# Abstract keyword and Abstraction

**Example**

```
class DB_Driver{
       public void getDriver(){
               System.out.println("Type-1 Driver");
       }
}
class New_DB_Driver extends DB_Driver{
      public void getDriver(){
             System.out.println("Type-4-Driver");
      }
}
class Test{
      public static void main(String args[]){
                    DB_Driver driver=new New_DB_Driver();
                    driver.getDriver();
     }
}
```

- In the above example, method overriding is implemented, in method overriding, for super class method call JVM has to execute subclass method, not super class method.
- In method overriding, always JVM is executing only subclass method, not super class method.
- In method overriding, it is not suggestible to manage super class method body without execution, so that, we have to remove superclass method body as part of code optimization.
- In Java applications, if we want to declare a method without body then we must declare that method as "Abstract Method".
- If we want to declare abstract methods then the respective class must be an abstract class.

**Example**

```
abstract class DB_Driver{
        public abstract void getDriver();
}
class New_DB_Driver extends DB_Driver{
      public void getDriver(){
              System.out.println("Type-4 Driver");
      }
}

public class Test{
       public static void main(String args[]){
                     DB_Driver driver=new New_DB_Driver();
                     driver.getDriver();
       }
}
```

- In Java applications, if we declare any abstract class with abstract methods, then it is convention to implement all the abstract methods by taking sub classes.
- To access the abstract class members, we have to create an object for the subclass and we have to create a reference variable either for abstract class or for the subclass.
- If we create reference variables for abstract class then we are able to access only abstract class members, we are unable to access subclass own members.
- If we declare a reference variable for subclass then we are able to access both abstract class members and subclass members.

**Example**

```
abstract class A{
        void m1(){
                System.out.println("m1-A");
        }

        abstract void m2();
        abstract void m3();
}
class B extends A{
      void m2(){
              System.out.println("m2-B");
      }
      void m3(){
              System.out.println("m3-B");
      }
      void m4(){
              System.out.println("m4-B");
      }
}

public class Test{
        public static void main(String args[]){
               A a=new B();
               a.m1();
               a.m2();
               a.m3();
               //a.m4();---error

               B b=new B();
               b.m1();
               b.m2();
               b.m3();
               b.m4();
       }
}
```

In Java applications,it is not possible to create Objects for abstract classes but it is possible to provide constructors in abstract classes,because,to recognize abstract class instance variables in order to store them in the subclass objects.

**EX:**

```
abstract class A{
  A(){
      System.out.println("A-Con");
  }
}
class B extends A{
      B(){
             System.out.println("B-Con");
      }
}
public class Test{
       public static void main(String[] args){
              B b=new B();
       }
}
```

In Java applications, if we declare any abstract class with abstract methods then it is mandatory to implement all the abstract methods in the respective subclass.
If we implement only some of the abstract methods in the respective subclass then compiler will rise an error, where to come out from compilation error we have to declare the respective subclass as an abstract class and we have to provide implementation for the remaining abstract methods by taking another subclass in multilevel inheritance.

**EX:**

```
abstract class A{
      abstract void m1();
      abstract void m2();
      abstract void m3();
}
abstract class B extends A{
       void m1(){
             System.out.println("m1-A");
       }
}
class C extends B{
      void m2(){
              System.out.println("m2-C");
     }
     void m3(){
              System.out.println("m3-C");
     }
}
public class Test{
       public static void main(String[] args){
              A a=new C();
              a.m1();
              a.m2();
              a.m3();
       }
}
```

In Java applications, if we want to declare an abstract class then it is not at all mandatory to have at least one abstract method, it is possible to declare abstract class without having abstract methods but if we want to declare a method as an abstract method then the respective class must be an abstract class.

```
abstract class A{
      void m1(){
              System.out.println("m1-A");
      }
}
class B extends A{
        void m2(){
                System.out.println("m2-B");
         }
}
public class Test{
       public static void main(String args[]){
              A a=new B();
              a.m1();
              //a.m2();---------->Error

               B b=new B();
               b.m1();
               b.m2();
      }
}
```

In Java applications, it is possible to extend an abstract class to concrete class and from concrete class to abstract class.

```
class A{
      void m1(){
             System.out.println("m1-A");
      }
}
abstract class B extends A{
        abstract void m2();
}
class C extends B{
        void m2(){
                System.out.println("m2-C");
        }
        void m3(){
                System.out.println("m3-C");
        }
}
public class Test{
       public static void main(String args[]){
              A a=new C();
              a.m1();
              //a.m2();  error
             //a.m3();  error
```

```
            B b=new C();
            b.m1();
            b.m2();
            //b.m3(); error

            C c=new C();
            c.m1();
            c.m2();
            c.m3();
    }
}
```

**Note:**
In Java applications, it is not possible to extend a class to the same class, if we do the same then the compiler will raise an error like "cyclic inheritance involving".

```
class A extends A{
}
Status: Compilation Error: “cyclic inheritance involving".

class A extends B{
}
class B extends A{
}
```

**Status:** Compilation Error: “cyclic inheritance involving".

**final class**

- If a class is marked as final, then the class won’t participate in inheritance,if we try to do so then it would result in “CompileTime Error”.

Eg: String, StringBuffer,Integer,Float,.....

**final variable**

- If a variable is marked as final, then those variables are treated as compile time constants and we should not change the value of those variables.
- If we try to change the value of those variables then it would result in “CompileTimeError”.

**final method**

- If a method is declared as final then those methods we can’t override, if we try to do so it would result in “CompileTimeError”.

**Note**

**Every method present inside a final class is always final by default whether we are declaring or not. But every variable present inside a final class need not be final.**

The main advantage of the final keyword is we can achieve security.
Whereas the main disadvantage is we are missing the key benefits of oops:
polymorsim (because of final methods), inheritance (because of final classes) hence if there is no specific requirement never recommended to use final keyboard.