY");
break;
case MON:
System.out.println("MONDAY");
break;
case TUE:
System.out.println("TUESDAY");
break;
case WED:
System.out.println("WEDNESDAY");
break;
case THU:
System.out.println("THURSDAY");
break;
case FRI:
System.out.println("FRIDAY");
break;
case SAT:
System.out.println("SATURDAY");
break;  }   }    }
class Main{
public static void main(String args[])  {
Days q = new Days();
 Scanner sc=new Scanner(System.in);
 int n= sc.nextInt();
q.answer(n); }  }
```

```
interface Shape
{
   void area(int a, int b);
   void print1();
  }
class Triangle implements Shape
{
  public void area(int a,int b)
   {
      int area = a*b;
     System.out.println(area);
   }
   public void print1()
   {
      System.out.println("Hello");
   }
}
```

```
class Main
    {
   p s v m (String[] args) {
   Triangle obj=new Triangle();
    obj.area(5,10);
     }
    }
```

Why And When To Use Interfaces?

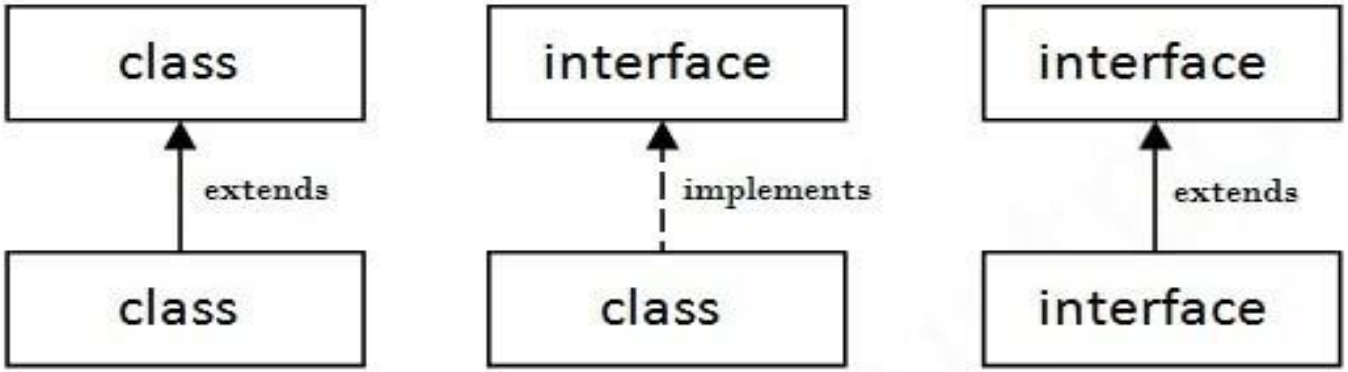
1) **To achieve security** - hide certain details and only show the important details of an object (interface).

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.

Note: To implement multiple interfaces, separate them with a comma

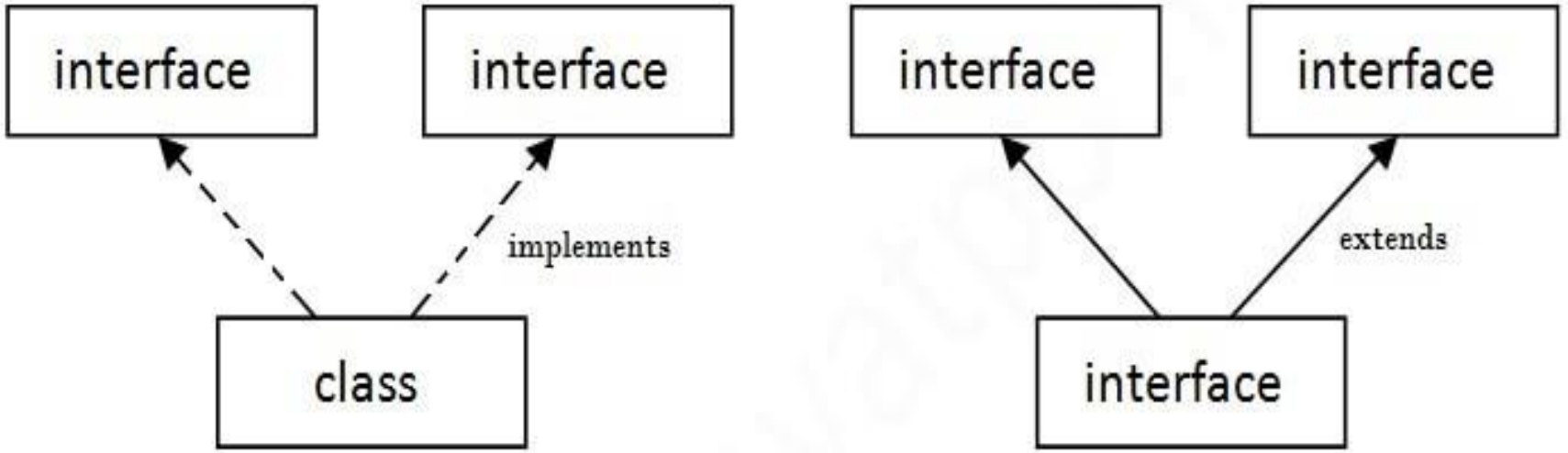
The relationship between classes and interfaces

- A class extends another class, an interface extends another interface, but a **class implements an interface**.



Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

- Multiple inheritance is not supported in the case of class because of ambiguity (Same method name in multiple classes).
- However, it is supported in case of an interface because there is no ambiguity.
- It is because its implementation is provided by the sub class.

```
interface Printable{
void print();
}
interface Showable{
void print();
}
class TestInterface1 implements Printable, Showable{
public void print()
{
System.out.println("Hello");
}
public static void main(String args[]){
TestInterface1 obj = new TestInterface1();
obj.print();
 }
}
```

Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
 public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Package

- Both a naming and a visibility control mechanism.
- Classes can be defined inside a package that are not accessible by code outside that package
- Class members can be defined that are exposed only to other members of the same package
- General form of the **package** statement:

  package *pkg*;

- Hierarchy of packages is created by separating 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*]];

Packages

- To organize classes and interfaces.
- It is a group of similar types of classes, interfaces and sub-packages.

Two types of packages in Java

1. Built-in package (packages from the Java API)
2. User-defined package (create your own packages)

Built-in packages

- The **Java API** is a library of prewritten classes, that are free to use, included in the Java Development Environment.
- The library is divided into **packages** and **classes**.
- import a **single class** (along with its methods and attributes), or a **whole package that contain all the classes** that belong to the specified package.
- To use a class or a package from the library - **to use the import keyword**

Syntax

- import root*package*. *name*.*Class*;

  // Import a single class

- import root*package*. *name*.*;

  // Import the whole package

Import a Class

- Example,

  Scanner class, **which is used to get user input**

  ```
  import java.util.Scanner;
  ```

→ java is a top level package
→ util is a sub package
→ and Scanner is a class which is present in the sub package util.

Import a Package

- To import a whole package, end the sentence with an asterisk sign (*).
- The following example will import ALL the classes in the java.util package

```
import java.util.*;
```

Built in pakages

- lang
- awt
- Javax
- java
- swing
- net
- io
- util
- sql etc.

Package Name	Description
java.lang	Contains language support classes (for e.g classes which defines primitive data types, math operations, etc.) . This package is automatically imported.
java.io	Contains classes for supporting input / output operations.
java.util	Contains utility classes which implement data structures like Linked List, Hash Table, Dictionary, etc and support for Date / Time operations.
java.applet	Contains classes for creating Applets.
java.awt	Contains classes for implementing the components of graphical user interface (like buttons, menus, etc.).
java.net	Contains classes for supporting networking operations.

Example:

```
package mypackage;
public class Simple{
public void displayMessage()
{
   System.out.println("Welcome to package");
}
}
//Main.java
import mypackage.Simple;
public class Main {
   public static void main(String[] args) {
         Simple myObject = new Simple();
            myObject.displayMessage();
   }
}
```

Compilation & Run

D:\subashini new\kongu\JP\pgms>javac mypackage/Simple.java

D:\subashini new\kongu\JP\pgms>javac Main.java

D:\subashini new\kongu\JP\pgms>java Main
Welcome to package

Class Member Access

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
 } }
```

This is file Derived.java:

```
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
 // class only
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub); } }
```

This is file SamePackage.java:

```
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
 }
 }
```

```
package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
// class or package only
System.out.println("n = " + n);
// class only
 System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub); }
 }
```

This is file OtherPackage.java:

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
System.out.println("n = " + p.n);
// class only
System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub); } }
```