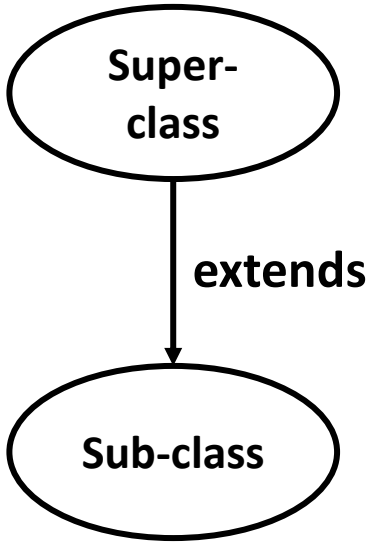
Inheritance

Overriding & super keyword

Inheritance

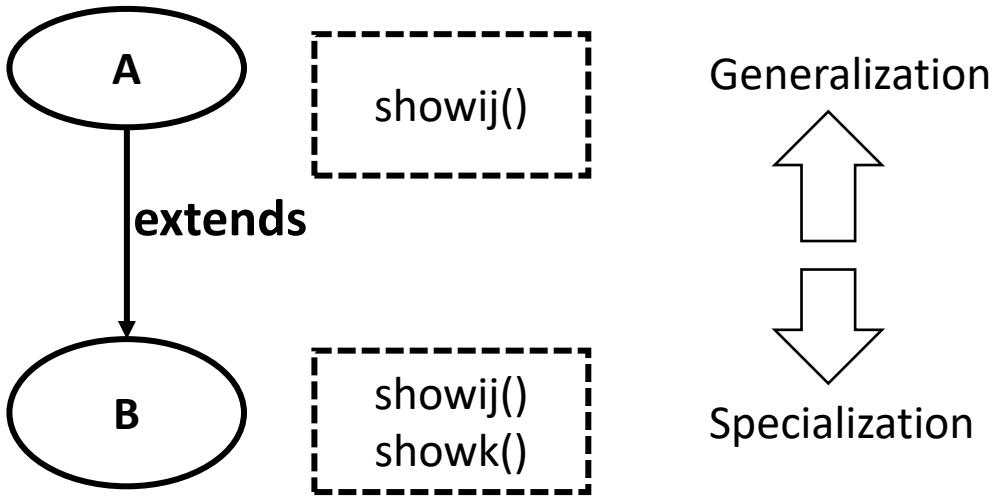
- Inheritance is an important feature of OOP.
- Inheritance is a mechanism in which one class acquires all the properties and behaviors of a parent class.
- **extends** keyword used for inheritance.
- A class that is inherited is called a **superclass**.
- The class that inherits the other class is called a **subclass**.
- Therefore, a subclass is a specialized version of a superclass.

```
class subclass-name extends superclass-name
{
// body of class
}
```



Inheritance

```
class A {
        int i=5, j=10;
        void showij()
        {
        System.out.println("i and j: " + i + " " + j);
        }
}
```




```
class B extends A {
        int k=100;
        void showk()
        {
                System.out.println("k: " + k);
        }
        public static void main(String args []) {
        A superOb = new A();
        B subOb = new B();
        System.out.println("Method with super_Object: ");
        superOb.showij();
        System.out.println("Method with sub_Object: ");
        subOb.showij();
        subOb.showk();
        }
}
```

Inheritance

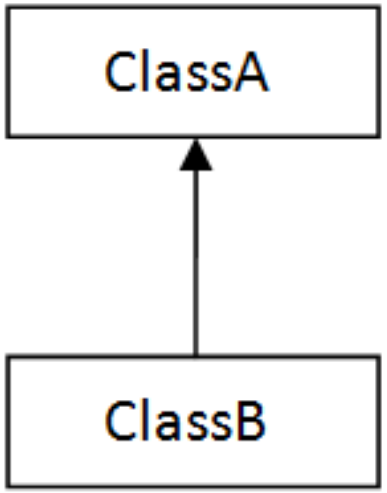
- We can only specify one superclass for any subclass.
- Java does not support the inheritance of multiple super-classes into a single subclass.
- But it is possible that we can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass.
- However, no class can be a superclass of itself.
- The inherited fields can be used directly, just like any other fields.
- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.
- A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

Types of Inheritance in Java

- Single Inheritance:
- Multilevel Inheritance:
- Multiple Inheritance
- Hybrid Inheritance

Types of Inheritance in Java

- Single Inheritance: In single inheritance, subclasses inherit the features of one superclass.
- class A serves as a base class for the derived class B.



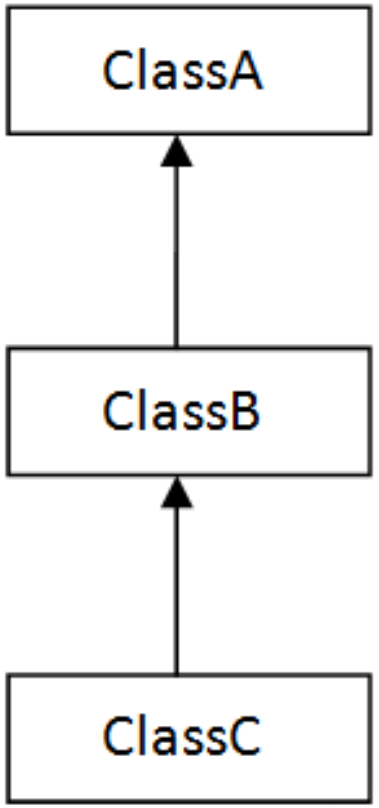
Example:

```
class A
    {
    // body of the class
    }


class B extends A
    {
    // body of the class
    }
```

Types of Inheritance in Java

- Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.
- class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



Example:

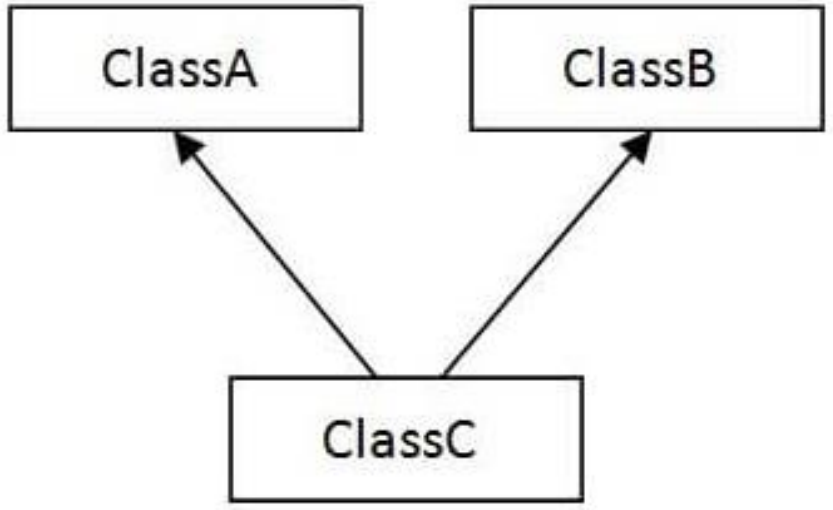
```
class A
    {
    // body of the class
    }

class B extends A
    {
    // body of the class
    }

class C extends B
    {
    // body of the class
    }
```

Types of Inheritance in Java

- Multiple Inheritance (Through Interfaces): In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes.
- Java does not support multiple inheritances with classes. In java, we can achieve multiple inheritances only through Interfaces.
- class A and B serves as a base class for the derived class C.



Example:

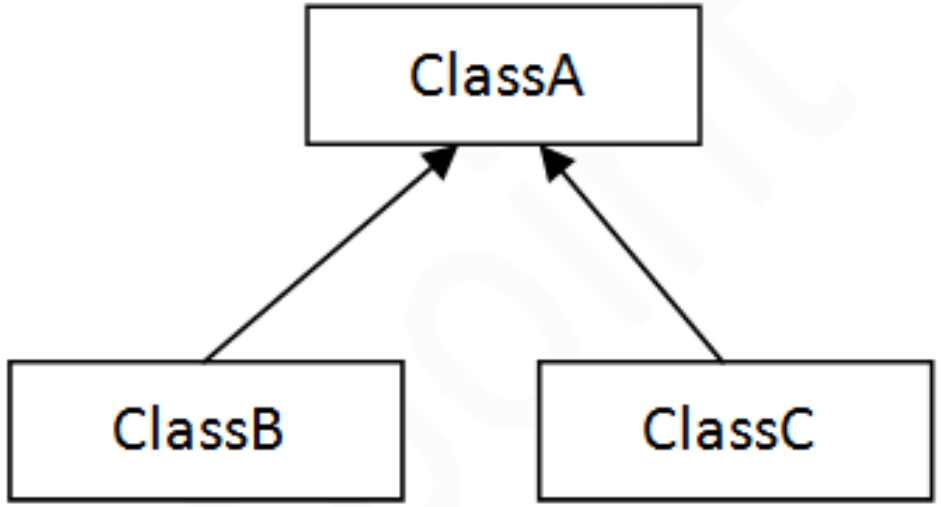
```
class A
    {
    // body of the class
    }
Interface B
    {
    // body of the Interface
    }
class C extends A implements B
    {
    // body of the class
    }
```

Example:

```
Interface A
    {
    // body of the Interface
    }
Interface B
    {
    // body of the Interface
    }
class C implements A, B
    {
    // body of the class
    }
```

Types of Inheritance in Java

- Hierarchical Inheritance: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.
- class A serves as a base class for the derived class B and C.

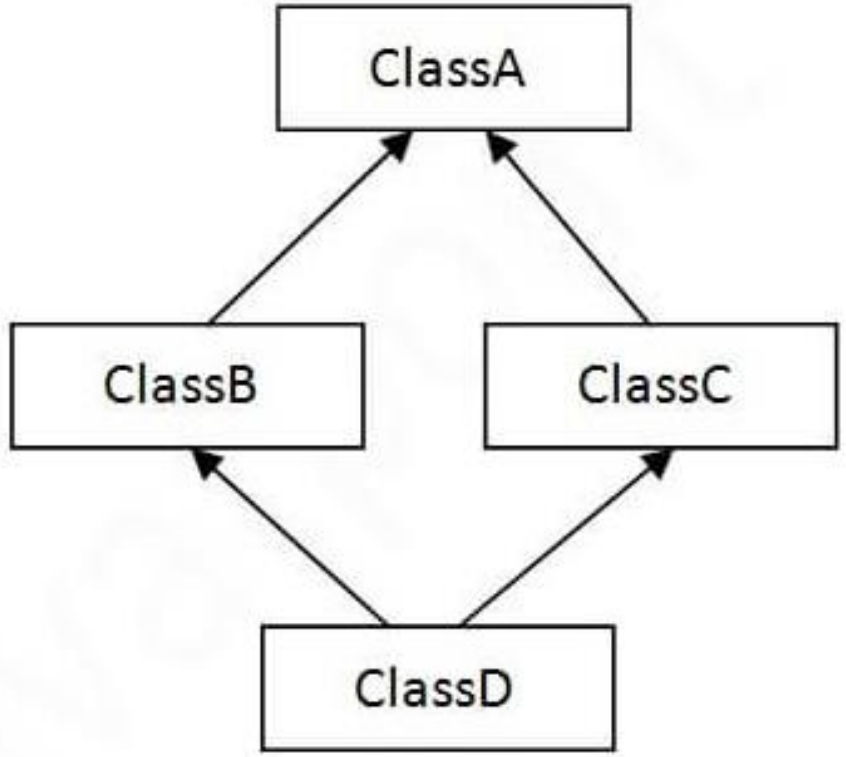


Example:

```
class A
    {
    // body of the class
    }
class B extends A
    {
    // body of the class
    }
class C extends A
    {
    // body of the class
    }
```

Types of Inheritance in Java

- Hybrid Inheritance(Through Interfaces): It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritances with classes, hybrid inheritance is also not possible with classes.
- class A serves as a base class for the derived class B and C.
- And again class B and C serves as a base class for the derived class C.



- All are interface
- A as interface and B/C interface, D as a class
- A as interface and B/C class (B/C interface), D as a class

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
class B extends A {
        int k=100;
        void showk()
        {
                System.out.println("k: " + k);
        }
        public static void main(String args []) {
        A superOb = new A();
        A superRef = new B();
        System.out.println("Method with super_Object: ");
        superOb.showij();
        System.out.println("Method with super_Reference : ");
         superRef.showij();
         superRef.showk();
        }
}
```

```
class A {
        int i=5, j=10;
        void showij()
        {
        System.out.println("i and j: " + i + " " + j);
        }
        }
```

A Superclass Variable Can Reference a Subclass Object

- References denote objects and are used to manipulate objects.
- When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.
- This is why superRef can’t access showk() even when it refers to a sub-class object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it.

Using super

- **super** keyword has two general forms.
  i. To access a member of the superclass
  ii. To call the superclass constructor
- When a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

  super.*member*
- Here, *member* can be either a *method* or an *instance variable*.
- **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Using super

- Example:

```
class A {
        int i=5;
}

class B extends A {
        int i=100;
        void show()
        {
                System.out.println("i in superclass: " + super.i);
                System.out.println("i in subclass: " + i);
        }
        public static void main(String args []) {
                B superRef = new B();
                 superRef.show();
        }
}
```

Using super

- Example:

```
class A {
        int i=5;
}

class B extends A {
        int i=100;
        void show()
        {
                System.out.println("i in superclass: " + super.i);
                System.out.println("i in subclass: " + i);
        }
        public static void main(String args []) {
                B superRef = new B();
                 superRef.show();
        }
}
```

Output:

```
i in superclass: 5
i in subclass: 100
```

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.

Method Overriding

```
class A {
        void show ()
        {
        System.out.print("show() in super class");
        }
}
```

```
class B extends A {
        void show()
        {
        System.out.print("show() in sub class");
        }
        public static void main(String args []) {
                B subOb = new B();
                subOb.show();
        }
}
```

Method Overriding

```
class A {
        void show ()
        {
        System.out.print("show() in super class");
        }
}
```

```
class B extends A {
        void show()
        {
        System.out.print("show() in sub class");
        }
        public static void main(String args []) {
                B subOb = new B();
                subOb.show();
        }
}
```

Output:

show() in sub class

Method Overriding

- In the example, the version of show( ) inside B overrides the version declared in A.
- If you wish to access the superclass version of an overridden method, you can do so by using **super**.

Method Overriding

```
class A {
        void show ()
        {
        System.out.print("show() in super class");
        }
}
```

```
class B extends A {
        void show()
        {
        super.show();
        System.out.print("show() in sub class");
        }
        public static void main(String args []) {
                B subOb = new B();
                subOb.show();
        }
}
```

Method Overriding

```
class A {
        void show ()
        {
        System.out.print("show() in super class");
        }
}
```

```
class B extends A {
        void show()
        {
        super.show();
        System.out.print("show() in sub class");
        }
        public static void main(String args []) {
                B subOb = new B();
                subOb.show();
        }
}
```

Output:

show() in super class
show() in sub class

Method Overriding

- A Superclass Variable Can Reference a Subclass Object:
- Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the *type of the object* being referred to at the time the call occurs.
- It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.
- If a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Method Overriding

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

class B extends A{
        void show ()
        {
        System.out.println("show() in B class");
        }

}
```

```
class C extends B {
        void show()
        {
        System.out.println("show() in C class");
        }
        public static void main(String args []) {
                A a = new A();
                B b = new B();
                C c = new C();
                A r;
                r = a;
                r.show();
                r = b;
                r.show();
                r = c;
                r.show();
        }}
```

Method Overriding

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

class B extends A{
        void show ()
        {
        System.out.println("show() in B class");
        }

}
```

Output:

show() in A class
show() in B class
show() in C class

```
class C extends B {
        void show()
        {
        System.out.println("show() in C class");
        }
        public static void main(String args []) {
                A a = new A();
                B b = new B();
                C c = new C();
                A r;
                r = a;
                r.show();
                r = b;
                r.show();
                r = c;
                r.show();
}}
```

Method Overriding

- Important points:
- The new method definition must have the same method signature, i.e., the method name, and the types and the number of parameters, including their order, are the same as in the overridden method.
- Whether parameters in the overriding method should be *final* is at the discretion of the subclass. A method's signature does not comprise the final modifier of parameters, only their types and order.
- The return type of the overriding method can be a *subtype* of the return type of the overridden method (called covariant return).
- The new method definition cannot narrow the accessibility of the method, but it can widen it.
- The new method definition can only throw all or none, or a subset of the checked exceptions (including their subclasses) that are specified in the throws clause of the overridden method in the superclass

Overriding vs. Overloading

Comparison Criteria	Overriding	Overloading
Method name	Must be the same.	Must be the same.
Argument list	Must be the same.	Must be different.
Return type	Can be the same type or a covariant type.	Can be different.
`throws` clause	Must not throw new checked exceptions. Can narrow exceptions thrown.	Can be different.
Accessibility	Can make it less restrictive, but not more restrictive.	Can be different.
Declaration context	A method can only be overridden in a subclass.	A method can be overloaded in the same class or in a subclass.
Method call resolution	The *runtime type* of the reference, i.e., the type of the object referenced at *runtime*, determines which method is selected for execution.	At compile time, the *declared type* of the reference is used to determine which method will be executed at runtime.