PES UNIVERSITY ONLINE

OBJECT ORIENTED PROGRAMMING WITH JAVA

Dr. N MEHALA
Department of Computer Science and Engineering

INHERITANCE BASICS

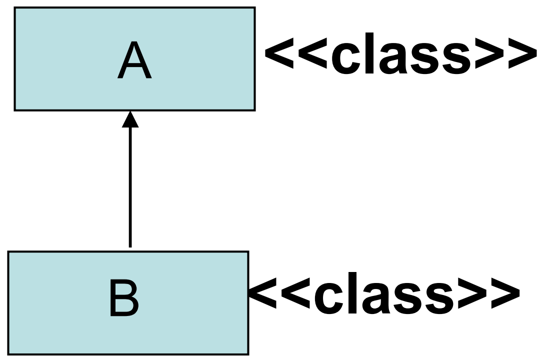
1. **Reusability is achieved by *INHERITANCE***
2. **Java classes Can be Reused by extending a class. Extending an existing class is nothing but reusing properties of the existing classes.**
3. *Inheritance* allows a software developer to derive a new class from an existing one
4. **The class whose properties are extended is known as *super or base or parent class.***
5. **The class which extends the properties of super class is known as *sub or derived or child class***
6. ***A class can either extends another class or can implement an interface***

Forms of Inheritance

class B extends A { ….. }

A super class

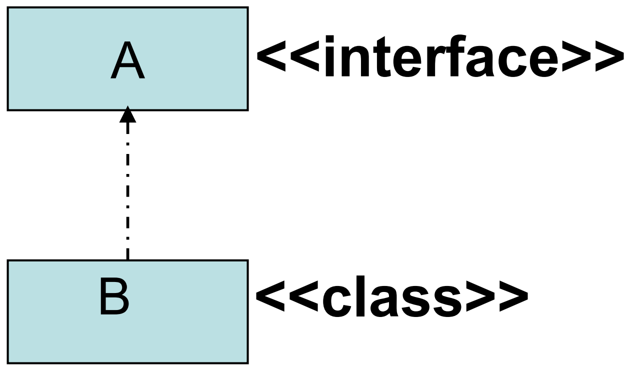
B sub class



class B implements A { ….. }

A interface

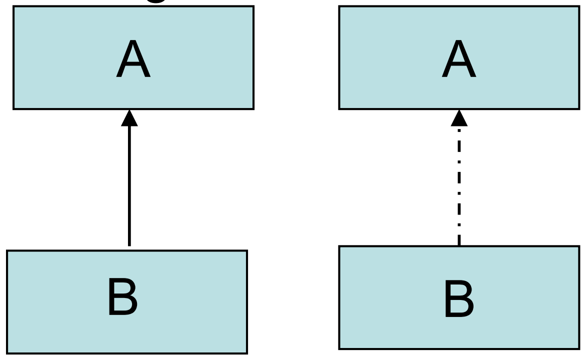
B sub class



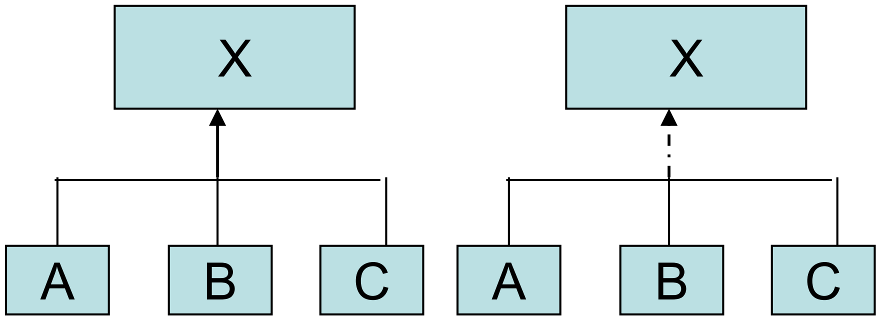
OBJECT ORIENTED PROGRAMMING WITH JAVA

Various Forms of Inheritance

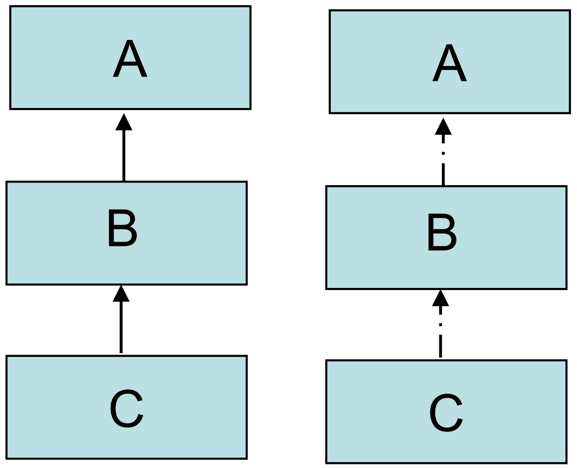
Single Inheritance



Hierarchical Inheritance



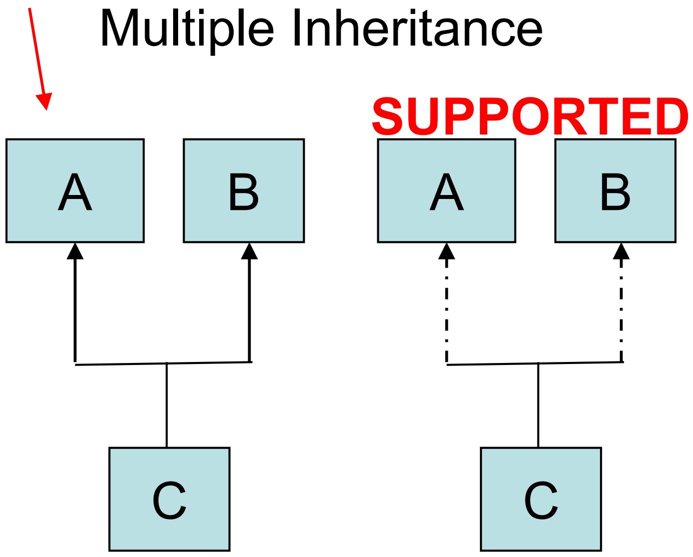
MultiLevel Inheritance



NOT SUPPORTED BY JAVA

Multiple Inheritance

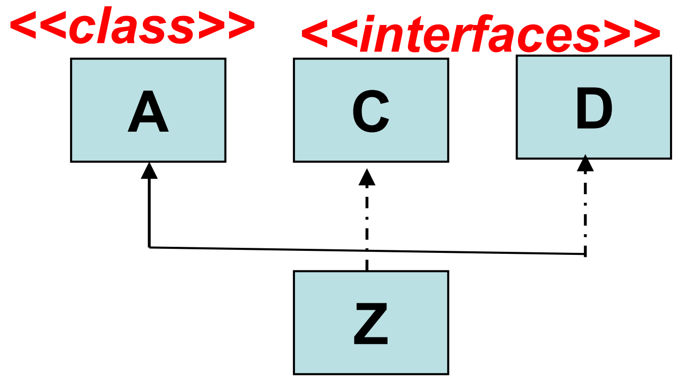
SUPPORTED BY JAVA



An interface can **extend** any number of interfaces but one interface **cannot implement** another interface, because if any interface is implemented then its methods must be defined and interface never has the definition of any method.

Forms of Inheritance

- Mulitiple Inheritance can be implemented by implementing multiple interfaces not by extending multiple classes



Example :

class Z extends A implements C , D

{ …………}

OK

class A extends B,C
{
}

WRONG

class A extends B extends C
{
}

WRONG

Defining a Subclass

Syntax :

```
class <subclass name> extends <superclass name>
{
 variable declarations;
 method declarations;
}
```

1. **Extends keyword signifies that properties of the super class are extended to sub class**
2. **Sub class will not inherit private members of super class**

Access Control

Access Modifiers → Access Location ↓	public	protected	<u>friendly</u>	<u>private protected</u>	<u>private</u>
Same Class	<u>Yes</u>	<u>Yes</u>	<u>Yes</u>	<u>Yes</u>	<u>Yes</u>
sub classes in same package	<u>Yes</u>	<u>Yes</u>	<u>Yes</u>	<u>Yes</u>	<u>No</u>
Other Classes in Same package	<u>Yes</u>	<u>Yes</u>	<u>Yes</u>	<u>No</u>	<u>No</u>
Subclasses in other packages	<u>Yes</u>	<u>Yes</u>	<u>No</u>	Yes	<u>No</u>
Non-subclasses in other packages	<u>Yes</u>	<u>No</u>	<u>No</u>	<u>No</u>	<u>No</u>

INHERITANCE BASICS

1. Whenever a sub class object is created ,super class constructor is called first.

2. If super class constructor does not have any constructor of its own OR has an unparametrized constructor then it is automatically called by Java Run Time by using call super()

3. If a super class has a parameterized constructor then it is the responsibility of the sub class constructor to call the super class constructor by call

   super(<parameters required by super class>)

4. Call to super class constructor must be the first statement in sub class constructor

INHERITANCE BASICS

When super class has a Unparametrized constructor

```
class A
{
A()
{
System.out.println("This is constructor of class A");
}
} // End of class A
class B extends A
{
B()
{
super();
System.out.println("This is constructor of class B");
}
} // End of class B
```

Optional

Cont…..

INHERITANCE BASICS

```
class inhtest
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

OUTPUT

```
This is constructor of class A
This is constructor of class B
```

INHERITANCE BASICS

```
class A
{
A()
{
System.out.println("This is class A");
}
}
class B extends A
{
B()
{System.out.println("This is class B");}
}
class inherit1
{
public static void main(String args[])
{
B b1 = new B();
}   }
```

File Name is xyz.java

```
/*
E:\Java>javac xyz.java

E:\Java>java xyz
Exception in thread "main"
java.lang.NoClassDefFoundError:
xyz

E:\Java>java inherit1
This is class A
This is class B

E:\Java>
*/
```

INHERITANCE BASICS

```
class A
{
private A()
{
System.out.println("This is class A");
}
}
class B extends A
{
B()
{
System.out.println("This is class B");
}
}
class inherit2
{
public static void main(String args[])
{
B b1 = new B();
}
}
```

Private Constructor in super class

```
/*
E:\Java>javac xyz1.java
xyz1.java:12: A() has private access
in A
{
^
1 error
*/
```

INHERITANCE BASICS

```
class A
{
private A()
{
System.out.println("This is class A");
}
A()
{
System.out.println("This is class A");
}
}
class B extends A
{
B()
{
System.out.println("This is class B");
}
}
class inherit2
{
public static void main(String args[])
{
B b1 = new B();
} }
```

```
/*
E:\Java>javac xyz2.java
xyz2.java:7: A() is already defined in
A
A()
^
xyz2.java:16: A() has private access
in A
{
^
2 errors
*/
```

When Super class has a parameterized constructor.

```
class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is constructor
of class A");
} }
class B extends A
{
private int b;
private double c;
B(int b,double c)
{
this.b=b;
this.c=c;
System.out.println("This is constructor
of class B");
} }
```

```
B b1 = new B(10,8.6);
```

```
D:\java\bin>javac inhtest.java
inhtest.java:15: cannot find
symbol
symbol  : constructor A()
location: class A
{
^
1 errors
```

INHERITANCE BASICS

```
class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is
constructor of class A");
} }
class B extends A
{
private int b;
private double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
System.out.println("This is
constructor of class B");
} }
```

```
B b1 = new B(8,10,8.6);
```

OUTPUT

```
This is constructor of class A
This is constructor of class B
```

INHERITANCE BASICS

```
class A
{
private int a;
protected String name;
A(int a, String n)
{
this.a = a;
this.name = n;
}
void print()
{
System.out.println("a="+a);
}
}
```

```
class B extends A
{
int b;
double c;
B(int a,String n,int b,double c)
{
super(a,n);
this.b=b;
this.c =c;
}
void show()
{
//System.out.println("a="+a);
print();
System.out.println("name="+name);
System.out.println("b="+b);
System.out.println("c="+c);
}
}
```

Can Not use *a*
a is private in super class

Calls print() from super class A

INHERITANCE BASICS

```
class xyz3
{
public static void main(String args[])
{
B b1 = new B(10,"OOP",8,10.56);
b1.show();
}
}
```

```
E:\Java>java xyz3
a=10
name=OOP
b=8
c=10.56
```

USE OF super KEYWORD

- Can be used to call super class constrctor

  super();

  super(<parameter-list>);

- Can refer to super class instance variables/Methods

  super.<super class instance variable/Method>

INHERITANCE BASICS

```
class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is constructor
of class A");
}
void print()
{
System.out.println("a="+a);
}
void display()
{
System.out.println("hello This is Display
in A");
}
}
```

```
class B extends A
{
private int b;
private double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
System.out.println("This is constructor
of class B");
}
void show()
{
print();
System.out.println("b="+b);
System.out.println("c="+c);
}
}
```

INHERITANCE BASICS

```
class inhtest1
{
public static void main(String args[])
{
B b1 = new B(10,8,4.5);
b1.show();
}
}
/* OutPUt
D:\java\bin>java inhtest1
This is constructor of class A
This is constructor of class B
a=10
b=8
c=4.5
*/
```

OBJECT ORIENTED PROGRAMMING WITH JAVA

INHERITANCE BASICS

```
class A
{
private int a;
A( int a)
{
this.a =a;
System.out.println("This is constructor
of class A");
}
void show()
{
System.out.println("a="+a);
}
void display()
{
System.out.println("hello This is Display
in A");
}
}
```

```
class B extends A
{
private int b;
private double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
System.out.println("This is constructor
of class B");
}
void show()
{
// show();
super.show();
System.out.println("b="+b);
System.out.println("c="+c);
display();
}
}
```

INHERITANCE BASICS

```
class inhtest1
{
public static void main(String args[])
{
B b1 = new B(10,8,4.5);
b1.show();
}
}
/* OutPut
D:\java\bin>java inhtest1
This is constructor of class A
This is constructor of class B
a=10
b=8
c=4.5
hello This is Display in A
*/
```

OBJECT ORIENTED PROGRAMMING WITH JAVA

INHERITANCE BASICS

```
class A
{
int a;
A( int a)
{ this.a =a; }
void show()
{
System.out.println("a="+a);
}
void display()
{
System.out.println("hello This is Display
in A");
}
}
```

```
class B extends A
{
int b;
double c;
B(int a,int b,double c)
{
super(a);
this.b=b;
this.c=c;
}
void show()
{
//super.show();
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("c="+c);
}
}
```

INHERITANCE BASICS

```
class inhtest2
{
public static void main(String args[])
{
B b1 = new B(10,20,8.4);
b1.show();
}
}
/*
D:\java\bin>java inhtest2
a=10
b=20
c=8.4
*/
```

INHERITANCE BASICS

```
class A
{
int a;
A( int a)
{ this.a =a; }
}
```

```
class B extends A
{
int a;   // super class variable a hides here
int b;
double c;
B(int a,int b,double c)
{
super(100);
this.a = a;
this.b=b;
this.c=c;
}
void show()
{
// How can we print the value of super class variable "a"?
System.out.println("Super class a="+super.a);
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("c="+c);
}   }
```

Use of super to refer to super class varible a

INHERITANCE BASICS

```
class inhtest2
{
public static void main(String args[])
{
B b1 = new B(10,20,8.4);
b1.show();
}
}

/* Out Put
D:\java\bin>java inhtest2
Super class a=100
a=10
b=20
c=8.4
*/
```

Dynamic Method Dispatch or Runtime Polymorphism

- Method overriding is one of the ways in which Java supports Runtime Polymorphism.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

OBJECT ORIENTED PROGRAMMING WITH JAVA

Example

```
class A
{
    void m1()
    {
        System.out.println("Inside A's
m1 method");
    }
}

class B extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside B's m1
method");
    }
}
```

```
class C extends A
{
    // overriding m1()
    void m1()
    {
        System.out.println("Inside C's m1
method");
    }
}
```

OBJECT ORIENTED PROGRAMMING WITH JAVA

Example

```
// Driver class
class Dispatch
{
    public static void main(String args[])
    {
          // object of type A
        A a = new A();

        // object of type B
        B b = new B();

        // object of type C
        C c = new C();

        // obtain a reference of type A
        A ref;

        // ref refers to an A object
        ref = a;

        // calling A's version of m1()
        ref.m1();

    // now ref refers to a B object
        ref = b;

        // calling B's version of m1()
        ref.m1();

        // now ref refers to a C object
        ref = c;

        // calling C's version of m1()
        ref.m1();
    }
}
```

OUTPUT:

```
Inside A's m1 method
Inside B's m1 method
Inside C's m1 method
```

EXAMPLE 2

```
// class A
class A
{
    int x = 10;
}

// class B
class B extends A
{
    int x = 20;
}

// Driver class
public class Test
{
    public static void main(String args[])
    {
        A a = new B(); // object of type B

        // Data member of class A will be accessed
        System.out.println(a.x);
    }
}
```

Output:

```
10
```

In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.**

Advantages of Dynamic Method Dispatch

- Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.
- It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

Exercise1

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```

The second class, a subclass of `Animal`, is called `Cat`:

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

Output :

```
The static method in Animal
The instance method in Cat
```

Exercise1

The distinction between hiding a static method and overriding an instance method has important implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Writing Final Classes and Methods

- You can declare some or all of a class's methods *final*. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.
- You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the getFirstPlayer method in this ChessAlgorithm class final:

```
class ChessAlgorithm
{……
 final ChessPlayer getFirstPlayer()
             {
             return ChessPlayer.WHITE;
              }
…
}
```

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the **String class.**

Exercise3

In Java, Constructor over-riding is possible?

In Java, Constructor overriding is not possible as the constructors are not inherited as overriding is always happens on child class or subclass but constructor name is same as a class name so constructor overriding is not possible but constructor overloading is possible.

Can we override static method in Java?
No, you cannot override a static method in Java because it's resolved at compile time. In order for overriding to work, a method should be virtual and resolved at runtime because objects are only available at runtime.

Can we overload a static method in Java?

Yes, you can overload a static method in Java. Overloading has nothing to do with runtime but the signature of each method must be different. In Java, to change the method signature, you must change either number of arguments, type of arguments or order of arguments.

Exercise 3: Static vs Dynamic binding

- Static binding is done during compile-time while dynamic binding is done during run-time.
- private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

THANK YOU

Dr. N MEHALA

Department of Computer Science and Engineering

mehala@pes.edu