Inheritance:

- Inheritance can be defined as the process where one class acquires the properties of another class

- The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

- Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

- Advantages of inheritance:

    - Code reusability

    - Used in method overriding (so runtime polymorphism can be achieved).

Syntax:

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.

Example:

```
// A simple example of inheritance.
// Create a superclass.
class A
{
      int i, j;
      void showij()
      {
            System.out.println("i and j: " + i + " " + j);
      }
}
// Create a subclass by extending class A.
class B extends A
{
      int k;
      void showk()
      {
            System.out.println("k: "+ k);
      }
void sum()
{
```

```
                System.out.println("i+j+k: " + (i+j+k));
                }
        }
        class SimpleInheritance
        {
                public static void main(String args [])
                {
                        B subOb = new B();

                        /* The subclass has access to all public members of its superclass. */
                        subOb.i = 7;
                        subOb.j = 8;
                        subOb.k = 9;

                        System.out.println("Contents of subOb: ");
                        subOb.showij();
                        subOb.showk();
                        System.out.println();
                        System.out.println("Sum of i, j and k in subOb:");
                        subOb.sum();
                }
        }
```

```
D:\Materials\JAVA Material\Unit 2\Inheritance>java SimpleInheritance
Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

Member Access and Inheritance

Although a subclass includes all of the members of its super class, it cannot access those members of the super class that have been declared as **private**. For example, consider the following simple class hierarchy:

/* In a class hierarchy, private members remain private to their class. This program contains an error and will not compile. */

```
        class A
        {
                int i; // public by default
                private int j;  //Private to A
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
                    void setij(int x, int y)
                    {
                            i = x;
                            j = y;
                    }
            }
            class B extends A
            {
                    int total;
                    void sum()
                    {
                            total = i + j;  //A's j is not accessible here
                    }
            }
            class SimpleInheritance2
            {
                    public static void main(String args[])
                    {
                            B subOb = new B();
                            subOb.setij(10, 12);
                            subOb.sum();
                            System.out.println("Total is " + subOb.total);
                    }
            }
```

```
D:\Materials\JAVA Material\Unit 2\Inheritance>javac SimpleInheritance2.java
SimpleInheritance2.java:22: error: j has private access in A
              total = i + j;  //A's j is not accessible here
                          ^
1 error
```

A super class variable can reference a subclass object:

A reference variable of a super class can be assigned a reference to any subclass derived from that super class.

Ex:

```
            class A
            {
                    void callMe()
                    {
                            System.out.print("Hello");
                    }
            }
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
class B extends A
{
        void callMe()
        {
                System.out.print("Hi ");
        }
}
class Reference
{
        public static void main(String args[])
        {
                A ref;
                B b = new B();
                ref = b;
                ref.callMe();
        }
}
```

Output: Hi

Using super

Whenever the derived class inherits the base class features, there is a possibility that base class features are similar to derived class features and JVM gets an ambiguity. To overcome this, super is used to refer super class properties.

The **super** keyword in java is a reference variable that is used to refer parent class. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

- super is used to refer immediate parent class instance variable.
    super.parent_instance_variable_name;
- super is used to invoke immediate parent class method
    super.parent_class_method_name();
- super is used to invoke immediate parent class constructor.
    super(arglist); // parameterized constructor
    super(); //default

Usage 1. - Using super to refer super class property

```
class Vehicle
{
        int speed=50;
}
class Bike extends Vehicle
{
         int speed=100;
         void display()
         {
            System.out.println("Vehicle Speed:"+super.speed);//will print speed of vehicle
            System.out.println("Bike Speed:"+speed);//will print speed of bike
         }
}
class SuperVarible
{
            public static void main(String args[])
            {
                    Bike b=new Bike();
                    b.display();
            }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java SuperVarible
Vehicle Speed:50
Bike Speed:100
```

Usage 2. - super is used to invoke immediate parent class method

```
class Student
{
         void message()
         {
                 System.out.println("Good Morning Sir");
         }
}

class Faculty extends Student
{
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
        void message()
        {
                System.out.println("Good Morning Students");
        }
        void display()
        {
                message();//will invoke or call current class message() method
                super.message();//will invoke or call parent class message() method
        }
}
class SuperClassMethod
{
        public static void main(String args[])
        {
                Faculty f=new Faculty();
                f.display();
        }
}
```

Output:


```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java SuperClassMethod
Good Morning Students
Good Morning Sir
```

Usage 3. - super is used to invoke immediate parent class constructor

```
class Vehicle
{
        Vehicle( )
        {
                System.out.println("Vehicle is created");
        }
}

class Bike extends Vehicle
{
        Bike()
        {
                super( );//will invoke parent class constructor
                System.out.println("Bike is created");
        }

}
class SuperClassConst
{
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
        public static void main(String args[])
          {
              Bike b=new Bike();
          }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java SuperClassConst
Vehicle is created
Bike is created
```

Calling Constructors:

Constructors are called in order of derivation, from super class to sub class. Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default constructor of each super class will be executed. The following program illustrates when constructors are executed:

```
// Create a super class.
class A {
        A() {
                System.out.println("Inside A's constructor.");
        }
}
// Create a subclass by extending class A.
class B extends A
{
        B() {
                System.out.println("Inside B's constructor.");
        }
}
// Create another subclass by extending B.
class C extends B {
        C() {
                System.out.println("Inside C's constructor.");
        }
}
class CallingConstructor
{
        public static void main(String args[])
        {
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
                    C c = new C();
              }
       }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java CallingConstructor
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

Method Overriding:

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Example:

```
// Method overriding.
class A
{
       int i, j;
       A(int a, int b)
       {
              i = a;
              j = b;
       }
       // display i and j
       void show()
       {
              System.out.println("i and j: " + i + " " + j);
       }
}

class B extends A
{
       int k;
       B(int a, int b, int c)
       {
              super(a, b);
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
                k = c;
        }
        // display k – this overrides show() in A
        void show()
        {
                System.out.println("k: " + k);
        }
}
class Override
{
        public static void main(String args[]) {
                B subOb = new B(1, 2, 3);
                subOb.show(); // this calls show() in B
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java Override
k: 3
```

Dynamic Method Dispatch:

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Example: //A Superclass Variable Can Reference to a Subclass Object

```
class A
{
        void callMe()
        {
                System.out.println("Inside A");
        }
}
class B extends A
{
        void callMe()
        {
                System.out.println("Inside B");
        }
}

class C extends B
{
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
        void callMe()
        {
                System.out.println("Inside C");
        }
}
class DynamicMethodDispatch
{
        public static void main(String args[])
        {
                A a=new A();
                B b = new B();
                C c = new C();
                A ref;
                ref = a;
                ref.callMe();

                ref = b;
                ref.callMe();

                ref=c;
                ref.callMe();
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\Inheritance>java DynamicMethodDispatch
Inside A
Inside B
Inside C
```

Applying Method Overriding:

```
import java.util.*;

class Student
{
        int n;
        String name;
        void read()
        {
                Scanner s=new Scanner(System.in);
                System.out.println("Enter no and Name");
                n=s.nextInt();
                name=s.nextLine();

        }
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```java
        void show()
        {
                System.out.println("No:"+n);
                System.out.println("Name:"+name);
        }
}
class ITStudent extends Student
{
        void read()
        {
                super.read();
        }
        void show()
        {
                super.show();
        }
}
class CSEStudent extends Student
{
        void read()
        {
                super.read();
        }
        void show()
        {
                super.show();
        }

}
class ECEStudent extends Student
{
        void read()
        {
                super.read();
        }
        void show()
        {
                super.show();
        }
}
class MainMethod
{
        public static void main(String ar[])
        {
                ITStudent it=new ITStudent();
                CSEStudent cse=new CSEStudent();
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
                    ECEStudent ece=new ECEStudent();

                    Student sref;

                    sref=it;
                    sref.read();
                    sref.show();

                    sref=cse;
                    sref.read();
                    sref.show();

                    sref=ece;
                    sref.read();
                    sref.show();
              }
        }
```

Abstract Class

- An abstract class is a class that contains one or more abstract methods.
- An abstract method is a method without method body.

  Syntax:

  abstract return_type method_name(parameter_list);

- An abstract class can contain instance variables, constructors, concrete methods in addition to abstract methods.
- All the abstract methods of abstract class should be implemented in its sub classes.
- If any abstract method is not implemented in its subclasses, then that sub class must be declared as abstract.
- We cannot create an object to abstract class, but we can create reference of abstract class.
- Also, you cannot declare abstract constructors or abstract static methods.

Example 1: Java program to illustrate abstract class.

```
        abstract class MyClass
        {
              abstract void calculate(double x);
        }
        class Sub1 extends MyClass
        {
              void calculate(double x)
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
			{
				System.out.println("Square  :"+(x*x));
			}
		}
		class Sub2 extends MyClass
		{
			void calculate(double x)
			{
				System.out.println("Square Root :"+Math.sqrt(x));
			}
		}
		class Sub3 extends MyClass
		{
			void calculate(double x)
			{
				System.out.println("Cube :"+(x*x*x));
			}
		}
		class AC
		{
			public static void main(String arg[])
			{
				Sub1 obj1=new Sub1();
				Sub2 obj2=new Sub2();
				Sub3 obj3=new Sub3();

				obj1.calculate(20);
				obj2.calculate(20);
				obj3.calculate(20);
			}
		}
```

Example 2: Java program to illustrate abstract class.

```
		// A Simple demonstration of abstract.
		abstract class A
		{
			abstract void callme();
			// concrete methods are still allowed in abstract classes
			void callmetoo() {
				System.out.println("This is a concrete method.");
			}
		}
		 class B extends A
		{
			void callme()
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
                {
                        System.out.println("B's implementation of callme.");
                }
        }

        class AbstractDemo
        {
                public static void main(String args[])
                {
                        B b = new B();
                        b.callme();
                        b.callmetoo();
                }
        }
```

Uses of Final:

Final can be used in three ways:

- To prevent modifications to the instance variable
- To Prevent method overriding
- To prevent inheritance

## Use 2: To prevent method overriding

```
        class Vehicle
        {
                final void run()
                {
                        System.out.println("running");
                }
        }
        class Bike extends Vehicle
        {
                void run()
                {
                        System.out.println("running safely with 100kmph");
                }
        }

        class FinalMethod
        {
                public static void main(String args[])
                {
                        Bike b= new Bike();
                        b.run();
```

```
                    }
              }
Error Information:
```

```
D:\Materials\JAVA Material\Unit 2\Inheritance>javac FinalMethod.java
FinalMethod.java:10: error: run() in Bike cannot override run() in Vehicle
        void run()
             ^
  overridden method is final
1 error
```

Use 3: To prevent inheritance

```
final class Vehicle
{
        void run()
        {
                System.out.println("running");
        }
}
class Bike extends Vehicle
{
        void run()
        {
                System.out.println("running safely with 100kmph");
        }
}

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

```
D:\Materials\JAVA Material\Unit 2\Inheritance>javac FinalClass.java
FinalClass.java:8: error: cannot inherit from final Vehicle
class Bike extends Vehicle
                   ^
1 error
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

Interfaces:

- A named collection of method declarations.

- A Java interface is a collection of constants and abstract methods

- Since all methods in an interface are abstract, the abstract modifier is usually left off

- Using interface, you can specify what a class must do, but not how it does.

- Interface fields are public, static and final by default, and methods are public and abstract.
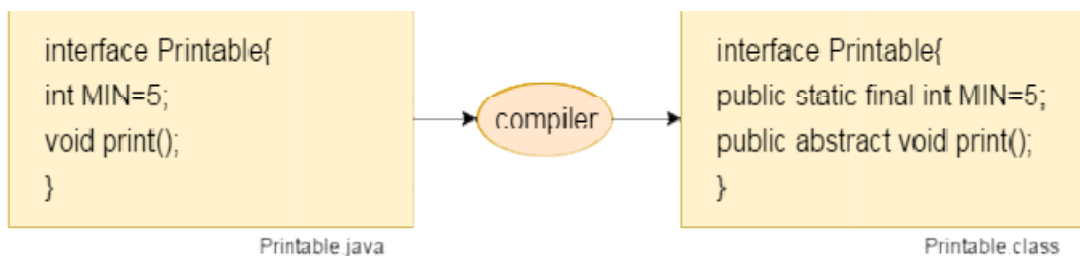
Advantages of interfaces:

- It is used to achieve abstraction.

- By interface, we can support the functionality of multiple inheritances.

Syntax:

access_specifier interface interfae_name
{
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        //...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}

Implementing Interfaces:

- Once an interface has been defined, one or more classes can implement that interface.

- To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

- The general form of a class that includes the **implements clause looks like this:**

    class *classname [extends superclass] [implements interface1 [,interface2...]] {*

        // class-body

    }

- If a class implements more than one interface, the interfaces are separated with a comma.

- The methods that implement an interface must be public. Also, the type signature of implementing method must match exactly the type signature specified in interface definition.

Example 1: Write a java program to implement interface.

```
interface Moveable
{
        int AVG_SPEED=30;
        void Move();
}
class Move implements Moveable
{
         void Move(){
                System .out. println ("Average speed is:  "+AVG_SPEED  );
        }
}
class Vehicle
{
        public static void main (String[] arg)
        {
                Move m = new Move();
                m.Move();
        }
}
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

Example 2: Write a java program to implement interface.

```
interface Teacher
{
        void display1();
}
interface Student
{
        void display2();
}
class College implements Teacher, Student
{
        public void display1()
        {
                System.out.println("Hi I am Teacher");
        }
        public void display2()
        {
                System.out.println("Hi I am Student");
        }
}
class CollegeData
{
        public static void main(String arh[])
        {
                College c=new College();
                c.display1();
                c.display2();
        }
}
```

<u>Accessing implementations through interface references:</u>

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

Ex:

```
interface Test {

    void call();
}

class InterfaceTest implements Test {

    public void call()
    {
        System.out.println("call method called");
    }
}

public class IntefaceReferences {

        public static void main(String[] args)
        {
                Test f ;
                InterfaceTest it= new InterfaceTest();
                f=it;
                f.call();
            }
}
```

Variables in Interfaces:

We can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When we include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations). If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant fields into the class name space as **final** variables.

Example: Java program to demonstrate variables in interface.

```
interface left
{
        int i=10;
}
interface right
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
        {
                int i=100;
        }
class Test implements left,right
        {
                public static void main(String args[])
                {
                        System.out.println(left.i);//10 will be printed
                        System.out.println(right.i);//100 will be printed*/
                }
        }
```

Interfaces can be extended:

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example: Java program to demonstrate interfaces can be extended with extend keyword.

```
interface Teacher
{
        void display1();
}
interface Student
{
        void display2();
}
interface T_S extends Teacher, Student
{
        void display3();
}
 class College implements T_S
{
        public void display1()
        {
                System.out.println("Hi I am Teacher");
        }
        public void display2()
        {
                System.out.println("Hi I am Student");
        }
        public void display3()
        {
                System.out.println("Hi I am Teacher_Student");
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
        }
}
class Class_Interface
{
        public static void main(String arh[])
        {
                College c=new College();
                c.display1();
                c.display2();
                c.display3();
        }

}
```

Example 2: Java program to implement interface and inheriting the properties from a class.

```
interface Teacher

{
        void display1();
}
class Student
{
        void display2()
        {
                System.out.println("Hi I am Student");
        }
}
class College extends Student implements Teacher
{
        public void display1()
        {
                System.out.println("Hi I am Teacher");
        }
}
class Interface_Class
{
        public static void main(String arh[])
        {
                College c=new College();
                c.display1();
                c.display2();
        }
}
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

Difference between Interface and Abstract class:

Abstract Class	Interface
Contains some abstract methods and some concrete methods	Only abstract methods
Contains instance variables	Only static and final variables
Doesn't support multiple inheritance	Supports
public class Apple extends Food { … }	public class Person implements Student,Athlete,Chef { … }

Packages:

- A Package can be defined as a grouping of related types(classes, interfaces)
- A package represents a directory that contains related group of classes and interfaces.
- Packages are used in Java in order to prevent naming conflicts.
- There are two types of packages in Java.
    1. Pre-defined Packages(built-in)
    2. User defined packages

Pre-defined Packages:

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. This package is also called as **Collections.**
java.applet	Contains classes for creating Applets.
java.awt	Contains classes for implementing the components of graphical user interface (like buttons, menus, etc.).

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

java.net	Contains classes for supporting networking operations.
javax.swing	This package helps to develop GUI like java.awt. The 'x' in javax represents that it is an extended package which means it is a package developed from another package by adding new features to it. In fact, javax.swing is an extended package of java.awt.
java.sql	This package helps to connect to databases like Oracle/Sybase/Microsoft Access to perform different operations.

Defining a Package(User defined):

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.

This is the general form of the **package** statement:

package *pkg*;

Here, *pkg* is the name of the package.

For example, the following statement creates a package called **MyPackage**:

package MyPackage;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same **package** statement.

The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate 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*]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

23

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

Example: Package demonstration

```
package pack;
public class Addition
{
        int x,y;
        public Addition(int a, int b)
        {
                x=a;
                y=b;
        }
        public void sum()
        {
                System.out.println("Sum :"+(x+y));
        }
}
```

Step 1: Save the above file with Addition.java

```
package pack;

public class Subtraction
{
        int x,y;
        public Subtraction(int a, int b)
        {
                x=a;
                y=b;
        }
        public void diff()
        {
                System.out.println("Difference :"+(x-y));
        }
}
```

Step 2: Save the above file with Subtraction.java

Step 3: Compilation

To compile the java files use the following commands

javac -d   directory_path   name_of_the_java file

Javac –d   .   name_of_the_java file

Note: -d is a switching options creates a new directory with package name. Directory path represents in which location you want to create package and . (dot) represents current working directory.

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
D:\Materials\JAVA Material\Unit 2\PackExamples>javac -d . Addition.java

D:\Materials\JAVA Material\Unit 2\PackExamples>javac -d . Subtraction.java
```

Step 4: Access package from another package

There are three ways to use package in another package:

1. **With fully qualified name.**

```
class UseofPack
{
        public static void main(String arg[])
        {
                pack.Addition a=new  pack.Addition(10,15);
                a.sum();
                pack.Subtraction s=new pack.Subtraction(20,15);
                s.difference();
        }
}
```

2. **import package.classname;**

```
import pack.Addition;
import pack.Subtraction;
class UseofPack
{
        public static void main(String arg[])
        {
                 Addition a=new  Addition(10,15);
                a.sum();
                Subtraction s=new Subtraction(20,15);
                s.difference();
        }
}
```

3. **import package.*;**

```
import pack.*;
class UseofPack
{
        public static void main(String arg[])
        {
                 Addition a=new  Addition(10,15);
                a.sum();
                Subtraction s=new Subtraction(20,15);
                s.difference();
        }
}
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

Note: Don't place Addition.java, Subtraction.java files parallel to the pack directory. If you place JVM searches for the class files in the current working directory not in the pack directory.

Access Protection

- Access protection defines actually how much an element (class, method, variable) is exposed to other classes and packages.

- There are four types of access specifiers available in java:

    1. Visible to the class only (private).

    2. Visible to the package (default). No modifiers are needed.

    3. Visible to the package and all subclasses (protected)

    4. Visible to the world (public)

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

Example:

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**.

Name of the package: pkg1

This file is Protection.java

```
package pkg1;

public class Protection
{
        int n = 1;
        private int n_priv = 2;
        protected int n_prot = 3;
        public int n_publ = 4;
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
        public Protection()
        {
                System.out.println("base constructor");
                System.out.println("n = " + n);
                System.out.println("n_priv = " + n_priv);
                System.out.println("n_prot = " + n_prot);
                System.out.println("n_publ = " + n_publ);
        }
}
```

This is file Derived.java:

```
package pkg1;

class Derived extends Protection
{
   Derived()
   {
    System.out.println("Same package - derived (from base) constructor");
    System.out.println("n = " + n);

    /* class only
     *  System.out.println("n_priv = "4 + n_priv); */

    System.out.println("n_prot = " + n_prot);
    System.out.println("n_publ = " +n_publ);
   }
}
```

This is file SamePackage.java

```
package pkg1;

class SamePackage
{
   SamePackage()
   {
    Protection pro = new Protection();
    System.out.println("same package - other constructor");
    System.out.println("n = " + pro.n);

    /* class only
     *  System.out.println("n_priv = " + pro.n_priv); */

    System.out.println("n_prot = " + pro.n_prot);
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.

```
              System.out.println("n_publ = " + pro.n_publ);
          }
      }
```

Name of the package: pkg2

This is file Protection2.java:

```
package pkg2;

class Protection2 extends pkg1.Protection
{
        Protection2()
        {
            System.out.println("Other  package-Derived  (from  Package  1-Base)
        Constructor");

         /* class or package only
         *  System.out.println("n = " + n); */

         /* class only
         *  System.out.println("n_priv = " + n_priv); */

         System.out.println("n_prot = " + n_prot);
         System.out.println("n_publ = " + n_publ);
        }
}
```

This is file **OtherPackage.java**

```
        package pkg2;

        class OtherPackage
        {
          OtherPackage()
          {
          pkg1.Protection pro = new pkg1.Protection();

          System.out.println("other package - Non sub class constructor");

          /* class or package only
          *  System.out.println("n = " + pro.n); */

          /* class only
          *  System.out.println("n_priv = " + pro.n_priv); */

          /* class, subclass or package only
```

28

```
     *  System.out.println("n_prot = " + pro.n_prot); */

        System.out.println("n_publ = " + pro.n_publ);
      }
    }
```

If you want to try these t two packages, here are two test files you can use. The one for package **pkg1** is shown here:

```
/* demo package pkg1 */

package pkg1;

/* instantiate the various classes in pkg1 */
public class Demo
{
   public static void main(String args[])
   {
     Derived obj2 = new Derived();
     SamePackage obj3 = new SamePackage();
   }
}
```

The test file for package pkg2 is

```
package pkg2;

/* instantiate the various classes in pkg2 */
public class Demo2
{
   public static void main(String args[])
   {
     Protection2 obj1 = new Protection2();
     OtherPackage obj2 = new OtherPackage();
   }
}
```

```
D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>javac -d . Demo.java

D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>javac -d . Demo2.java

D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>java pkg1.Demo
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
Same package - derived (from base) constructor
n = 1
n_prot = 3
n_publ = 4
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
same package - other constructor
n = 1
n_prot = 3
n_publ = 4
```

```
D:\Materials\JAVA Material\Unit 2\Packages\AccessSpecifier>java pkg2.Demo2
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
Other package - Derived (from Package 1-Base)Constructor
n_prot = 3
n_publ = 4
base constructor
n = 1
n_priv = 2
n_prot = 3
n_publ = 4
other package - Non sub class constructor
n_publ = 4
```

Dr. Suresh Yadlapati, M. Tech, Ph. D, Dept. of IT, PVPSIT.