Object Oriented Programming

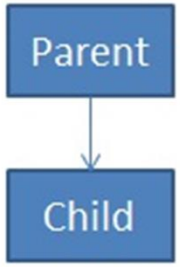
UNIT - II

Kanak Kalyani

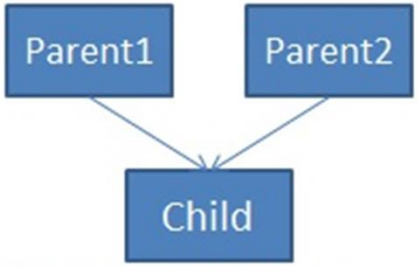
Inheritance

- **Inheritance** in java is used to add additional functionalities to the existing class. Inheritance is used to extend the present class by adding some more properties to it. Inheritance is used to reuse the present tried and tested code so that you may not have to write them and compile them again.

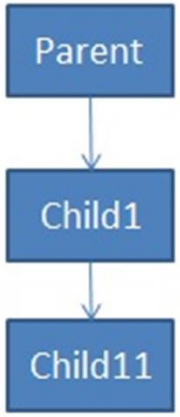
Types of Inheritance



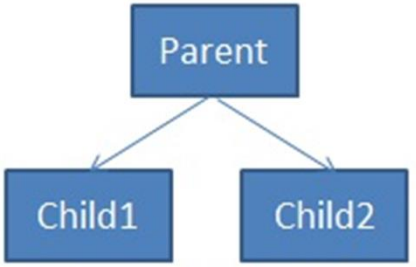
1. Single Inheritance



2. Multiple Inheritance

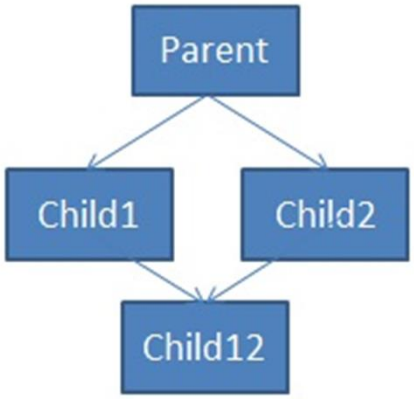


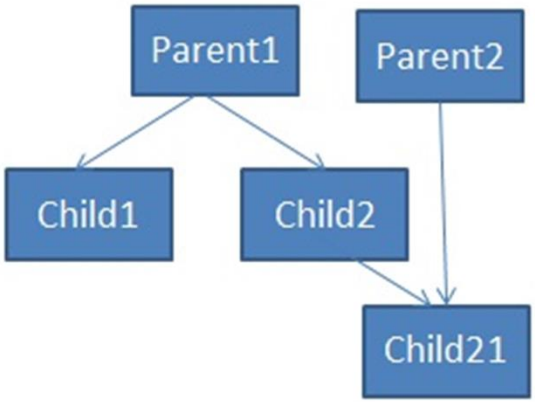
3. Multi-Level Inheritance



4. Hierarchical Inheritance

5. Hybrid Inheritance Variations (Mix of Single & Multiple Inheritance)





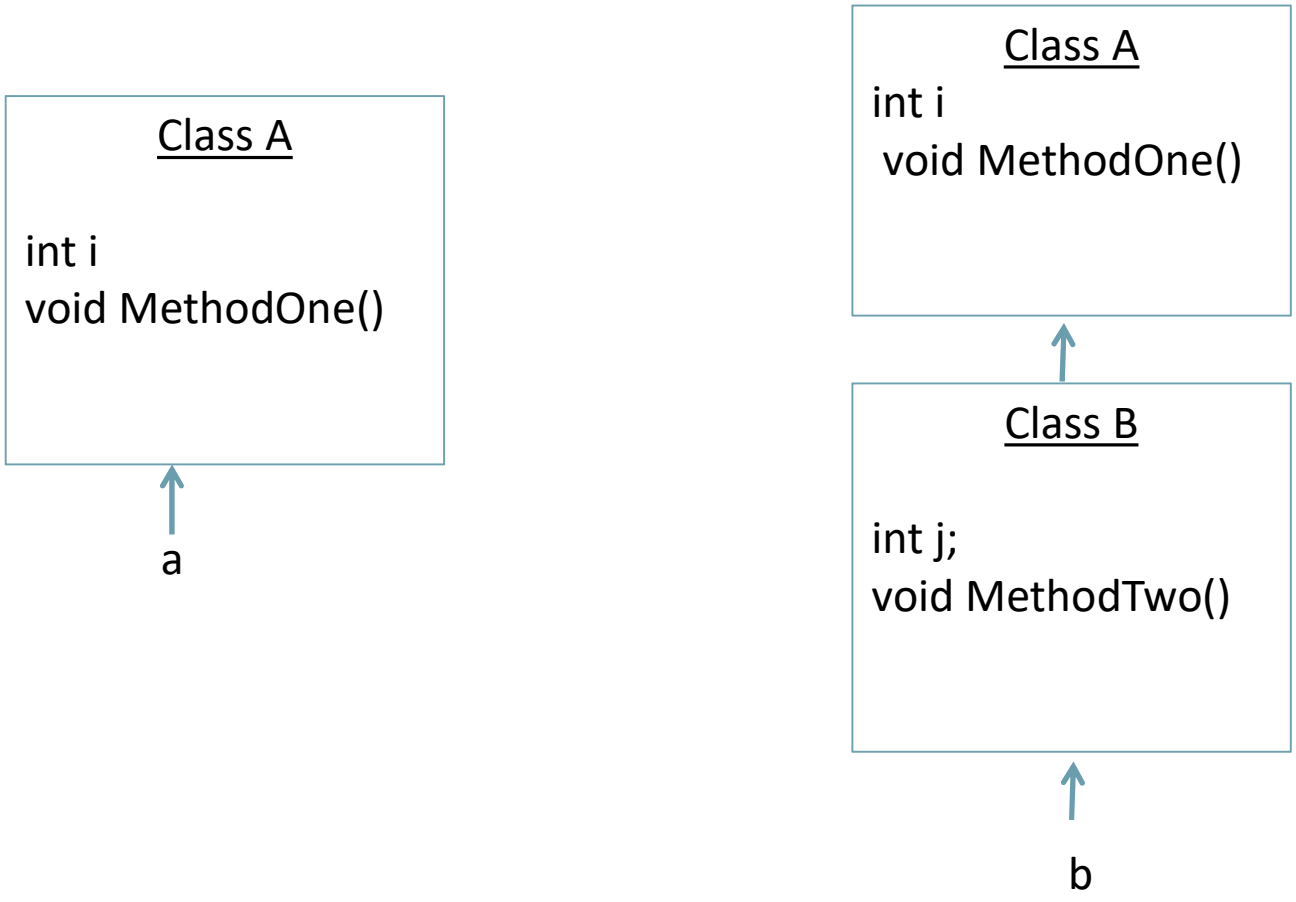
Note: The ones marked in red are not supported by Java

JavaLearningAcademy.com

Inheritance in java is implemented by using **extend** keyword like below,

```
class A
{
        int i;
        void methodOne()
        {
                System.out.println("From methodOne");
        }
}

class B extends A
{
        int j;
        void methodTwo()
        {
                System.out.println("From methodTwo");
        }
}

class MainClass{
        public static void main(String[] args){
                B b = new B();
                b.methodTwo();
                b.methodOne();
                b.i=10;
                b.j=20;
        }}
```

Memory Allocation



Example

```
class Box {
double width;
double height;
double depth;

// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}

// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}

// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

Example

```
// Here, Box is extended to include
weight.
class BoxWeight extends Box {
double weight; // weight of box
// constructor for BoxWeight
BoxWeight(double w, double h, double
d, double m) {
width = w;
height = h;
depth = d;
weight = m;
}
}
```

```
class DemoBoxWeight {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15,
34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4,
0.076);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " +
mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " +
mybox2.weight);
}
```

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

Example

```
// Here, Box is extended to include color.
class ColorBox extends Box {
int color; // color of box
ColorBox(double w, double h, double d, int c) {
width = w;
height = h;
depth = d;
color = c;
}
}
```

Super Keyword

```
class BoxWeight extends Box {

double weight; // weight of box

BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;

}
}
```

Super Keyword Example

```
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
weight = -1;
}
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}
```

Super Keyword Example

```
class DemoSuper {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);

double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " +
mybox1.weight);
System.out.println();

vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " +
mybox2.weight);
```

Super Keyword Example

```
vol = mybox3.volume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + my
System.out.println();
vol = myclone.volume();
System.out.println("Volume of myclone is " + vo
System.out.println("Weight of myclone is " + my
System.out.println();
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol
System.out.println("Weight of mycube is " + myc
System.out.println();
}
}
```

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076

Volume of mybox3 is -1.0
Weight of mybox3 is -1.0

Volume of myclone is 3000.0
Weight of myclone is 34.3

Volume of mycube is 27.0
Weight of mycube is 2.0
```

Super Keyword

super.*member*

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass

Using super to overcome name hiding.

```
class A {
int i;
void show() {
System.out.println("i in Class A " + i);
}

}
// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
void show() {
super.show();
System.out.println("i in B: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

Multilevel Hierarchy

```
class Shipment extends BoxWeight {
double cost;
// construct clone of an object
Shipment(Shipment ob) { // pass object to constructor
super(ob);
cost = ob.cost;
}
// constructor when all parameters are specified
Shipment(double w, double h, double d,
double m, double c) {
super(w, h, d, m); // call superclass constructor
cost = c;
}
// default constructor
Shipment() {
super();
cost = -1;
}
// constructor used when cube is created
Shipment(double len, double m, double c) {
super(len, m);
cost = c;
}
}
```

Multilevel Hierarchy

```
class DemoShipment {
public static void main(String args[]) {
Shipment shipment1 =
new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 =
new Shipment(2, 3, 4, 0.76, 1.28);
double vol;
vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is "
+ shipment1.weight);
System.out.println("Shipping cost: $" + shipment1.cost);
System.out.println();
vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is "
+ shipment2.weight);
System.out.println("Shipping cost: $" + shipment2.cost);
}
}
```

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

How Constructors are executed

```
class X
{
   X()
   {
      System.out.println("Inside X's constructor.");
   }
}
  // Create a subclass by extending class A.
class Y extends X
{
   Y()
   {
      System.out.println("Inside Y's constructor.");
   }
}
  // Create another subclass by extending B.
class Z extends Y
{
   Z()
   {
      System.out.println("Inside Z's constructor.");
   }
}
```

```
public class CallingCons
{
   public static void main(String args[])
   {
      Z z = new Z();
   }
}
```

OUTPUT

Inside X's constructor.
Inside Y's constructor.
Inside Z's constructor

Example

- Create a class person. Derive two classes' employee and student from class person. Members of class person are name, date of birth. Members of class employee are salary and designation. Members of class student are CGPA and branch. Use proper constructors and methods to store and display values. Write appropriate main() to test all the classes.

Method Overriding

Method Overriding :

```
class Xsuper
{
   int y;
   Xsuper(int y){
      this.y=y;
   }
   void display(){
      System.out.println("super y = " +y);
   }
}
class Xsub extends Xsuper
{
   int z;
   Xsub(int z , int y){
      super(y);
      this.z=z;
   }
   void display(){
      System.out.println("super y = " +y);
      System.out.println("sub z = " +z);
   }
}
 public class TestOverride{
   public static void main(String[] args)
   {
      Xsub s1 = new Xsub(100,200);
      s1.display();
   }
}
```

Method Overriding

Method Overriding :

```
class Xsuper
{
   int y;
   Xsuper(int y){
      this.y=y;
   }
   void display(){
      System.out.println("super y = " +y);
   }
}
class Xsub extends Xsuper
{
   int z;
   Xsub(int z , int y){
      super(y);
      this.z=z;
   }
   void display(){
            super.display();
      System.out.println("sub z = " +z);
   }
}
 public class TestOverride{
   public static void main(String[] args)
   {
      Xsub s1 = new Xsub(100,200);
      s1.display();
   }
}
```

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}

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);
k = c;
}

void show(String msg) {
    System.out.println(msg + k);
}
}}
class DEMO {
public static void main(String args[]) {

B subOb = new B(1, 2, 3);
subOb.show("This is k: ");
subOb.show();

}}
```

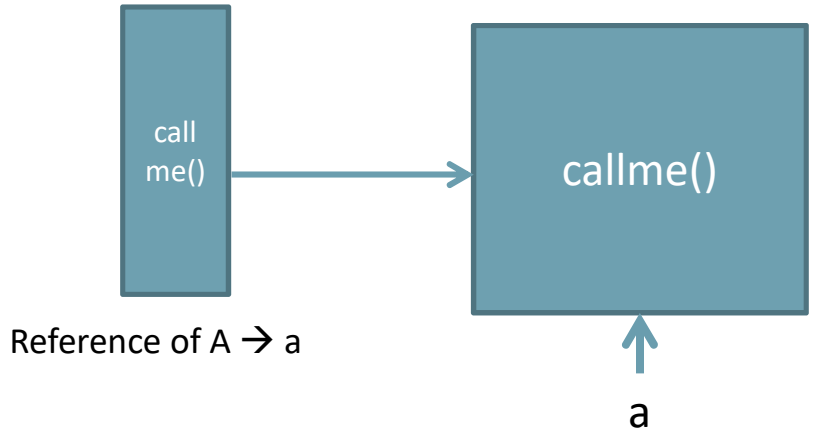
Run Time Polymorphism
Dynamic Method Dispatch

Output :

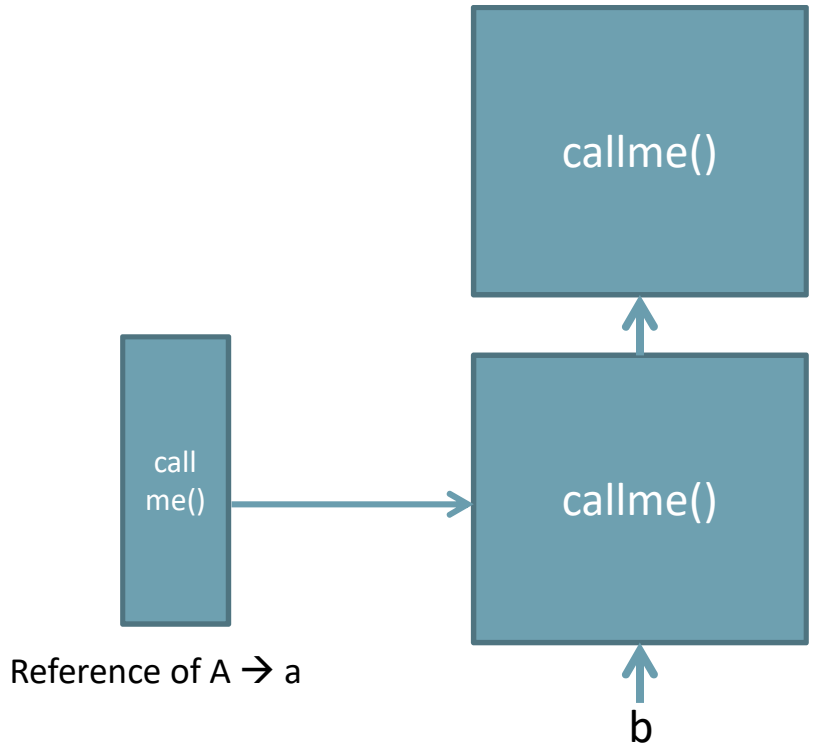
Inside A's callme method
Inside B's callme method
Inside C's callme method

```
class A
{
Int i=0;
   void callme()  {
      System.out.println("Inside A's callme
method");
   }
}

class B extends A
{      // override callme()

Int j=0;
void callme()  {
      System.out.println("Inside B's callme
method");
   }}
class C extends B {
Int k=0;
  // override callme()
  void callme()  {
   }
}
 public class TestDynamic
{
   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; // obtain a reference of type A
      r = a; // r refers to an A object
      r.callme(); // calls A's version of callme

      r = b; // r refers to a B object
      r.callme(); // calls B's version of callme

      r = c; // r refers to a C object
      r.callme(); // calls C's version of callme
   }
```

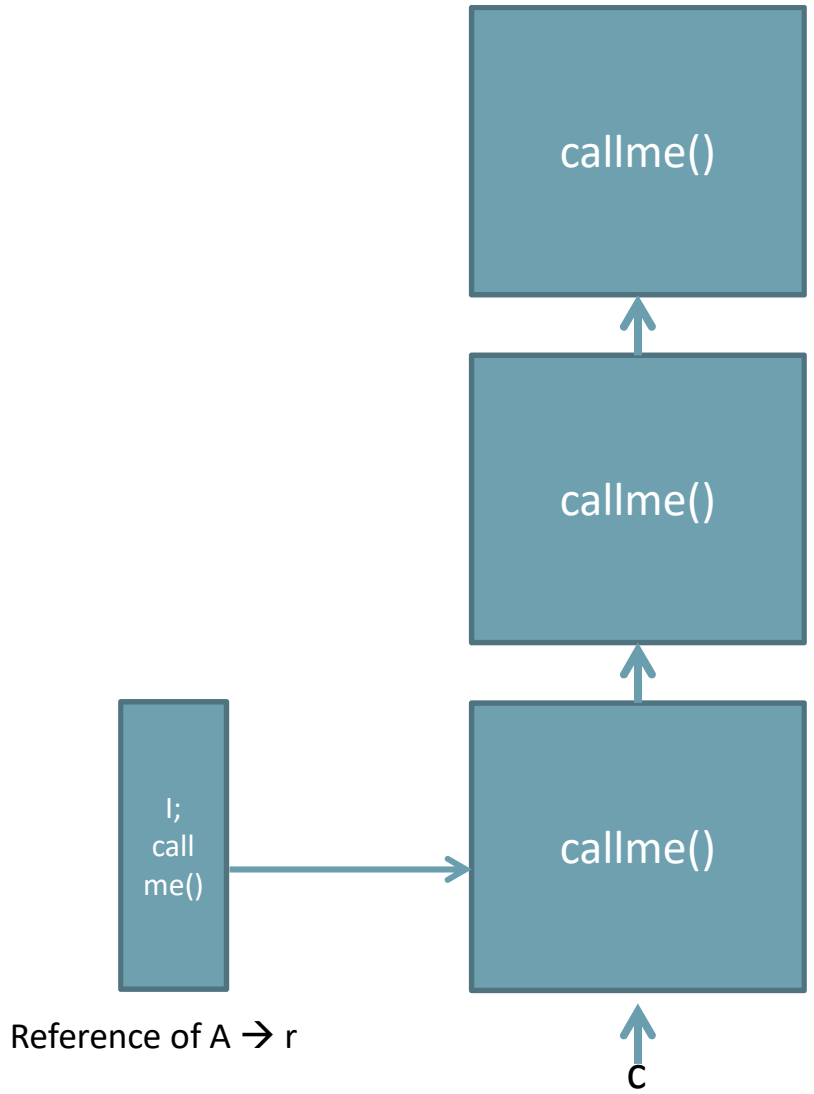
Dynamic Method Dispatch



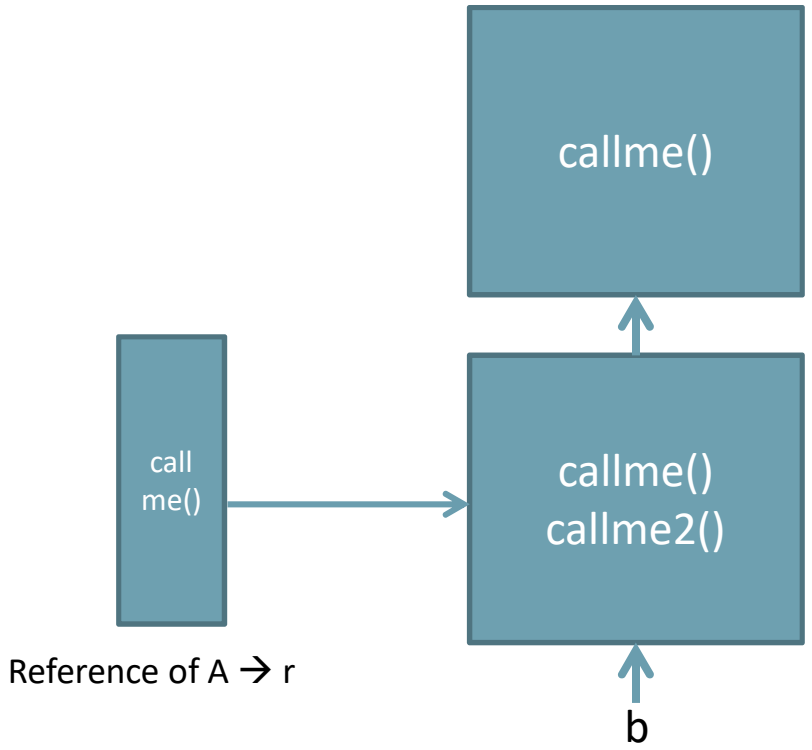
Dynamic Method Dispatch



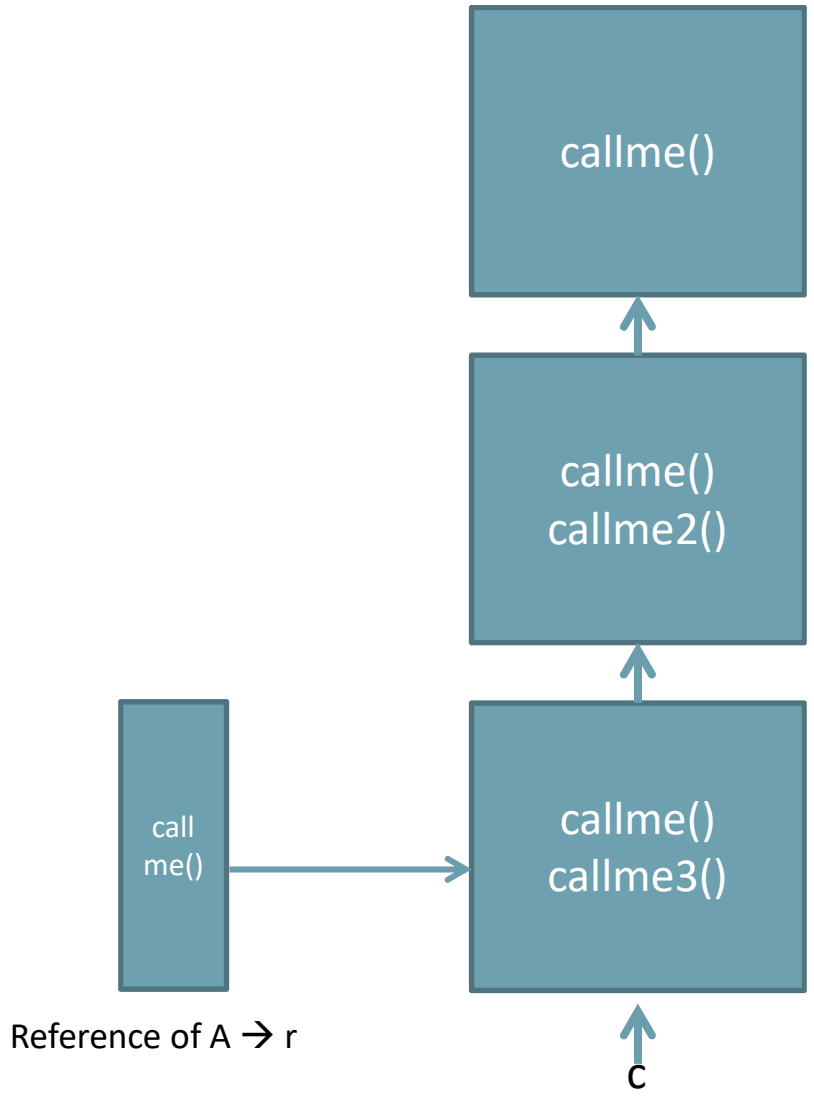
Dynamic Method Dispatch



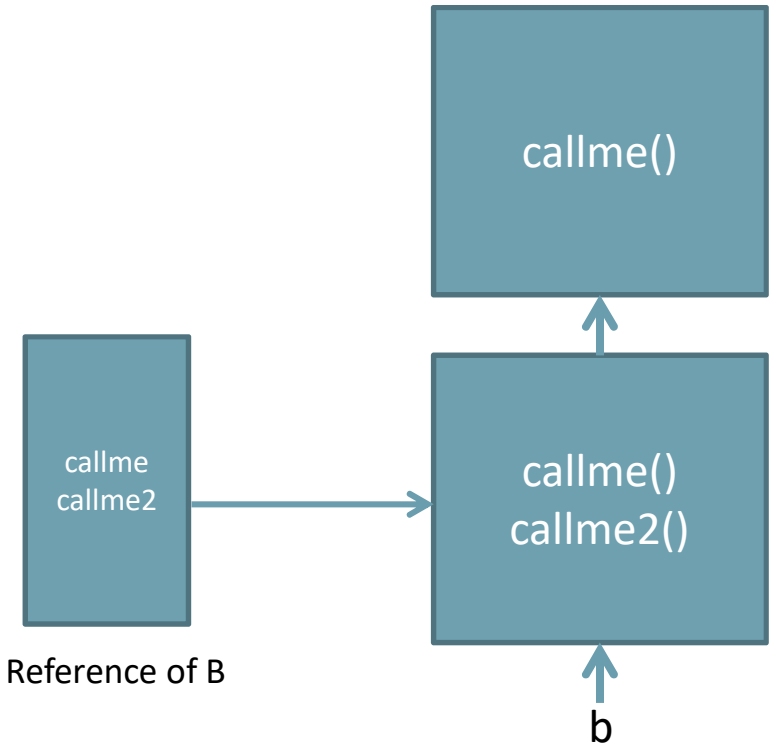
Dynamic Method Dispatch



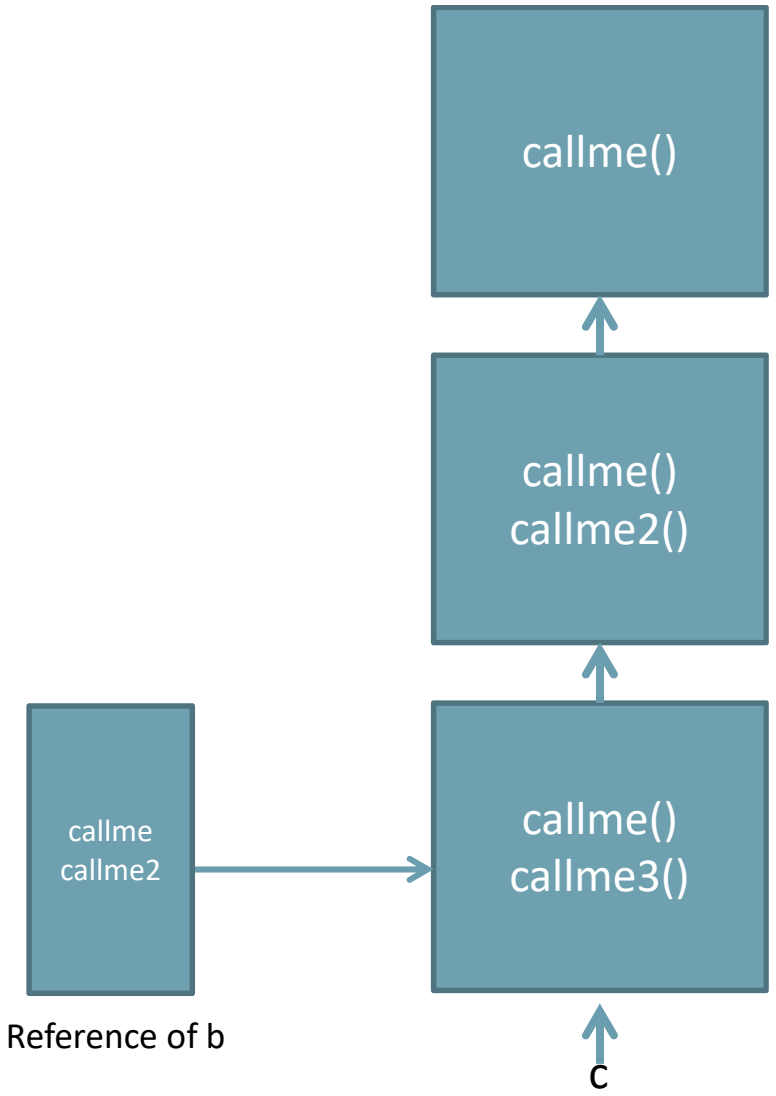
Dynamic Method Dispatch



Dynamic Method Dispatch



Dynamic Method Dispatch



- Create a class figure. Derive 2 classes rectangle and triangle. Create a method Area which will return area of their respective classes.
- Create an array of figure class and demonstrate dynamic method dispatch.

Example

```
// Using run-time polymorphism.
class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
double area() {
System.out.println("Area for Figure is
undefined.");
return 0;
}
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
}
}
return dim1 * dim2 / 2;
}
```

Example

```
class FindAreas {

public static void main(String args[]) {

Figure figref[]=new Figure[10];

figref [0]= new Rectangle(9, 5);

System.out.println("Area is " + figref[0].area());

figref [1]=new Triangle(10, 8);

System.out.println("Area is " + figref[1].area());

}
```

Do it Yourself

A publishing company that markets both book and audiocassette versions of its works.

Create a class publication that stores the title (a string) and price (type float) of a publication.

From this class derive two classes: book, which adds a page count (type int), and tape, which adds a playing time in minutes (type float).

Each of these three classes should have a setdata() function to initialize the data members and a display() function to display its data.

Write a main() program to test the book and tape classes by creating instances of them.

Write a program to demonstrate the concept of Dynamic Polymorphism, for the publishing company.

Final Keyword

***final variable*:** When we want to declare constant variable in java we use final keyword.

Syntax : final variable name = value;

final method: When we put final keyword before method than it becomes final method.

To prevent overriding of method final keyword is used, means final method can't be override.

Syntax: final methodname(arg)

final class: To prevent overriding of method final keyword is used, means final method can't be override.

Syntax : final class class_name { ... }

Final Variable

```
class aa
{
   final int a=10;
   public void ainc()   {
      a++;   // The final field aa.a cannot be assigned
   }
}

public class Final_Demo
{
   public static void main(String[] args)
   {
      aa ob = new aa();
      aa.ainc();
   }
}
```

Final Method

```
public class aa
{
   int a=10;
   public final void ainc()  {
      a++;  // The final field aa.a
cannot be assigned
   }
}

class bb extends aa{
   public void ainc()  //Cannot
override the final method from aa
   {
      System.out.println("a = " + a);
   }
}
```

```
public class Final_Demo
{
   public static void main(String[]
args)
   {
      bb b1 = new bb();
      b1.ainc();
   }
}
```

Final Class

```
final class aa{
    int a=10;
   public void ainc()   {
      a++;   // The final field aa.a cannot be assigned
   }
}
class bb extends aa{    // The type bb cannot subclass the final class aa
   public void ainc()
   {
      System.out.println("a = " + a);
   }
}
public class Final_Demo {
   public static void main(String[] args)
   {
      bb b1 = new bb();
      b1.ainc();
   }}
```

Abstract Keyword

Sometimes there are scenarios in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

So there is a need to define a generalized method which all subclasses will share, but doesn't need an implementation.

Java's solution to this problem is the *abstract method*.

Abstract means INCOMPLETE

Java defines abstract methods and abstract classes.

Example

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

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

Example - Figure

```
// Using abstract methods and classes.
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);
}
@override
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
.
```

```
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
@override
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas {
public static void main(String args[]) {

// Figure f = new Figure(10, 10); // illegal now

Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);

Figure figref; // this is OK, no object is created
figref = r;

System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
} }
```

Interfaces

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- That is, using **interface**, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

Interface - Syntax

access interface *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;
}

Beginning with JDK 8, it is possible to add a *default implementation* to an interface method

Interface- Example

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

class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
    System.out.println("callback called with " + p);
    }

void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
    "may also define other members, too.");
    }
}
```

Interface- Example

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

```
class TestIface {
public static void main(String args[]) {
Callback c1 = new Client();
c1.callback(42);
}
}
```

Interface- Example

```
// Another implementation of Callback.
class AnotherClient implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
    System.out.println("Another version of callback");
    System.out.println("p squared is " + (p*p));
    }
}

class TestIface2 {
    public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();
    c.callback(42);
    c = ob; // c now refers to AnotherClient object
    c.callback(42);
    }
}
```

```
callback called with 42
Another version of callback
p squared is 1764
```

Interface – Partial Implementation

```
abstract class Incomplete implements Callback {
int a, b;
void show() {
System.out.println(a + " " + b);
}
//...
}
```

Interfaces Can Be Extended

```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

interface B extends A {
  void meth3();
}

class MyClass implements B {
  public void meth1() {
    System.out.println("Implement
meth1().");
  }

  public void meth2() {
    System.out.println("Implement
meth2().");
  }

  public void meth3() {
    System.out.println("Implement
meth3().");
  }
}

class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();
    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

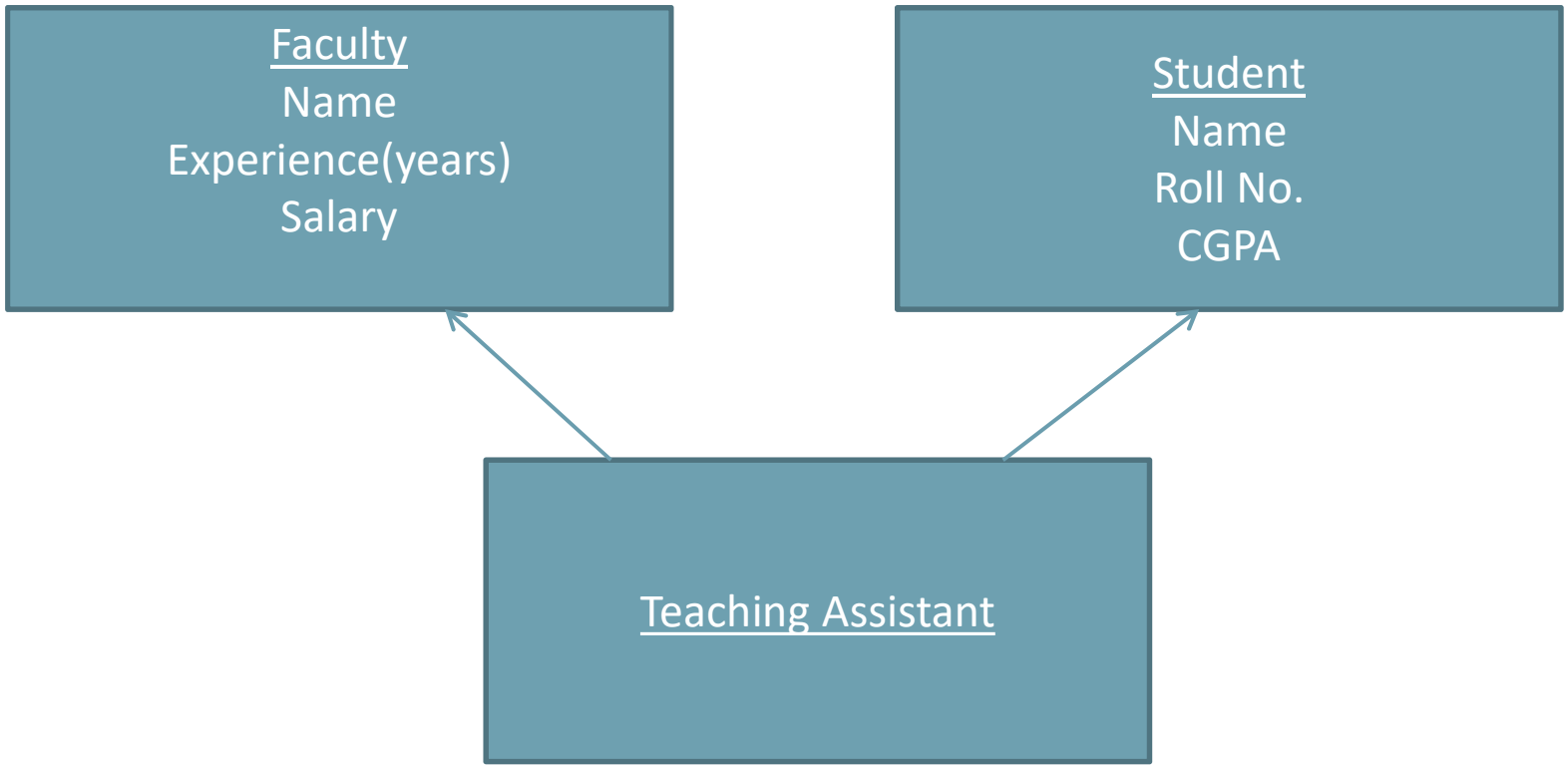
Example - Figure

```
// Using abstract methods and classes.
interface Figure {
double area();
}


class Rectangle implements Figure {
double l,b;
Rectangle(double a, double b) {
l=a;
this.b=b
}
@override
public double area() {
System.out.println("Inside Area for Rectangle.");
return l*b;
}
}

class Triangle extends Figure {
double dim1;
double dim2;
Triangle(double a, double b) {
dim1 = a;
dim2 = b;
}
@override
public double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class InterfaceExample {
public static void main(String args[]) {

Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
} }
```

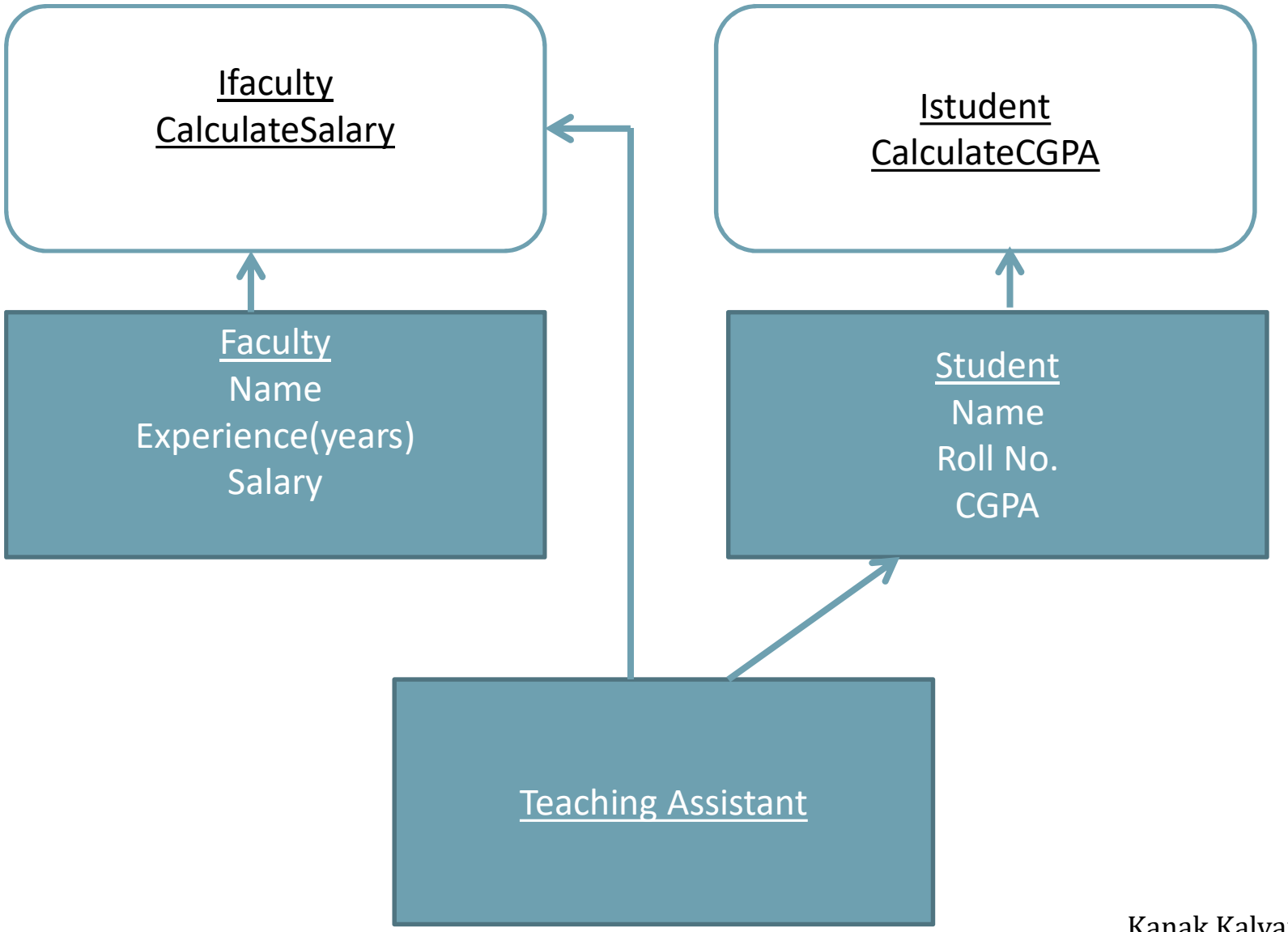
Do it Yourself

- Consider a class Employee with data members as employee id and employee name. Create an interface taxable which has method calculateTax().
- Derive a class permanent from class employee and interface taxable. Data member for permanent is Salary, include function AdditionalEarning which stores information if there is additional earning. Calculate tax for the permanent employee. If salary for an employee is between 1 lakh to 5 lakh tax is 10%. If salary is more than 5 lakh tax is 20%. Add a function to display the salary and tax.
- Derive a class HourlyEmployee from Employee. Data members are HoursWorked, RatePerHour.
- Include methods to calculate salary and display it. Write proper constructors and display methods for all the classes. Write a main() to demonstrate use of all the classes.

Multiple Inheritance



Multiple Inheritance



Marker Interface

Marker Interface in java is an **interface** with no fields or methods within it.

It is used to convey to the JVM that the class implementing an **interface** of this category will have some special behavior.

Hence, an empty **interface in java** is called a **marker interface.**

In java we have the following major marker interfaces as under:

Serializable interface

Cloneable interface

Remote interface

ThreadSafe interface

Syntax:

```
interface i
{

}
```

Functional Interfaces

A functional interface is an interface that specifies only one abstract method

```
interface MyNumber {
  double getValue();
}
```

Anonymous Class

While local classes are class declarations, anonymous classes are expressions, which means that you define the class in another expression.

```
interface HelloWorld {
      public void greet();
      public void greetSomeone(String someone);
   }

HelloWorld frenchGreeting = new HelloWorld() {
         String name = "tout le monde";
         public void greet() {
            greetSomeone("tout le monde");
         }
         public void greetSomeone(String someone) {
            name = someone;
            System.out.println("Salut " + name);
         }
      };


ABC ob=new ABC();
```

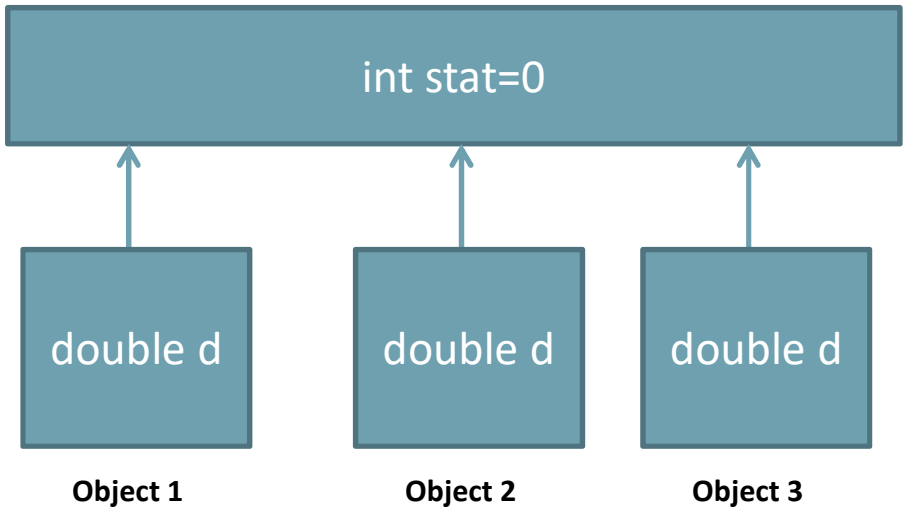
Static Keyword

- Static gives a mechanism to use a variable or method without creating object.
- Static Keyword can be used with METHODS and VARIABLES.
- Static methods and variables are called with the class name

Static Variables

- Instance variables declared as **static** are, essentially, **global variables**.
- When objects of its class are declared, no copy of a **static** variable is made.
- Instead, all instances of the class share the same **static** variable.

```
class A{
static int stat;
double d;
}
static void meth(){
stat=0;
}
```



Static Method

- Methods declared as **static** are, essentially, **global methods**
- Static methods are called by class name.

Methods declared as **static** have several restrictions:

- They can only directly call other **static** methods.
- They can only directly access **static** data.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance)

Example

```
class UseStatic {
    static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
    }
    Class demo{
    public static void main(String args[]) {
        UseStatic.meth(42);
    }
}
```

Example

```
Class Demostat {
    static int a = 3;
    static int b;
    static c;
    double d;
    Demostat(){
        c++;
        d=-1;
    }
    Demostat(int d){
        c++;
        this.d=d;
    }
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        // System.out.println(“d = " + d);//gives
        error

    }
    static {
        System.out.println("Static block
        initialized.");
        b = a * 4;
        }
        void nonstat(int x){
            System.out.println("x = " + x);
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            System.out.println(“d = " + d);
        }
}
class maindemo{
public static void main(String args[]) {
        Demostat.meth(42);
        Demostat o1=new Demostat();
        Demostat o2=new Demostat(10);
        System.out.println(“Objects Created are:
        “+Demostat.c);
        o1.nonstat(10);
    }
}
```

Scanner Class

- **Scanner** can be used to read input from the console, a file, a string, or any source that implements the Readable interface or ReadableByteChannel

- Scanner conin = new Scanner(System.in);

In general, to use **Scanner**, follow this procedure:

1. Determine if a specific type of input is available by calling one of **Scanner**'s **hasNext***X* methods, where *X* is the type of data desired.
2. If input is available, read it by calling one of **Scanner**'s **next***X* methods.
3. Repeat the process until input is exhausted.
4. Close the **Scanner** by calling **close( )**.

```
import java.util.*;
public class ScannerExample {
public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.next();
        System.out.println("Name is: " + name);
        in.close();
        }
}
```

Methods in Scanner Class

Modifier & Type	Method	Description
void	close()	It is used to close this scanner.
boolean	hasNext()	It returns true if this scanner has another token in its input.
boolean	hasNextBoolean()	It is used to check if the next token in this scanner's input can be interpreted as a Boolean using the nextBoolean() method or not.
boolean	hasNextByte()	It is used to check if the next token in this scanner's input can be interpreted as a Byte
boolean	hasNextDouble()	It is used to check if the next token in this scanner's input can be interpreted as a Double using the nextDouble() method or not.
boolean	hasNextFloat()	It is used to check if the next token in this scanner's input can be interpreted as a Float using the nextFloat() method or not.
boolean	hasNextInt()	It is used to check if the next token in this scanner's input can be interpreted as an int using the nextInt() method or not.
boolean	hasNextLine()	It is used to check if there is another line in the input of this scanner or not.
boolean	hasNextLong()	It is used to check if the next token in this scanner's input can be interpreted as a Long using the nextLong() method or not.
boolean	hasNextShort()	It is used to check if the next token in this scanner's input can be interpreted as a Short using the nextShort() method or not.

Methods in Scanner Class

String	next()	It is used to get the next complete token from the scanner which is in use.
boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
byte	nextByte()	It scans the next token of the input as a byte.
double	nextDouble()	It scans the next token of the input as a double.
float	nextFloat()	It scans the next token of the input as a float.
int	nextInt()	It scans the next token of the input as an Int.
String	nextLine()	It is used to get the input string that was skipped of the Scanner object.
long	nextLong()	It scans the next token of the input as a long.
short	nextShort()	It scans the next token of the input as a short.
void	remove()	It is used when remove operation is not supported by this implementation of Iterator.

```
import java.util.*;
public class ScannerClassExample1 {
        public static void main(String args[]){
                String s = "Hello, This is Java";
                Scanner scan = new Scanner(s);
                System.out.println("Boolean Result: " + scan.hasNext());
                System.out.println("String: " +scan.nextLine());
                scan.close();
                System.out.println("--------Enter Your Details---");
                Scanner in = new Scanner(System.in);
               System.out.print("Enter your name: ");
               String name = in.next();
               System.out.println("Name: " + name);
               System.out.print("Enter your age: ");
               int i = in.nextInt();
               System.out.println("Age: " + i);
               System.out.print("Enter your salary: ");
               double d = in.nextDouble();
               System.out.println("Salary: " + d);
              in.close();
                }
 }
```

String Class

Creating a String

There are two ways to create string in Java:

String literal String s = “Welcome to OOP Lecture”;

Using *new* keyword String s = new String (“Welcome to OOP Lecture”);

Some Constructors defined in String Class

- **String(char[] char_arr)**

char char_arr[] = {‘H', 'e', ‘l', ‘l', ‘o'};

String s = new String(char_arr);

- **String(char[] char_array, int start_index, int count)**

char char_arr[] = {‘H', 'e', ‘l', ‘l', ‘o'};

String s = new String(char_arr , 1, 3);//ell

String s= “HelloWorld”
String s=new String(“HelloWorld”)

- **int length():** Returns the number of characters in the String.
- "HelloWorld".length(); // returns 10
- **Char charAt(int i):** Returns the character at i^{th} index.
- "HelloWorld".charAt(3);
- **String substring (int i):** Return the substring from the i^{th} index character to end.
  "HelloWorld".substring(3);
- **String substring (int i, int j):** Returns the substring from i to j-1 index.
- "HelloWorld".substring(2, 5);
- **String concat(String str):** Concatenates specified string to the end of this string.
- String s1 = ”Hello”; String s2 = ”World”;
- String output = s1.concat(s2); // returns “HelloWorld”
- **int indexOf (String s):** Returns the index within the string of the first occurrence of the specified string.
- String s = ”Learn Share Learn”;
- int output = s.indexOf(“Share”); // returns 6
- **int indexOf (String s, int i):** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.
- String s = ”Learn Share Learn”;
- int output = s.indexOf("ea",3);// returns 13

- **Int lastIndexOf(String s):** Returns the index within the string of the last occurrence of the specified string.
- String s = ”Learn Share Learn”;
- int output = s.lastIndexOf("a"); // returns 14
- **boolean equals(Object otherObj):** Compares this string to the specified object.
- Boolean out = “Hello”.equals(“Hello”); // returns true
- Boolean out = “hello”.equals(“Hello”); // returns false
- **boolean equalsIgnoreCase (String anotherString):** Compares string to another string, ignoring case considerations.
- Boolean out= “Hello”.equalsIgnoreCase(“Hello”); // returns true
- Boolean out = “hello”.equalsIgnoreCase(“Hello”); // returns true
- **int compareTo(String anotherString):** Compares two string lexicographically.
- int out = s1.compareTo(s2);
- **int compareToIgnoreCase(String anotherString):** Compares two string lexicographically, ignoring case considerations.
- int out = s1.compareToIgnoreCase(s2);

- **String toLowerCase():** Converts all the characters in the String to lower case.
- String word1 = “HeLLo”;
- String word3 = word1.toLowerCase(); // returns “hello"
- **String toUpperCase():** Converts all the characters in the String to upper case.
- String word1 = “HeLLo”; String word2 = word1.toUpperCase(); // returns “HELLO”
- **String trim():** Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.
- String word1 = “ Learn Share Learn “;
- String word2 = word1.trim(); // returns “Learn Share Learn”
- **String replace (char oldChar, char newChar):** Returns new string by replacing all occurrences of *oldChar* with *newChar.*
- String s1 = “HelloWorld“;
- String s2 = “HelloWorld”.replace(‘l’ ,’Q’); // returns “HeQQoWorQd”

Lambda expressions significantly enhance Java because of two primary reasons.

1) They add new syntax elements that increase the expressive power of the language. In the process, they streamline the way that certain common

constructs are implemented.

2) Addition of lambda expressions resulted in new capabilities being incorporated into the API library.

a) advantage of the parallel processing capabilities of multi-core environments,
b) handling of for-each style operations,
c) Support for pipeline operations on data
d) Provides catalyst for other new Java features, including the default method which lets you define default behavior for an interface method, and the method references

Lambda Expressions

lambda operator is **->**

Java defines two types of lambda bodies:

1) consists of a single expression **() -> 123.45**
2) consists of a block of code

    double **myMeth() { return 123.45; }**

3) Method without name

    () -> Math.random() * 100

4) (n) -> (n % 2)==0

Functional Interfaces

```
interface MyNumber {
  double getValue();
}

MyNumber myNum;
myNum = () -> 123.45;
System.out.println(myNum.getValue());

myNum = () -> Math.random() * 100;

ERROR!!
myNum = () -> "123.03"; // Error!
```

EXAMPLE 2:

```
interface NumericTest {
 boolean test(int n);
}

class LambdaDemo2 {
  public static void main(String args[])
  {
    // A lambda expression that tests if a number is even.
    NumericTest isEven = (n) -> (n % 2)==0;

    if(isEven.test(10)) System.out.println("10 is even");
    if(!isEven.test(9)) System.out.println("9 is not even");


    // Now, use a lambda expression that tests if a number
    // is non-negative.
    NumericTest isNonNeg = (n) -> n >= 0;

    if(isNonNeg.test(1)) System.out.println("1 is non-negative");
    if(!isNonNeg.test(-1)) System.out.println("-1 is negative");
  }
}
```

Block Lambda Expressions

```
interface NumericFunc {
  int func(int n);
}

class BlockLambdaDemo {
  public static void main(String args[])
  {

    // This block lambda computes the factorial of an int value.
    NumericFunc  factorial = (n) ->  {
      int result = 1;

      for(int i=1; i <= n; i++)
        result = i * result;
      return result;
    };
    System.out.println("The factoral of 3 is " + factorial.func(3));
    System.out.println("The factoral of 5 is " + factorial.func(5));
  }
}
```