Fragile Base Class Problem

```
class A{
    public void print( ){
        //System.out.print("Hello,");
        System.out.print("Good Morning,");
    }
}
class B extends A{
    @Override
    public void print( ){
        super.print();
        System.out.println("Have a nice day!!");
    }
}
class C extends A{
    @Override
    public void print( ){
        super.print();
        System.out.println("Good day!!");
    }
}
class Program{
    public static void main(String[] args) {
        A a = null;
        a = new B();    a.print();  //Good Morning,,Have a nice day!!
        a = new C();    a.print();  //Good Morning,,Good day!!
    }
}
```

- If we make changes in the super class then it is necessary to recompile super class as well as all its sub classes. This problem is called as fragile base class problem.
- We can avoid it by defining super type interface.

Interface

- It is a Java languages features
  - Which builds trust between service provider and service consumer
  - And which minimizes service provider dependancy.
- In General, interface is contract between service provider and service consumer.
- Non Primitive / reference types in Java:
  - Interface
  - Class
  - Enum
  - Array
- Interface is non primitive / reference type in java
- interface is keyword in Java.

```
interface Printable{
    //TODO
}
```

- Inside interface, we can declare following members:
    - Netsted interface
    - Field
    - Abstract method
    - Default method
    - Static method
- Example:
    - java.lang.AutoCloseable
    - java.io.Closeable
    - java.lang.Cloneable
    - java.lang.Iterable
    - java.util.Iterator
    - java.lang.Comparable
    - java.util.Comparator
    - java.lang.Runnable
    - java.util.Collection
    - java.util.List
    - java.util.Set
    - java.util.Queue
    - java.util.Map
- Consider following code:

```
interface Printable{
    int number = 10;
    //public static final int number = 10;

    void print();
    //public abstract void print();
}
```

- Interface fields are by default public static & final and interface methods are by default public & abstract.
- Consider the following code:

```
package org.example;
interface Printable{
    int number = 10;
    //public static final int number = 10;

    void print();
    //public abstract void print();
```

```
}
abstract class Test implements Printable {

}
```

- Consider the following code:

```
interface Printable{
    int number = 10;
    //public static final int number = 10;

    void print();
    //public abstract void print();
}
class Test implements Printable {
    @Override
    public void print() {
        System.out.println("Number  :   "+Printable.number);
    }
}
public class Program {
    public static void main(String[] args) {
        Printable p = new Test();   //Polymorphic declaration
        p.print();  //Dynamic method dispatch
    }
}
```

Types of inheritance in OOPS

- Interface inheritance
  - Single inheritance [Allowed in Java]
  - Multiple inheritance [Allowed in Java]
  - Hierarchical inheritance [Allowed in Java]
  - Multilevel inheritance [Allowed in Java]
- Implementation inheritance
  - Single inheritance [Allowed in Java]
  - Multiple inheritance [Not Allowed in Java]
  - Hierarchical inheritance [Allowed in Java]
  - Multilevel inheritance [Allowed in Java]

Inheritance syntax

```
//Interfaces: I1, I2, I3
//Classes:    C1, C2, C3
I2 extends C1       //Not OK
I2 implements C1    //Not OK
I2 implements I1    //NOT OK
```

```
I2 extends I1       //OK: Interface inheritance
I3 extends I1, I2   //OK: Multiple  Interface inheritance
C1 extends I1       //NOT OK
C1 implements I1    //OK: Interface Implementation Inheritance
C1 implements I1, I2, I3    //OK: Multiple Interface Implementation Inheritance
C2 implements C1;   //NOT OK
C2 extends C1;      //OK: Implementation inheritance
C3 extends C1,C2;      //NOT OK: Multiple Implementation inheritance
C2 implements I1 extends C1;    //NOT OK
C2 extends C1 implements I1;    //OK
C2 extends C1 implements I1, I2, I3;    //OK
```

- In case of inheritance, if super type and sub type are interfaces then it is called interface inheritance.
- In Java, interface can extend more than one interfaces. It is called as multiple interface inheritance.
- We can not extend interface from class. In other words, super type of interface must be interface.
- In case of inheritance, if super type is interface and sub type is class then it is called interface implementation inheritance.
- Class can implement more than one interfaces. It is also called as multiple interface implementation inheritance.
- In case of inheritance if super type and sub type is class then it is called implementation inheritance.
- In Java, class can extend only one class. In other words Java do not support multiple implementation inheritance.

Consider following code:

```
interface A{
    int num1 = 10;
    int num4 = 40;
    int num5 = 70;
}
interface B{
    int num2 = 20;
    int num4 = 50;
    int num5 = 80;
}
interface C extends A, B{
    int num3 = 30;
    int num4 = 60;
}
public class Program {
    public static void main(String[] args) {
        System.out.println("Num5    :   "+A.num5);  //70
        System.out.println("Num5    :   "+B.num5);  //80
        //System.out.println("Num5  :   "+C.num5);  //The field C.num5 is
ambiguous
    }
    public static void main2(String[] args) {
        System.out.println("Num4    :   "+A.num4);  //40
        System.out.println("Num4    :   "+B.num4);  //50
        System.out.println("Num4    :   "+C.num4);  //60
```

```
    }
    public static void main1(String[] args) {
        System.out.println("Num1    :   "+A.num1);  //10
        System.out.println("Num1    :   "+C.num1);  //10

        System.out.println("Num2    :   "+B.num2);  //20
        System.out.println("Num2    :   "+C.num2);  //20

        System.out.println("Num3    :   "+C.num3);  //30
    }
}
```

Consider following code

```
interface A{
    void f1();
    void f3();
}
interface B{
    void f2();
    void f3();
}
class C implements A, B{
    @Override
    public void f1() {
        System.out.println("C.f1()");
    }
    @Override
    public void f2() {
        System.out.println("C.f2()");
    }
    @Override
    public void f3() {
        System.out.println("C.f3");
    }
}
public class Program {
    public static void main(String[] args) {
        A a = new C();  //Polymorphic declaration / upcasting
        a.f1(); //C.f1()
        a.f3(); //C.f3

        B b = new C();  //Polymorphic declaration / upcasting
        b.f2(); //C.f2()
        b.f3(); //C.f3
    }
}
```

abstract method

*If implementation of a method in super-class is not possible/incomplete, then method is declared as abstract. *Abstract method does not have definition/implementation. *According to client's requirement, if implementation of super class method is logically 100% incomplete then we should declare super class method abstract. *abstract is a keyword in java. *We can not provide body to the abstract method. *If we declare method abstract then it is mandatory to declare class abstract.

- The super-class abstract methods must be overridden in sub-class; otherwise sub-class should also be marked abstract.
- The abstract methods are forced to be implemented in sub-class. It ensures that sub-class will have corresponding functionality.
- Either we should override abstract method in sub class or we should declare sub class abstract.
- The abstract method cannot be private, final, or static.
- Example: abstract methods declared in Number class are:
    - abstract int intValue();
    - abstract float floatValue();
    - abstract double doubleValue();
    - abstract long longValue();

```
abstract class A{
   public abstract void f3( );  //Logically 100% incomplete
}
public class Program {
   public static void main(String[] args) {
    A a = null;     //OK
    a = new A( );   //Cannot instantiate the type A
   }
}
```

- We cannot instantiate abstract class but we can create reference of abstract class.
- Solution 1

```
abstract class A{
   public final void f1() { //Logically 100% complete
    System.out.println("A.f1");
   }
   public void f2( ) {//Logically partial complete
    System.out.println("A.f2");
   }
   public abstract void f3( );  //Logically 100% incomplete

}
abstract class B extends A{

}
public class Program {
   public static void main(String[] args) {
    A a = null;     //OK
```

```
    }
}
```

```
- Abstract class example:
- java.lang.Enum
- java.lang.Number
- java.util.Calendar
```

abstract class

- If implementation of a class is logically incomplete, then the class should be declared abstract.
- If class contains one or more abstract methods, then class must be declared as abstract.
- An abstract class can have zero or more abstract methods.
- Abstract class object cannot be created; however its reference can be created.
- Abstract class can have fields, methods, and constructor.
- Its constructor is called when sub-class object is created and initializes its (abstract class) fields.
- If object of a class is not logical (corresponds to real-world entity), then class can be declared as abstract.

```
abstract class A{
   public abstract void f3( );  //Logically 100% incomplete
}
class B extends A{
   public void f3( ) {
    System.out.println("B.f3");
   }
}
public class Program {
   public static void main(String[] args) {
    A a = null;     //OK
    a = new B( );   //OK: Upcasting
        a.f3( );        //OK: B.f3
   }
}
```

Consider following code:

```
interface Printable{
    void f1();
    void f2();
    void f3();
}
abstract class AbtractPrintable implements Printable{
    @Override
    public void f1() {  }
```

```
    @Override
    public void f2() {  }
    @Override
    public void f3() {  }
}
class B extends AbtractPrintable{
    @Override
    public void f1() {
        System.out.println("B.f1");
    }
}
class C extends AbtractPrintable{
    @Override
    public void f2() {
        System.out.println("C.f2");
    }
}
class D extends AbtractPrintable{
    @Override
    public void f3() {
        System.out.println("D.f3");
    }
}
public class Program {
    public static void main(String[] args) {
        Printable p = null;
        p = new B();    p.f1();
        p = new C();    p.f2();
        p = new D();    p.f3();
    }
}
```

What is the difference between abstract class and interface? / When we should use abstract class and interface?

Abstract class

```
abstract class Shape{
    public abstract void calculateArea( );
}
class Rectangle extends Shape{
    @Override
    public void calculateArea(){
        //TODO
    }
}
class Circle extends Shape{
    @Override
    public void calculateArea(){
```

```
        //TODO
    }
}
class Triangle extends Shape{
    @Override
    public void calculateArea(){
        //TODO
    }
}
```

- If "is-a" relationship is exist between super type & sub type and if we want to maintain same method signature/design in all the sub clases then we should declare super type abstract.

```
Shape[] arr = new Shape[ 3 ];
arr[ 0 ] = new Rectangle();
arr[ 1 ] = new Circle();
arr[ 2 ]  = new Triangle();
```

- Using abstract class, we can group instances of related type together.
- Abstract class can extend only one abstract class / concrete class. In other words, using abstract class, we can not achieve multiple inheritance.
- We can define constructor inside abstract class.
- Abstract class may / may not contain abstract method.
- In General, if state is involved in super type then super type should be abstract class.

Interface

```
interface Printable{
    void printRecord( );
}
class Complex implements Printable{
    @Override
    public void printRecord(){
        System.out.println("Print complex number");
    }
}
class Point implements Printable{
    @Override
    public void printRecord(){
        System.out.println("Print point");
    }
}
class Date implements Printable{
    @Override
    public void printRecord(){
        System.out.println("Print date");
```

```
    }
}
```

- If "is-a" relationship is not exist(can-do relationship is exist) between super type & sub type and if we want to maintain same method design in all the sub classes then we should declare super type interface.

```
Printable[] arr = new Printable[ 3 ];
arr[ 0 ] = new Complex();
arr[ 1 ] = new Point();
arr[ 2 ] = new Date();
```

- Using interface, we can group instances of unrelated type together.
- Interface can extend more than one interfaces. In other words, using interface, we can achive multiple inheritance.
- We can not define constructor inside interface.
- Interface methods are by default abstract.
- In General, if state is not involved in super type then super type should be interface.

##equals

- We can not compare state of variable of primitive type using equals method.

```
public static void main(String[] args) {
    int num1 = 10;
    int num2 = 10;
    if( num1.equals( num2 ) ) //Cannot invoke equals(int) on the primitive type
int
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
}
```

- Realational operators: <, >, <=, >=, ==, !=
- relational operators generate boolean value(true/ false).
- If we want to compare state of variable of primitive/value type then we should use == operator which is relational operator.

```
public static void main(String[] args) {
    int num1 = 10;
    int num2 = 10;
    if( num1 == num2 ) //OK
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
```

```
    //Output: Equal
}
```

- We can use operator == with the variables of non primitive / reference type.

```
public static void main(String[] args) {
    Employee emp1 = new Employee("Sandeep", 378, 65000.50f);
    Employee emp2 = new Employee("Sandeep", 378, 65000.50f);
    if( emp1 == emp2 )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
        //Output: Not Equal
}
```

- If we use operator == with the variables of non primitive type then it doesn't compare state of instances. Rather it compares state of references.
- If we want to compare state of instance of reference / non primitive type then we should use equals method.
- equals is non final method of java.lang.Object class.
- Syntax:
    - public boolean equals(Object obj);
- If we do not define equals method inside class then super class's equals method will call. If any super class do not contain equals method then object class's equals method will call.
- Consider implementation of object class's equals method.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

```
public static void main(String[] args) {
    Employee emp1 = new Employee("Sandeep", 378, 65000.50f);
    Employee emp2 = new Employee("Sandeep", 378, 65000.50f);
    if( emp1.equals(emp2) )
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
        //Output: Not Equal
}
```

- As shown in above code, Object class's equals method do not compare state of instances. Rather it compares state of references.
- If we want to compare state of instances then we should override equals method inside class.
- Converting reference of sub class into reference of super class is called upcasting.

```
Employee emp = new Employee( "Sandeep", 378, 65000.50f );
Object o1 = ( Object )emp;  //OK: Upcasting
Object o2 = emp;  //OK: Upcasting
Object o3 = new Employee( "Sandeep", 378, 65000.50f );  //OK: Upcasting
```

- In case of upcasting, using super class reference we can access:
  - Inherited fields of super class(non private)
  - Inherited methods of super class(non private)
  - Overriden methods of sub class.
- In case of upcasting, if we want to access fields/non overriden methods of sub class then we should do downcasting.

```
//Employee this = emp1
//Object obj = emp2;    //Upcasting
@Override
public boolean equals( Object obj ) {
    if( obj != null ) {
        Employee other = (Employee) obj;    //Downcasting
        if( this.empid == other.empid)
            return true;
    }
    return false;
}
```

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Employee other = (Employee) obj;
    return empid == other.empid;
}
```

System Date and Time

- Using Calendar class

```
import java.util.Calendar;
public class Program {
    public static void main(String[] args) {
        Calendar c = Calendar.getInstance();
        //int day = c.get(Calendar.DAY_OF_MONTH);
```

```
        int day = c.get(Calendar.DATE);
        int month = c.get(Calendar.MONTH) + 1;
        int year = c.get(Calendar.YEAR);
        System.out.println(day+" / "+month+" / "+year);
    }
}
```

- Using Date class

```
public static void main(String[] args) {
    Date date = new Date();
    int day = date.getDate();
    int month = date.getMonth() + 1;
    int year = date.getYear() + 1900;
    System.out.println(day+" / "+month+" / "+year);
    }
```

- Using Localdate class

```
    public static void main(String[] args) {
        LocalDate ld = LocalDate.now();
        int day = ld.getDayOfMonth();
        int month = ld.getMonthValue();
        int year = ld.getYear();
        System.out.println(day+" / "+month+" / "+year);
    }
```

Methods of java.lang.Object class

1. public String toString( );
2. public boolean equals(Object obj);
3. public native int hashCode( );
4. protected native Object clone( ) throws CloneNotSupportedException
5. protected void finalize( )throws Throwable
6. public final native Class<?> getClass();
7. public final void wait( )throws InterruptedException
8. public final native void wait(long timeout)throws InterruptedException
9. public final void wait(long timeout, int nanos)throws InterruptedException
10. public final native void notify( );
11. public final native void notifyAll( );

Marker interfaces

- Interface that doesn't contain any method declaration is called as "Marker interface".
- These interfaces are used to mark or tag certain functionalities/features in implemented class. In other words, they associate some information (metadata) with the class.

- Marker interfaces are used to check if a feature is enabled/allowed for the class.
- Marker interface do not contain methods. But by implementing marker interface, compiler generate metadata for JVM. Which helps JVM to perform some action(to serialize /deserialize instance, to clone instance, to transfer instance remotely).
- Java has a few pre-defined marker interfaces. e.g. Serializable, Cloneable, etc.
    - java.io.Serializable -- Allows JVM to convert object state into sequence of bytes.
    - java.lang.Cloneable -- Allows JVM to create copy of the class object.

Shallow Copy

- Process of copying state of variable into another variable is called shallow copy.

```
int num1 = 10;
int num2 = num1;    //Shallow Copy of int variable
```

```
Date dt1 = new Date( 6,10,2022);
Date dt2 = dt1;    //Shallow Copy of reference variable
```

Cloneable interface

- Enable creating copy/clone of the object.
- If a class is Cloneable, Object.clone() method creates a shallow copy of the object. If class is not Cloneable, Object.clone() throws CloneNotSupportedException.
- A class should implement Cloneable and override clone() to create a deep/shallow copy of the object.
- If we want to create shallow copy of current instance then we should use clone method.
- clone is non final but native method of java.lang.Object class.
- Syntax:
    - protected native Object clone( ) throws CloneNotSupportedException
- By convention, the returned object should be obtained by calling super.clone. In other words, if we want to create new instance from exisiting instance then we should use super.clone() inside overriden clone method.
- Example:
    - java.lang.Cloneable
    - java.util.EventListener
    - java.util.RandomAccess
    - java.io.Serializable
    - java.rmi.Remote

- Without implementing Cloneable interface, if we try to create clone of instance then clone() method throws CloneNotSupportedException.

```
class Date implements Cloneable{
    private int day;
    private int month;
    private int year;

    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public Date clone( ) throws CloneNotSupportedException {
        Date other = ( Date )super.clone();
        return other;
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        try {
            Date dt1 = new Date( 6,10,2022);
            Date dt2 = dt1.clone(); //Shallow copy of instances

        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

- Deep Copy

```
class Date implements Cloneable{
    private int day;
    private int month;
    private int year;

    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public Date clone( ) throws CloneNotSupportedException {
        Date other = ( Date )super.clone();
        return other;
```

```
    }
}
```

```
class Employee implements Cloneable{
    private String name;    //null
    private int empid;      //0
    private float salary;   //0.0
    private Date joinDate;  //null
    public Employee(String name, int empid, float salary, int day, int month, int
year ) {
        this.name = name;
        this.empid = empid;
        this.salary = salary;
        this.joinDate = new Date( day, month, year );
    }

    public Employee clone( ) throws CloneNotSupportedException {
        //Deep Copy
        Employee other = (Employee) super.clone();
        other.joinDate = this.joinDate.clone();
        return other;
    }
}
```