# Inheritance and Abstract Classes

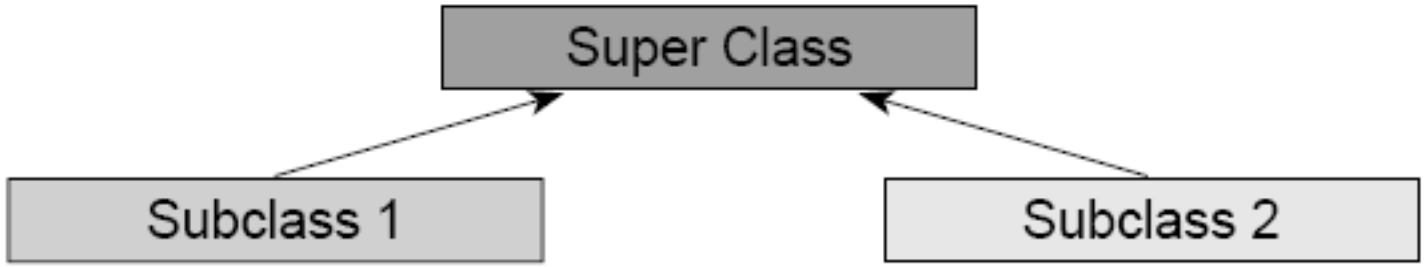
**MCA Sem II**
**Paper Code-CS2T05**
**(Object Oriented Programming)**

-- **Dr. Bhawna Sinha**
Department of MCA,
PWC
bhawna.sahay2004@gmail.com

# Inheritance

- Is the ability to derive something specific from something generic.
- *aids in the reuse of code.*
- A class can inherit the features of another class and add its own modification.
- The parent class is the *super class* and the child class is known as the s*ubclass*.
- A subclass inherits all the properties and methods of the super class.

# Inheritance

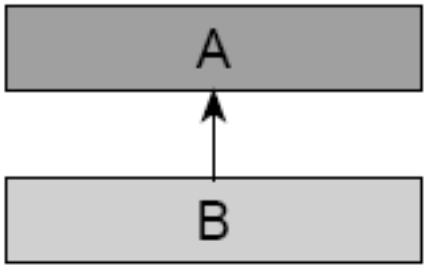


# Types of Inheritance

- Single
- Multiple
- Multilevel

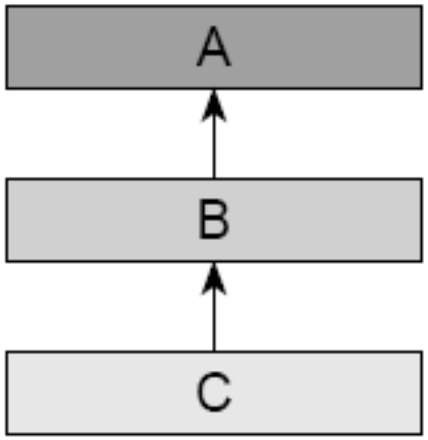
# Single level inheritance

Classes have only base class



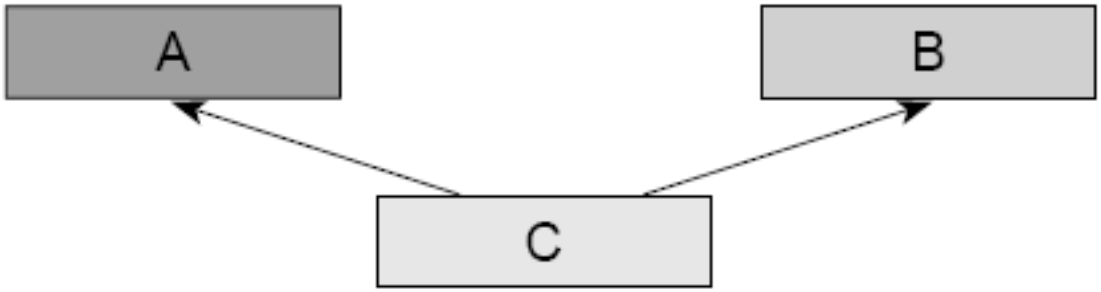
# Multi level

There is no limit to this chain of inheritance (as shown below) but getting down deeper to four or five levels makes code excessively complex.



# Multiple Inheritance

A class can inherit from more than one unrelated class



# Deriving Classes

- Classes are inherited from other class by declaring them as a part of its definition
- For e.g.
  - class MySubClass **extends** MySuperClass
- *extends* keyword declares that *MySubClass* inherits the parent class *MySuperClass*.

# Method Overriding

- 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.
- It is a feature that supports polymorphism.
- When an overridden method is called through the subclass object, it will always refer to the version of the method defined by the subclass.
- The superclass version of the method is hidden.

# Method Overriding Example

```
class A {
int i = 0;
void doOverride (int k) {
i = k;          } }
class B extends A {
void doOverride(int k) {
i = 2 * k;
System.out.println("The value of i is: "+i);      }
public static void main (String args[])
{     B b = new B();
      b.doOverride(12);
}}
```

# The output

The value of i is: 24

# Late binding Vs Early binding

- Binding is connecting a method call to a method body.
- When binding is performed before the program is executed, it is called ***early binding***.
- If binding is delayed till runtime it is late binding.
- Also known as *dynamic binding* or *runtime binding*.
- All the methods in Java use ***late binding (except static and final)***.

# Super Keyword

- For invoking the methods of the super class.
- For accessing the member variables of the super class.
- For invoking the constructors of the super class.

# Super Keyword (contd.)

```
class A {
void show()
{      System.out.println(“Super Class show method”);
} }
class B extends A
void show()
{      System.out.println(“Subclass show method”);          }
public static void main (String args[ ]) {
        A s1=new A();
        s1.show();
        B s2=new B ();
        s2.show();
}}
```

# *Problem and Solution*

- Two methods being called by two different objects (inherited), instead the job can be done by one object only, i.e. using super keyword.

# Case 1: Invoking Methods using super

```
class ANew {
void show()
{                System.out.println("Super Class show
                 method");       } }
class BNew extends ANew {
void show()
{
        super.show();
        System.out.println("Subclass show method");
}
public static void main (String args[ ]) {
        BNew s2=new BNew ();
        s2.show();       }}
```

# Case 2: Accessing variables using super

```
class Super_Variable {
int b=30;     }
class SuperClass extends Super_Variable {
int b=12;
void show() {
System.out.println("subclass class variable: "+ b);
System.out.println("superclass instance variable: "+ super.b);
}
public static void main (String args[]) {
SuperClass s=new SubClass();
s.show(); // call to show method of Sub Class B
}}
```

# The output

- subclass class variable: 12
- superclass instance variable: 30

# Case 3: constructors of the super class.

```
class Constructor_A {
Constructor_A()   {
System.out.println("Constructor A");   }}
class Constructor_B extends Constructor_A {
Constructor_B()   {
System.out.println("Constructor B");   }}
class Constructor_C extends Constructor_B {
Constructor_C()   {
System.out.println("Constructor C");   }
public static void main (String args[]) {
Constructor_C a=new Constructor_C();
}}
```

# The Output

Constructor A

Constructor B

Constructor C

# Case 3: (contd.)

```
class Constructor_A_Revised {
Constructor_A_Revised()
{
        System.out.println("Constructor A Revised");
}}
class Constructor_B_Revised extends Constructor_A_Revised {
/*Constructor_B_Revised()
{
        System.out.println("Constructor B");
}*/
Constructor_B_Revised(int a)                    {
        a++;
System.out.println("Constructor B Revised "+ a );          }}
class Constructor_C_Revised extends Constructor_B_Revised {
Constructor_C_Revised()    {
        super(11); // if omitted compile time error results
        System.out.println("Constructor C Revised");                    }
public static void main (String args[]) {
Constructor_C_Revised a=new Constructor_C_Revised();
}}
```

# The Output

Constructor A Revised

Constructor B Revised 12

Constructor C Revised

# final keyword

- Declaring constants (used with variable and argument declaration)
  - final int MAX=100;
- Disallowing method overriding (used with method declaration)
  - final void show (final int x)
- Disallowing inheritance (used with class declaration).
  - final class Demo {}

# Abstract class

- Abstract classes are classes with a generic concept, not related to a specific class.
- Abstract classes define partial behaviour and leave the rest for the subclasses to provide.
- contain one or more abstract methods.
- abstract method contains no implementation, i.e. no body.
- Abstract classes cannot be instantiated, but they can have reference variable.
- If the subclasses does not override the abstract methods of the abstract class, then it is mandatory for the subclasses to tag itself as *abstract.*

# Why create abstract methods?

- to force *same name and signature pattern* in all the subclasses
- subclasses should not use their own naming patterns
- They should have the flexibility to code these methods with their own specific requirements.

# Example of Abstract class

```
abstract class Animal
{
        String name;
        String species;
        Animal(String n, String s)
        {
                name=n;
                species=s;
        }
        void eat(String fooditem)
        {
        System.out.println(species +" "+ name + "likes to have "+
        fooditem);
        }
        abstract void sound();
}
```

# Summary

- The concept of Inheritance is derived from real life.
- The properties and methods of a parent class are inherited by the children or subclasses.
- Subclasses can provide new implementation of method using method overriding, keeping the method names and signatures same as that of the parent class.
- The super keyword can be used to access the overridden methods, variables and even the constructors of the super class.
- Abstract classes are used to force the subclasses to override abstract methods and provide body and code for them.
- The final keyword is used to create constants and disallow inheritance.