

# Java Programming

Shobana.G  
Assistant Professor,  
Rajiv Gandhi University of Knowledge& Technology

# Objectives

- To create classes, objects and make use of arrays and strings.
- They will also learn the concepts of inheritance and garbage collection.

# Class

- A class is a user-defined data type with the template that serves to define its properties.
- A class defines new data type. Once defined, this new data type can be used to create objects of that type.
- Class is a template for an object and object is the instance of the class.

# Class definition

```
class class-name
{
    data-type instance-variable1; data-type instance-
    variable2;
    -----;
    data-type instance-variableN;
    data-type method-name1(parameter-list)
    {
        //body of the method1
    }
    data-type    method-name2(parameter-list)
    {
        //body of the method2
    }
    -----;
    data-type method-nameN(parameter-list)
    {
        //body of the methodN
    }
}
```

# Adding variables to

```
class
class Country
{
    long population; float
    currencyRate; int
    noOfStates;
}
```

# Creating objects

Creating the objects of the class is also called as instantiating an object.

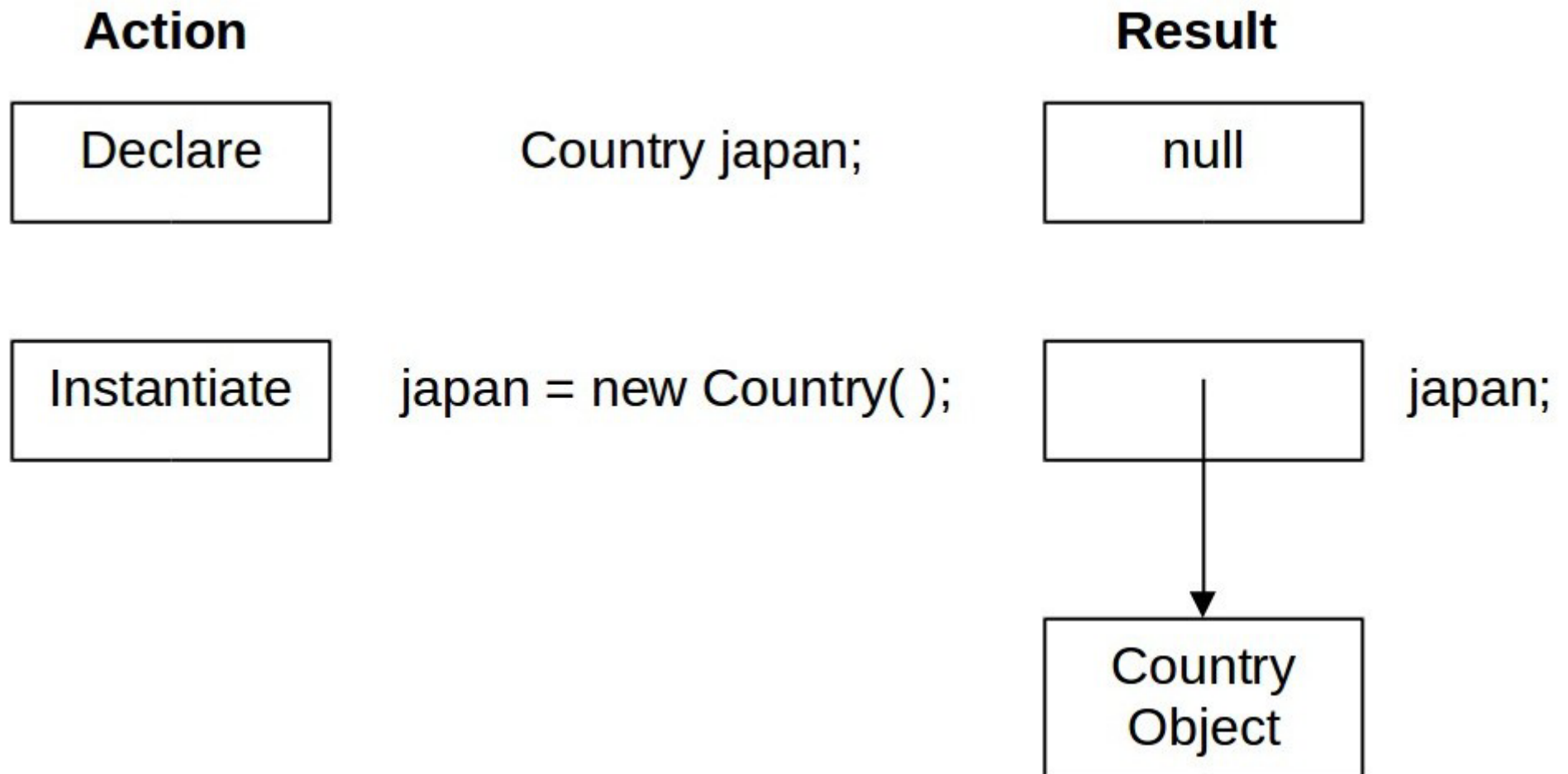
Example:

**Country japan; // declaration**

**japan = new Country(); // instantiation or**

**Country japan = new Country();**

# Declaring and creating object



# Accessing members

Member access follows the following  
syntax:

**objectname.variablename**



# Example:

```
japan.population = bhutan. 58612453;  
currencyRate      = 0.236;  
japan.noOfSates = 5;
```

# Adding methods

The general form of the declaration of the method is,

```
data-type method-name (parameter-list)  
{  
    Body of the method;  
}
```

The method declaration contains four different parts:

- Name of the method (method-name)
- Data type of the values returned by method (data-type)
- The list of parameters (parameter-list)
- Body of the method

# What is a method?

```
float area(int radius)
{
    final    float pi = 3.14f; val;
    float
    val = pi * (float) (radius * radius); return(val);
}
```

# Constructo r

- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and it is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
- Constructors look a little strange because they have no return type, not even void.

# Example:

```
class Country
{
    long population; int noOfStates; float currancyRate;
    Country(long x, int y) { population = x; noOfStates = y;
    }
}

void display()
{
    System.out.println("Population:"+population); System.out.println("No
    of states:"+noOfStates);
}
}
```

# Garbage Collection

- Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.
- An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object.
- An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

# Finalize method

- finalize method in java is a special method much like main method in java. finalize() is called before Garbage collector reclaim the Object, its last chance for any object to perform cleanup activity i.e. releasing any system resources held, closing connection if open etc.

# Example:

```
protected void finalize( ) throws Throwable { try {  
    System.out.println("Finalize of Sub Class");  
    //release resources, perform cleanup ;  
} catch(Throwable t){ throw t;  
} finally {  
    System.out.println("Calling finalize of Super Class");  
    super.finalize();  
}  
}
```



# Type of constructors

- Default constructor
- Parameterized  
constructor

# Example (Constructor Overloading):

```
class Box
{
    int height;
    int depth; int
    length;
    Box()
    {
        height = depth        = length = 10;
    }
    Box(int x,int y)
    {
        height = x; depth = y;
    }
}
```

# The 'this' keyword

- Many times it is necessary to refer to its own object in a method or a constructor.
- To allow this Java defines the 'this' keyword. If we are familiar with C++, it is not hard to learn and understand the concept of 'this' keyword.
- The 'this' is used inside the method or constructor to refer its own object.
- That is, 'this' is always a reference to the object of the current class' type.

# Example:

```
class Box
{
    int height; int depth; int length;
    Box(int height, int depth,
        {
            this.height = height; this.depth =    int length)
            depth; this.length = length;
        }
}
```

# Method overloading

- In Java it is possible to define two or more methods within the same class that are having the same name, but their parameter declarations are different.
- In the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- Method overloading is one of the ways of Java implementation of polymorphism.

# Example:

```
int add(int      x, int y) //version1
```

```
{
```

```
    cal = x + y;
```

```
    return(cal);
```

```
}
```

```
int add(int      z) //version2
```

```
{
```

```
    cal = z + 10;
```

```
    return(cal);
```

```
}
```

```
float add(float x, float      y) //version3
```

```
{
```

```
    val  = x + y;
```

```
    return(val);
```

```
}
```

# Static members

- Methods and variables defined inside the class are called as instance methods and instance variables. That is, a separate copy of them is created upon creation of each new object.
- But in some cases it is necessary to define a member that is common to all the objects and accessed without using a particular object.
- That is, the member belongs to the class as a whole rather than the objects created from the class. Such members can be created by preceding them with the keyword static.

# Example:

**static int cal; static float min = 1;**

**static void display(int x)**



# Static members

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- We can declare both methods and variables to be static.
- The most common example of a static member is `main()`. `main()` is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables.

# Restrictions to static

- They can only call other static methods.
- They must only access static data.
- They cannot refer to this or super in any way.

# Argument passing to methods

- Call by value
- Call by reference

# Call by value

```
class Test
```

```
{
```

```
    void meth(int    i, int j)
```

```
    {
```

```
        i++; j++;
```

```
    }
```

```
}
```

```
    ob.meth(a, b);
```

# Call by reference

```
class Test
{
    int a, b;

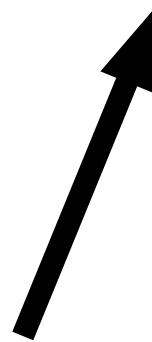
    void meth(Test o)
    {
        a.  ++;
        b.  ++;
    }
}
```

```
Ob1.meth(obj2);
```

# Command line

## arguments

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



Command line  
arguments

# Command line and variable length of arguments

```
javac prog.java
```

```
java prog Hello my name is Tushar
```

```
args[0]    = Hello
```

```
args[1]    = my
```

```
args[2]    = name
```

```
args[3]    = is
```

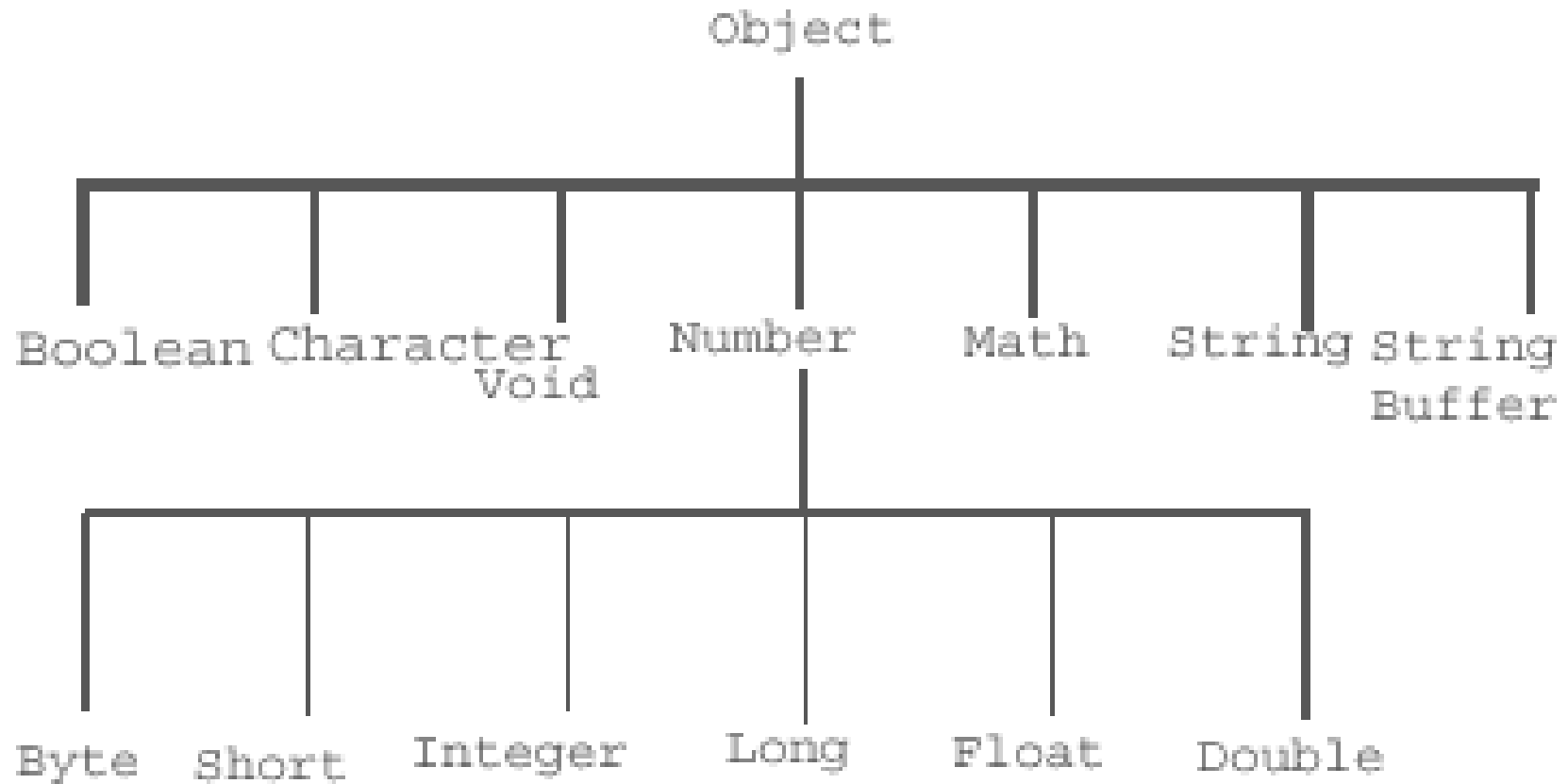
```
args[4]    = Tushar
```

# The Object class

- There is one special class, 'Object', defined by Java.
- All other classes are subclasses of 'Object'. That is, 'Object' is a super class of all other classes.
- This means that a reference variable of type 'Object' can refer to an object of any other class.
- Also, since arrays are implemented as classes, a variable of type 'Object' can also refer to any array.



# The object class



# Class visibility controls

- Public
- Private
- Protected
- default

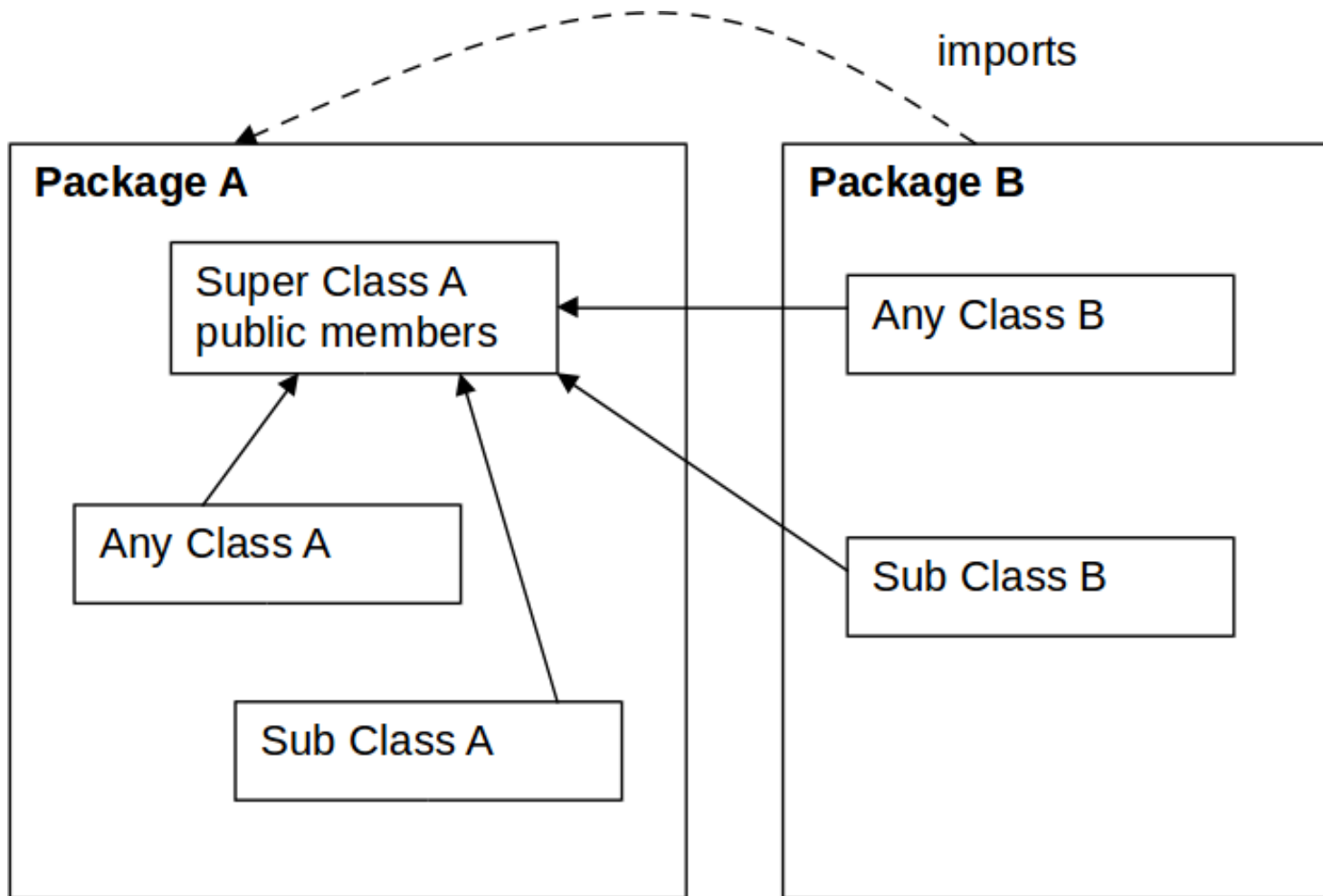
# Public access

- If we declare any variable and method as 'public', it will be accessible to all the entire class and visible to all the classes outside the class. It can be accessed from outside package also. For example,

```
public int val;
```

```
public void display();
```

# Public access



# Protected access

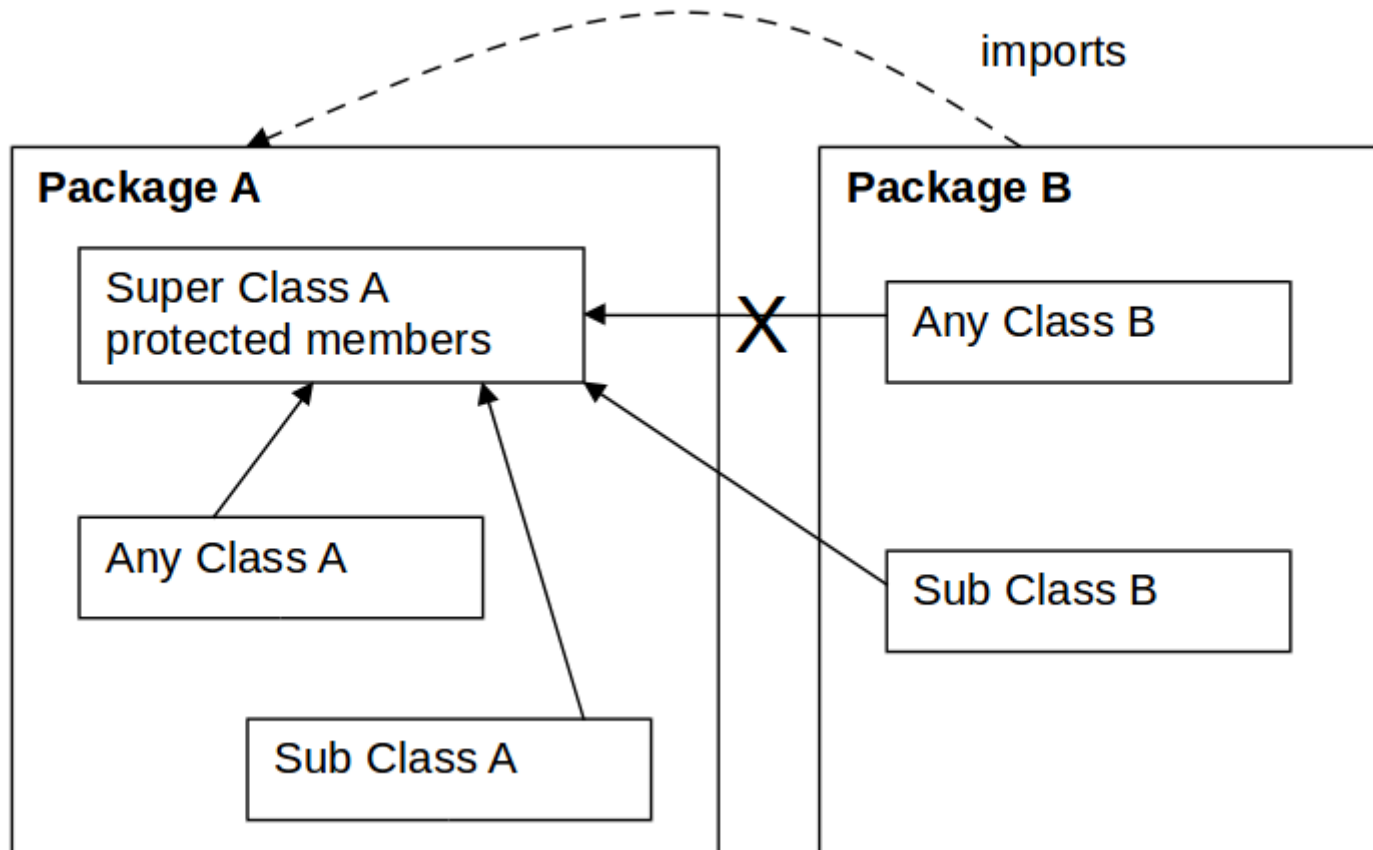
- A protected member is accessible in all classes in the package containing its class and by and by all subclasses of its class in any package where this class is visible.
- In other words, non-subclasses in other packages cannot access protected members from other packages. Means, this visibility control is strongly related to inheritance.

For example,

```
protected int val;
```

```
protected void display();
```

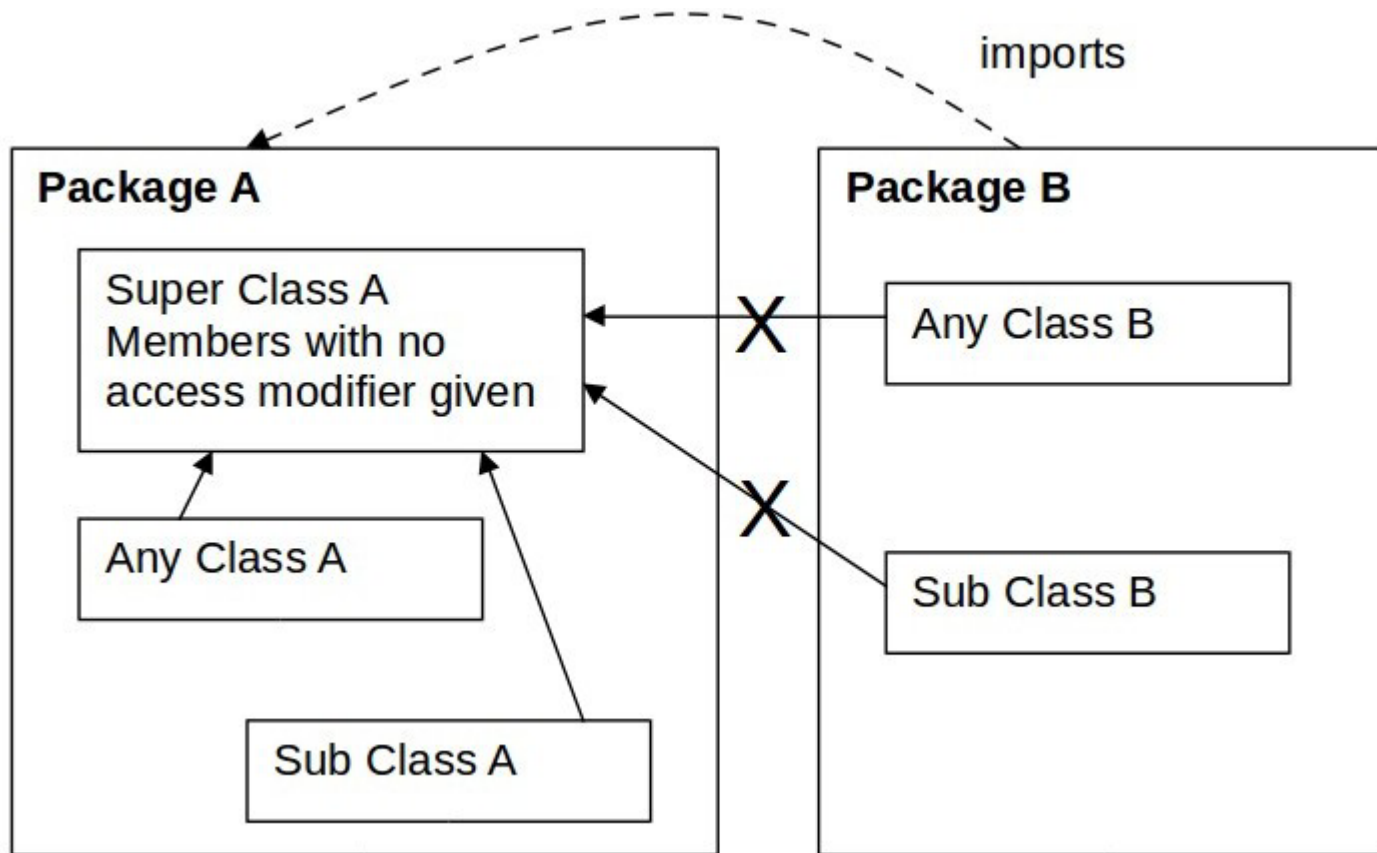
# Protected access



## Default access/package access/friendly access

- The default access can also be called as package access or friendly access.
- When no member accessibility modifier is specified, the member is only accessible inside its own package only.
- Even if its class is visible in another package, the member is not accessible there.
- Default member accessibility is more restrictive than the protected member accessibility.

# Default access





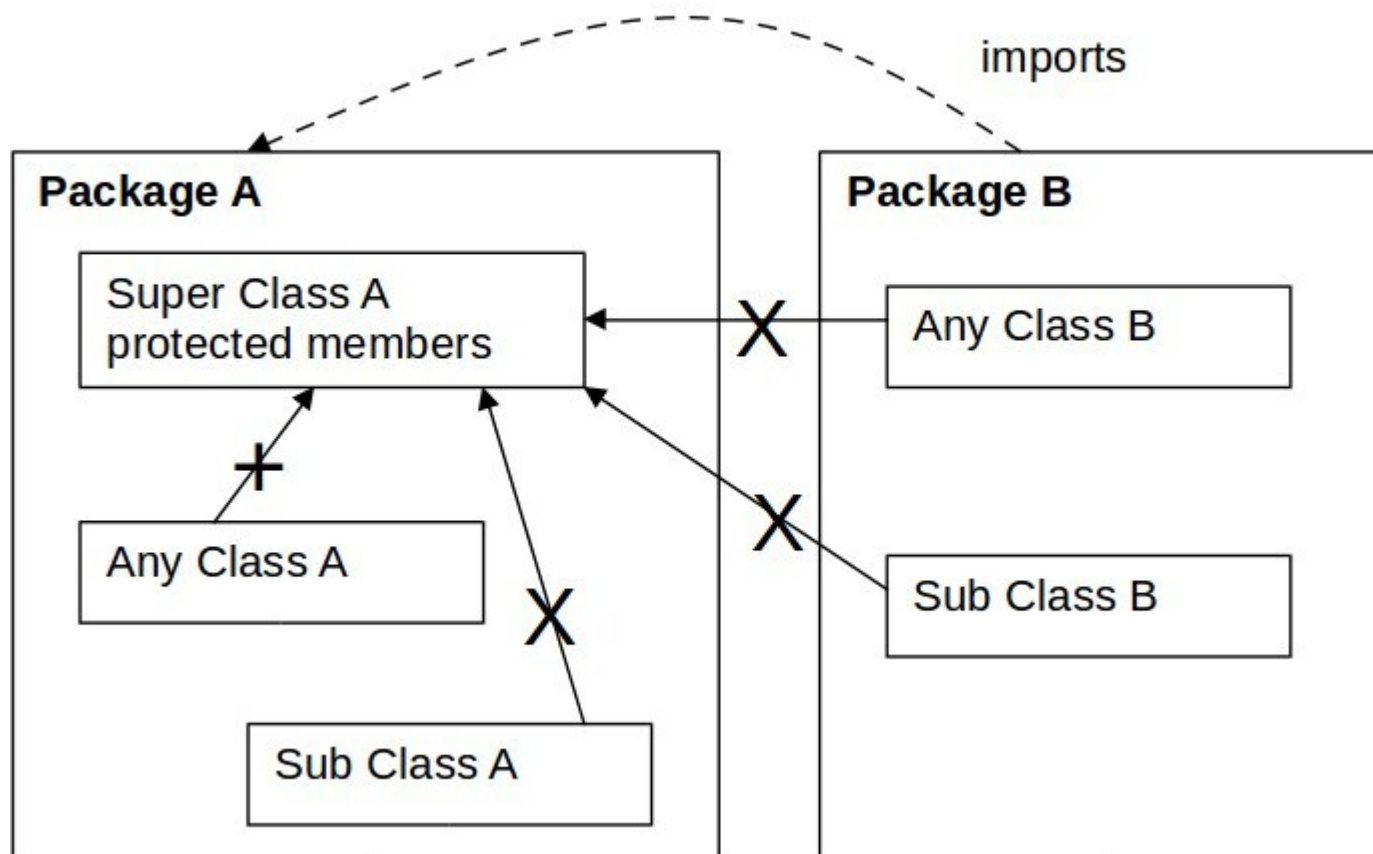
# Private access

- This is most restrictive than all other visibility controls.
- Private members are not accessible from any other class.
- They can not be accessed even from any subclass or any class from the same package. In order to declare a member as private, we just need to write 'private' in front of the member as,

```
private int val;
```

```
private void display();
```

# Private access



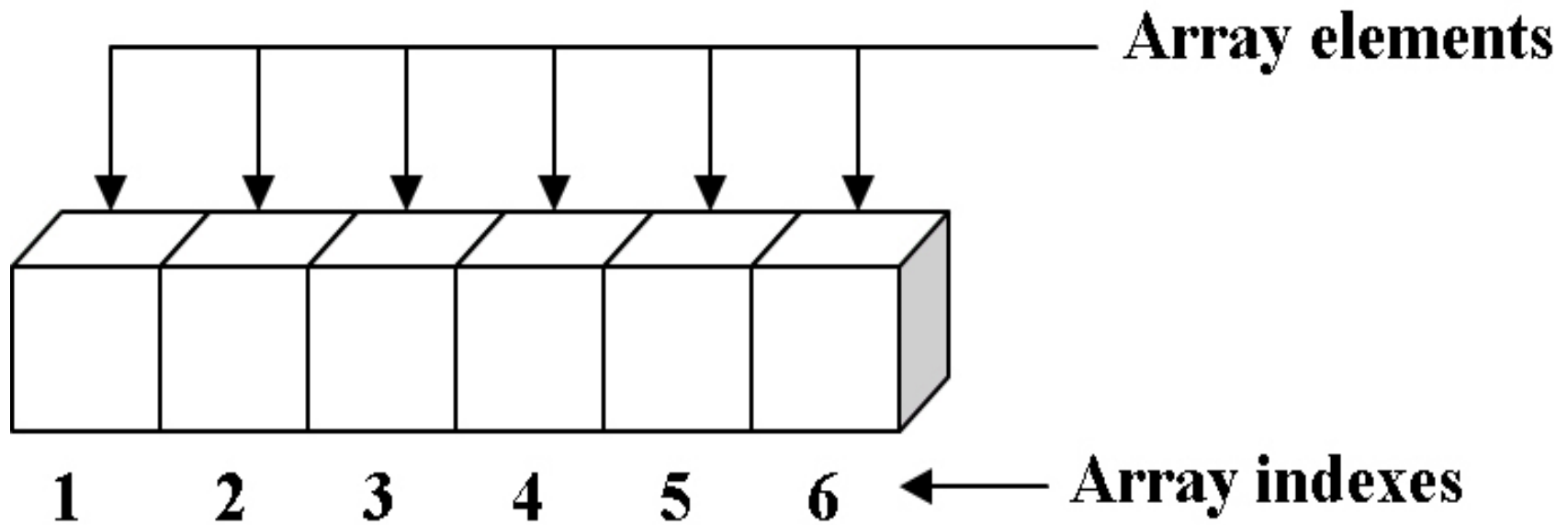
# Arrays

- An array is contiguous or related data items that share the common name.
- It offers the convenient means of grouping information. This means that all elements in the array have the same data type.
- A position in the array is indicated by a non-negative integer value called as index.
- An element at the given position is accessed by using this index.
- The size of array is fixed and can not increase to accommodate more elements.

# Types of array

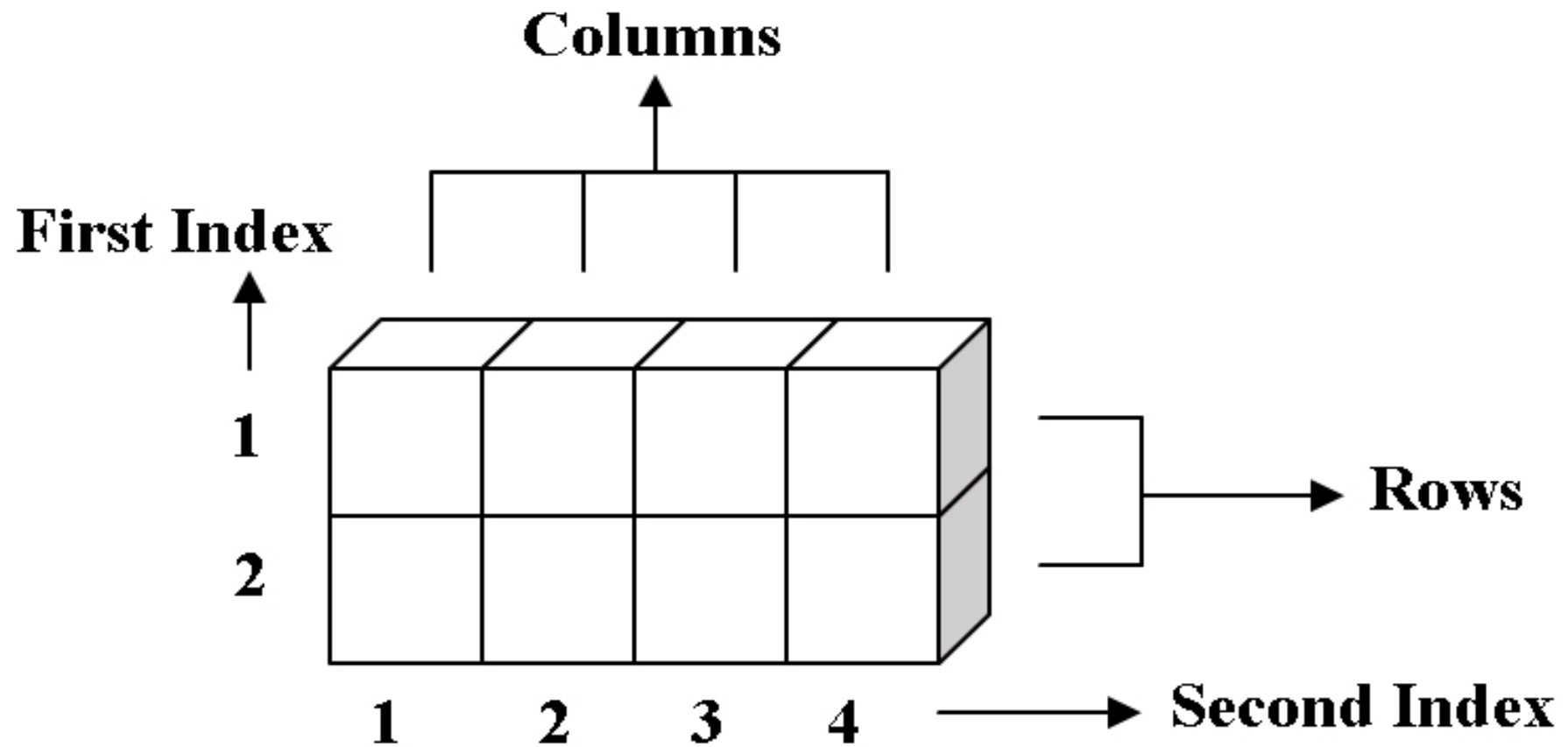
- One dimensional array
- Multidimensional  
array

# 1-D array



**One-dimensional array with six elements**

# 2-D array



**Two-dimensional array with eight elements**

# Array declaration

- One dimensional array is essentially, a list of same-typed variables. A list of items can be given one variable name using only one subscript. The general form of creating one dimensional array is,

*datatype variablename[ ] = new datatype[size];*

# Array declaration

```
int val[5] = new int[5];
```

```
int arr[] = {8, 6, 2, 4, 9, 3, 1};
```



# The strings

- In Java, a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String.
- Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.
- For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.

# Creating strings

- The String class has several constructors defined. To create an empty string the default constructor is used. For example:

```
String s = new String();
```

- It will create the instance of the string with no characters in it. If we want to initialize the string with values we can use following constructor.

```
String(char chars[])
```

- For example:

```
char chars[] = {'h', 'e', 'l', 'l', 'o'}; String s  
= new String(chars);
```

# String length

```
String s = "Pramod"; int len =  
s.length();
```

# String concat

```
String    s = "First";  
String    t = "Second";  
s.concat(t);
```

# Character at

```
String s = "Indian"; char ch = s.  
charAt(2);
```

```
//ch will be 'd' after this
```

# Equality

**String a = "Hello"; String b = "HELLO";**

**if(a.equals(b)) //false System.out.**

**println("Equals in case");**

**if(a.equalsIgnoreCase(b))//true**

**System.out.println("Equals in characters");**

# String Buffer

- String represents fixed-length, immutable character sequences.
- In contrast, StringBuffer represents growable and writeable character sequences.
- It is possible to insert the characters anywhere inside the StringBuffer.
- StringBuffer will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

# Creating StringBuffer

StringBuffer class defines three constructors:

different

**StringBuffer() StringBuffer(int  
size) StringBuffer(String str)**



# How to use StringBuffer?

```
StringBuffer sb = new StringBuffer("Oriya");
```

```
int len =      sb.length()); be 5
```

```
//len will
```

```
int cap =      sb.capacity()); be 21
```

```
//cap will
```

# Vector

- The package `java.util` contains a library of Java's utility classes.
- One of them is `Vector` class. It implements a dynamic array which can hold the array of any type and any number.
- These objects do not have to be homogenous. Capacity of the `Vector` can be increased automatically.

# Creation of Vector

Vector class defines three different constructors:

**Vector()**

**Vector(int           size)**

**Vector(int           size, int incr)**

# Operations of Vector

**void addElement(Object element) int  
capacity()**

**Object elementAt(int index) int size()**

**boolean removeElement(Object  
element)**

# Example:

```
Vector v = new Vector(3, 2); System.out.println("Initial size: " + v.size( )); System.out.println("Initial capacity: " + v.capacity( ));  
v.addElement(new Integer(23)); v.addElement(new Float(9.6f));
```

# Wrapper classes

- The primitive data types of Java such as int, char, float are not the part of any object hierarchy.
- They are always passed by value to the method not by reference.
- Object representations of primitive data types are called as wrapper classes.
- In essence, these classes encapsulate or wrap the primitive data types within the class. All the wrapper classes are defined in package java.lang.

# Wrapper classes

Data type	Wrapper class
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short
byte	Byte

# Example:

```
Character ch = new Character('X'); System.out.print("Char  
value: "); System.out.println(ch.charValue());
```

```
if(Character.isDigit('0')) System.out.println("0 is digit");
```

```
if(Character.isLowerCase('R')) System.out.println("R is  
lower case");
```



# Enumerated data type

- Enum is type like class and interface and can be used to define a set of Enum constants.
- Enum constants are implicitly static and final and you can not change there value once created.
- Enum in Java provides type-safety and can be used inside switch statment like int variables.

# Example:

```
public enum Color
{
    WHITE, BLACK, RED, YELLOW, BLUE;
}
```

# Inheritance

- Reusability is one of the most useful advantages of the Object Oriented Programming.
- Reusability means using something again and again without creating a new one.
- Java programming language supports this concept. The reusability can be obtained by creating new classes and reusing the properties of the existing one.
- The mechanism of deriving a new class from existing one is called as **inheritance**.
- The old class or existing class is known as super class and new class is known as subclass.
- Thus, a subclass is specialized version of super class. We can call super class as base class or parent class and subclass as derived class or child class.

# Types of inheritance

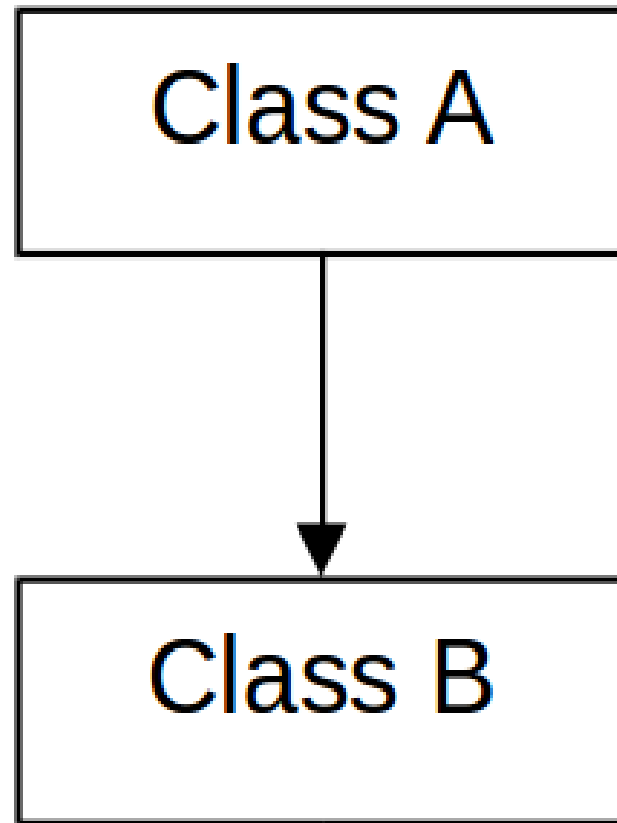
- Single Inheritance (only one super and subclass)
- Multiple Inheritance (several super classes for a subclass)
- Hierarchical Inheritance (one super class with many subclasses)
- Multilevel Inheritance (subclass is super class of another class)
- Hybrid Inheritance (combination of above three types)

# Creating inheritance

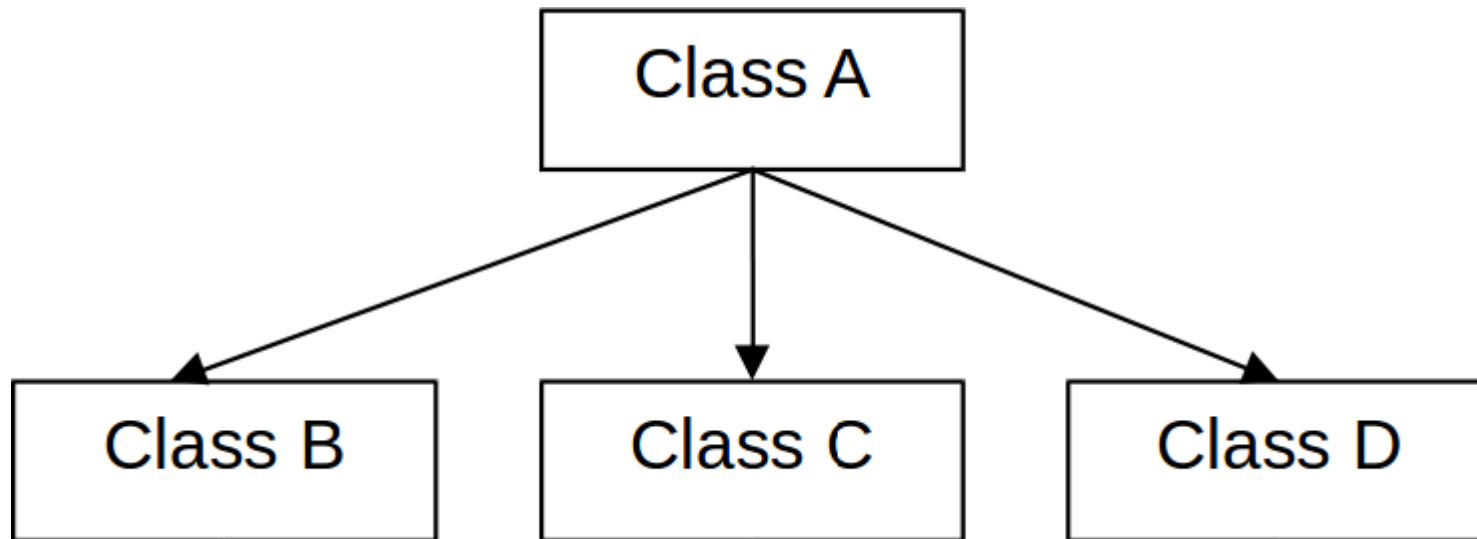
The class can be derived from another class by following the syntax:

```
class subclassname extends  
superclassname  
{  
    //Body of the class;  
}
```

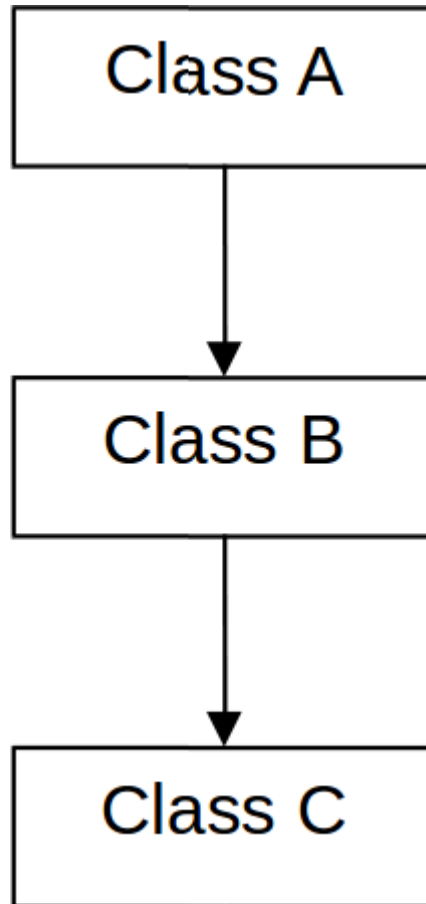
# Single inheritance



# Hierarchical inheritance

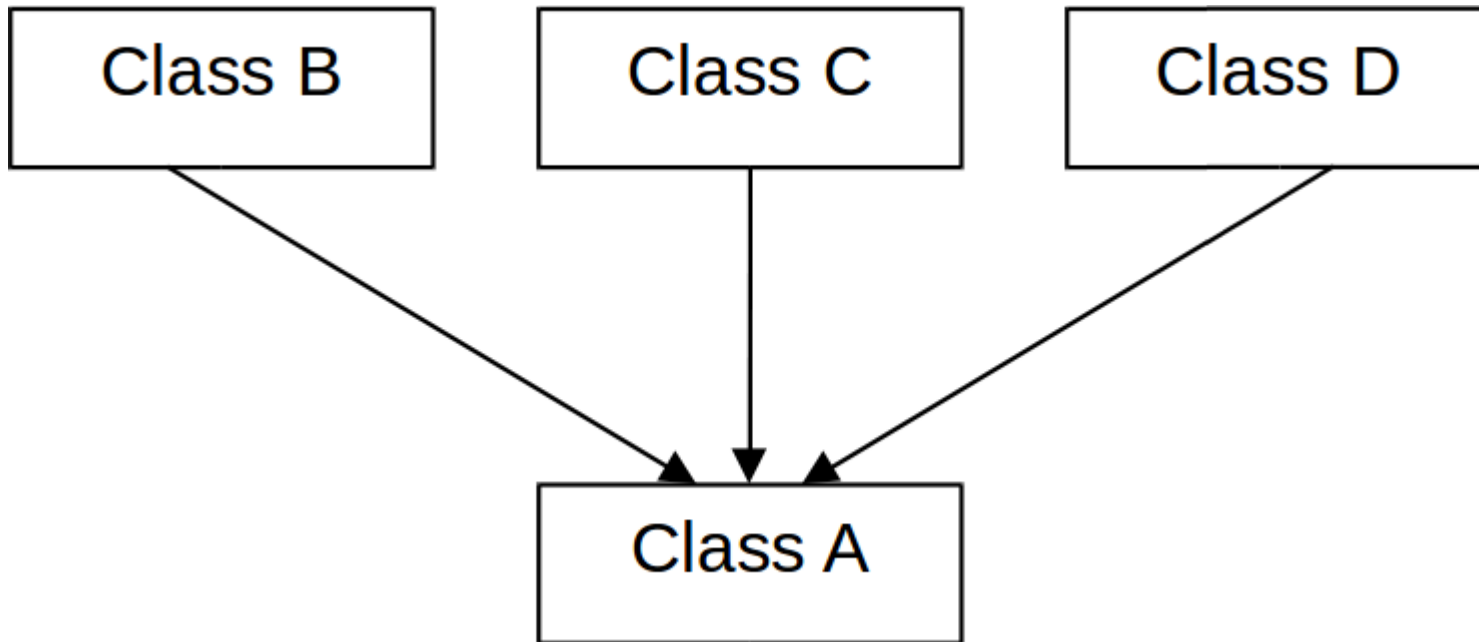


# Multilevel inheritance

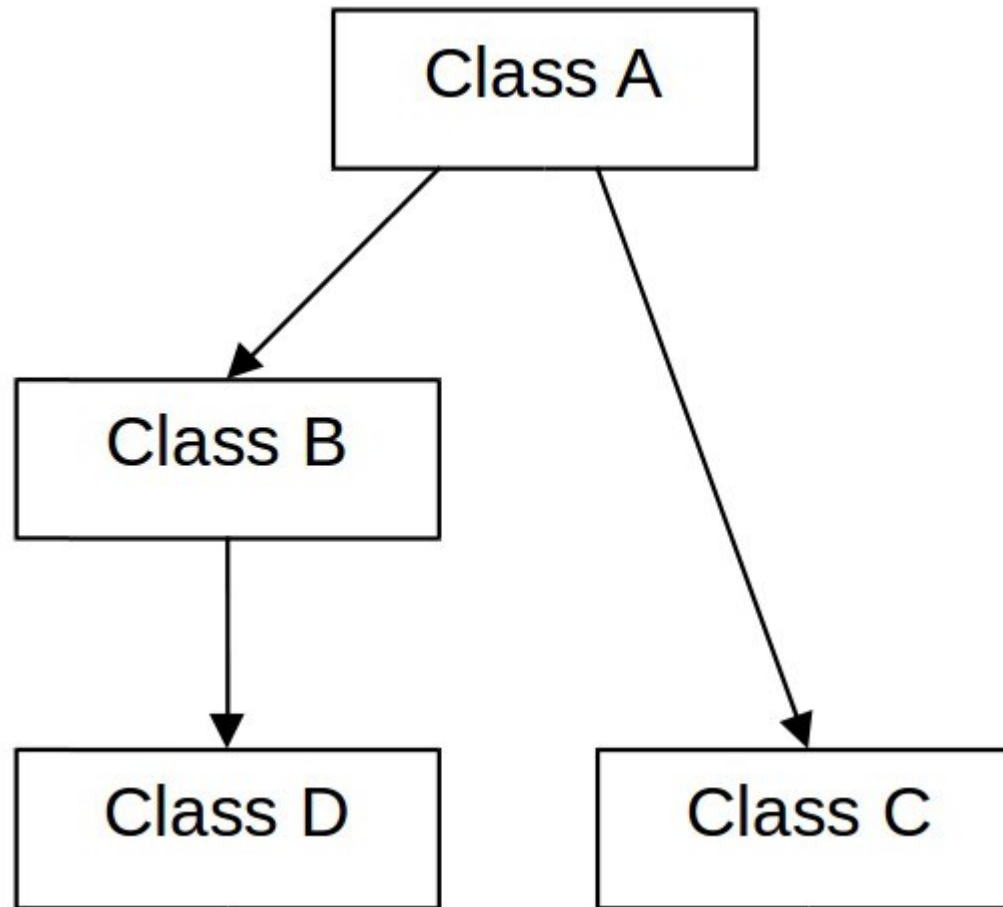




# Multiple inheritance



# Hybrid inheritance



# The 'super' keyword

- The keyword 'super' has specially been added for inheritance.
- Whenever the subclass needs to refer to its immediate super class, 'super' keyword is used. There are two different applications of the 'super'. That are,
  - Accessing members from super class that has been hidden by members of the subclass
  - Calling super class constructor

# Uses of 'super'

- **Accessing members from super class**
- In order to access the members from super class following form of the 'super' is used.

**super.variablename;**

- **Calling super class constructor**
- A subclass can call the constructor defined by super class by using the following form of the 'super' keyword.

**super(argumentlist);**

- The argument list specifies any arguments needed by constructor in the super class. Remember 'super ( )' must always be the first statement executed inside a subclass' constructor.

# Method overriding

- In a class hierarchy, when a method in a subclass has the same name, same arguments and same return type as a method in its super class, then the method in the subclass is said to override the method in the super class.
- 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 super class will be hidden.

# Example:

```
class Maths
{
    int var1, var2, var3; Maths(int x,
    int y)
    {
        var1 = x; var2 = y;
    }

    void calculate()           //statement1
    {
        var3 = var1 +         var2;
        System.out.println("Addition : "+var3);
    }
}

class Arithmetic extends Maths
{
    Arithmetic(int x,int y)
    {
        super(x,y);
    }

    void calculate()           //statement2
    {
        var3 = var1 -         var2;
        System.out.println("Subtraction : "+var3);
    }
}
```

# 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 the only way in which Java implements run-time polymorphism.
- If a super class contains a method that is overridden by a subclass, then when different types of objects are referred to through super class reference variable, different versions of the method are executed.

This concept is C++ counterpart of Virtual function in Java.

-

# Example:

```
class Principal {  
    void message(){ System.out.println("This  
        is Principal");  
}  
}  
class Professor extends Principal {  
    void message(){ System.out.println("This  
        is Professor");  
}  
}  
class Lecturer extends Professor {  
    void message(){ System.out.println("This  
        is Lecturer");  
}  
}
```



# Example continued...

```
class Dynamic {  
    public static void main(String args[]) {  
        Principal x = new Principal();  
        Professor y = new Professor();  
        Lecturer z = new Lecturer();  
        Principal ref; //reference variable  
                           of super class  
  
        ref = x; //statement1 ref.  
        message();  
  
        ref = y; //statement2 ref.  
        message();  
  
        ref = z; //statement3 ref.  
        message();  
    }  
}
```

# Abstract classes and methods

- Sometimes we will want to create a super class that only defines a generalized form that will be shared by all of its subclasses by leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement. This can be done by creating a class as an abstract.

# Abstract

- We can indicate that a method must always be redefined in a subclass by making the overriding compulsory.
- This can be achieved by preceding the keyword 'abstract' in the method definition with following form:

**abstract datatype methodname(parameters);**

# Abstract

```
abstract class White
{
    .....
    abstract void paint();
    .....
    .....
}
```

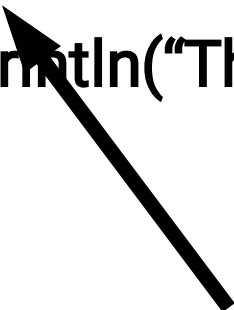
- We can not create the objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.

# Application of 'final' in inheritance

- Prevent the method overriding
- Prevent the inheritance

# Prevent method overriding

```
class First {  
    final void display() { System.out.println("This is first class");  
}  
  
}  
class Second extends void First {  
    display() {  
        System.out.println("This is second class");  
    }  
}
```



error

# Prevent inheritance

```
final class First
```

```
{
```

```
    // ....
```

```
}
```

```
class Second
```

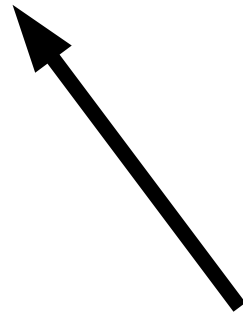
```
    extends
```

```
    First
```

```
{
```

```
    // ....
```

```
}
```



error

# References



## 222. Core Java Programming—A Practical Approach

*Tushar B. Kute*

### *Contents*

1. Introduction to Java; 2. Classes, Objects and Methods; 3. Arrays, Strings and Vectors; 4. Packages and Interfaces; 5. Exceptions and Multithreading; 6. Applets and Graphics Programming; 7. Streams of File I/O; Appendix; Index.

**ISBN:** 978-93-83828-29-6    **EDITION:** First, 2015    **SIZE:**  $7\frac{1}{4} \times 9\frac{1}{2}$



# Thank you



*This presentation is created using LibreOffice Impress 4.2.6.3 and  
available freely under GNU GPL*