Day 15

Fragile Base Class Base Problem

- If we make changes in the super class then it necessary to compile super class as well as all its sub classes. It is called as fragile base class problem.

Interface

- Set of rules is called as specification/standard.
- If we want to define specification for the sub classes then we should use interface in Java.
- We should use interface because:
    1. It helps to build trust between service provider and service consumer.
    2. It helps to minimize vendor dependancy. In other words, we can achieve loose coupling.
- Reference types in Java
    1. Interface
    2. Class
    3. Enum
    4. Array
- Interface is non primitive/reference type in Java.
- interface is keyword in Java.
- Syntax

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

- We can not instantiate interface but we can create reference it.

```
public static void main(String[] args) {
    printable p = null; //reference //OK
    p = new printable( );   //Not OK
}
```

- In Java, inside interface we can declare:
    1. Nested interface
    2. Field
    3. Abstract method
    4. Default method
    5. Static method
- We can define interface inside interface. It is called nested interface.

```
package java.util;
public interface Map<K,V>{
    public static interface Entry<K,V>{    //Nested interface
        K getKey( );
        V getValue( );
        V setValue( V value );
    }
}
```

- We can declare/define methods inside interface(I/F) but we can not declare/define constructor(ctor) inside interface.
- We can declare fields inside interface. It is by default "public static final".

```
interface A{
    int number = 10;
    //public static final int number = 10;
}
public class Program {
    public static void main(String[] args) {
        System.out.println(A.number);
    }
}
```

- Consider another example

```
javap java.sql.ResultSet    (press enter )
```

```
package java.sql;
interface ResultSet{
    public static final int HOLD_CURSORS_OVER_COMMIT = 1;
    public static final int CLOSE_CURSORS_AT_COMMIT = 2;
    public static final int FETCH_FORWARD = 1000;
    public static final int FETCH_REVERSE = 1001;
    public static final int FETCH_UNKNOWN = 1002;
    public static final int TYPE_FORWARD_ONLY = 1003;
    public static final int TYPE_SCROLL_INSENSITIVE = 1004;
    public static final int TYPE_SCROLL_SENSITIVE = 1005;
    public static final int CONCUR_READ_ONLY = 1007;
    public static final int CONCUR_UPDATABLE = 1008;
}
```

- We can write method inside interface. It is by default considered as public and abstrasct.

```

```

```
interface A{
    void print( ) ;
    //public abstract void print( ) ;
}
```

```
package java.util;
public interface Enumeration{
    boolean hasMoreElements();
    E nextElement();
}
```

- Using implements keyword, we can implement interface for the class.
- It is mandatory to implement all the abstract methods of interface in sub class otherwise sub class can be considered as abstract.
- Solution 1:

```
interface A{
    int number = 10;    //public static final int number = 10;
    void print( );      //public abstract void print( ) ;
}
abstract class B implements A{
}
```

- Solution 2:

```
interface A{    //ISI
    int number = 10;
    //public static final int number = 10;
    void print( ) ;
    //public abstract void print( ) ;
}
class B implements A{   //Service Provider
    @Override
    public void print() {
        System.out.println("Number  :   "+A.number);
    }
}
```

```
public class Program {  //Service Consumer
    public static void main(String[] args) {
        B b = new B( );
        b.print();  //OK : 10

        A a = new B( ); //Upcasting : Recommended
```

```
        a.print();  //OK : 10
    }
}
```

Choose correct syntax

- I1, I2, I3 => Interfaces
- C1, C2, C3 => Classes

1. I2 implements I1 //NOT OK
2. I2 extends I1 //OK
3. I3 extends I1, I2 //OK
4. I1 extends C1; //Not OK
5. C1 extends I1; //Not OK
6. C1 implements I1; //OK
7. C1 implements I1, I2; //OK
8. C2 implements C1; //Not OK
9. C2 extends C1; //OK
10. C3 extends C1, C2; //NOT OK
11. C2 implements I1 extends C1; //Not OK
12. C2 extends C1 implements I1 ; //OK
13. C2 extends C1 implements I1, I2 ; //OK

- Conclusion:
    1. Interace extends interface.
    2. Interface can extend more than one interfaces. In other words, Java support to multiple interface inheritance.
    3. Super type of interface must be interface.
    4. Class do not extend interface rather it implements interface. It is called interface implementation inheritance.
    5. Class can implement more than one interfaces. In other words, Java support to multiple interface implementation inheritance.
    6. Class can extend another class. It is called implementation inheritance.
    7. Class can not extends more than one class. In other words, Java do not support to multiple implementation inheritance.

Interface fields

```
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{   //OK : Multiple Interface Inheritance.
    int num3 = 30;
    int num4 = 60;
}
public class Program {
    public static void main(String[] args) {
        System.out.println("A.Num5  :   "+A.num5);  //OK    :   70
        System.out.println("B.Num5  :   "+B.num5);  //OK    :   80
        System.out.println("C.Num5  :   "+C.num5);  //NOT OK    :   The
field C.num5 is ambiguous
    }
    public static void main4(String[] args) {
        System.out.println("A.Num4  :   "+A.num4);  //OK    :   40
        System.out.println("B.Num4  :   "+B.num4);  //OK    :   50
        System.out.println("C.Num4  :   "+C.num4);  //OK    :   60
    }
    public static void main3(String[] args) {
        System.out.println("Num3    :   "+C.num3);  //OK    :   30
    }
    public static void main2(String[] args) {
        System.out.println("Num2    :   "+B.num2);  //OK    :   20
        System.out.println("Num2    :   "+C.num2);  //OK    :   20
    }
    public static void main1(String[] args) {
        System.out.println("Num1    :   "+A.num1);  //OK    :   10
        System.out.println("Num1    :   "+C.num1);  //OK    :   10
    }
}
```

Interface Abstract Methods

```
interface A{
    void f1( );
    void f3( );
}
interface B{
    void f2( );
    void f3( );
}
abstract class C{
    public abstract void f3( );
}
class D extends C implements A, B{
    @Override
    public void f1() {
        System.out.println("D.f1");
    }
    @Override
    public void f2() {
        System.out.println("D.f2");
```

```
    }
    @Override
    public void f3() {
        System.out.println("D.f3");
    }
}
public class Program {
    public static void main(String[] args) {
        A a = new D( );
        a.f1();
        a.f3();

        B b = new D( );
        b.f2();
        b.f3();

        C c = new D( );
        c.f3();
    }
}
```

Abstract Method versus default method.

- It is mandatory to override abstract method of I/F inside sub class.
- We can not provide body to the abstract method.
- It is optional to override default method of I/F inside sub class.
- It is madatory to provide body to the default method.
- Using "InterfaceName.super.methodName()", we can use default method of I/F inside sub class.
- Interface static methods are helper method which are not designed to override rather it is designed to help to default method and sub class methods.

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

```
        a.f1();

        B b = new C( );
        b.f1();
    }
}
```

- If name of default method of super I/F is same then overriding that method is mandatory.
- An interface which contains Single Abstract Method(SAM) is called Functional Interface.
- Functional Interface is also called SAM interface.
- It can contains:
    1. multiple default methods
    2. multiple static methods
    3. Single abstract method.

```
interface A{    //OK
    void f1( );
}
```

- If we want to ensure whether interface is Functional or not then we should use @FunctionalInterface annotation.

```
@FunctionalInterface
interface A{    //OK
    void f1( );//functional method or method descriptor.
}
```

- Abstract method defined inside functional interface is called functional method or method descriptor.
- java.util.function package contains all SUN/ORACLE supplied functional interface.
- Functional interfaces declared in java.util.function package are:
    1. Predicate
        - boolean test(T t)
    2. Consumer
        - void accept(T t)
    3. Supplier
        - T get()
    4. Function<T,R>
        - R apply(T t)
    5. UnaryOperator

Following are commonly used interfaces in Core Java:

```
1. java.lang.AutoCloseable
2. java.lang.Cloneable
```

```
3. java.lang.Iterable
4. java.util.Iterator
5. java.lang.Comparable
6. java.util.Comparator
7. java.io.Serializable
```

Comparable and Comparator Interface implementation

- if we want to sort array then we should use Arrays.sort() method.
- In case of primitive types, Arrays.sort( ) implicitly use Dual-Pivot Quicksort algorithm.

```
public class Program {
    private static void print(int[] arr) {
        if( arr != null )
            System.out.println(Arrays.toString(arr));
        System.out.println();
    }
    public static void main(String[] args) {
        int[] arr = new int[] { 5, 1, 4, 2, 3 };
        Program.print( arr );   //[5, 1, 4, 2, 3]
        Arrays.sort(arr);   //Dual-Pivot Quicksort
        Program.print( arr );   //[1, 2, 3, 4, 5]
    }
}
```

```
Employee[] arr = new Employee[ 14 ];

arr[ 0 ] = new Employee(7369,"SMITH","CLERK","1980-12-
17",800.00f,"RESEARCH");

arr[ 1 ] = new Employee(7499,"ALLEN","SALESMAN","1981-02-
20",1600.00f,"SALES");

arr[ 2 ] = new Employee(7521,"WARD","SALESMAN","1981-02-
22",1250.00f,"SALES");

arr[ 3 ] = new Employee(7566,"JONES","MANAGER","1981-04-
02",2975.00f,"RESEARCH");

arr[ 4 ] = new Employee(7654,"MARTIN","SALESMAN","1981-09-
28",1250.00f,"SALES");

arr[ 5 ] = new Employee(7698,"BLAKE","MANAGER","1981-05-
01",2850.00f,"SALES");

arr[ 6 ] = new Employee(7782,"CLARK","MANAGER","1981-06-
09",2450.00f,"ACCOUNTING");
```

```
arr[ 7 ] = new Employee(7788,"SCOTT","ANALYST","1982-12-
09",3000.00f,"RESEARCH");

arr[ 8 ] = new Employee(7839,"KING","PRESIDENT","1981-11-
17",5000.00f,"ACCOUNTING");

arr[ 9 ] = new Employee(7844,"TURNER","SALESMAN","1981-09-
08",1500.00f,"SALES");

arr[ 10 ] = new Employee(7876,"ADAMS","CLERK","1983-01-
12",1100.00f,"RESEARCH");

arr[ 11 ] = new Employee(7900,"JAMES","CLERK","1981-12-
03",950.00f,"SALES");

arr[ 12 ] = new Employee(7902,"FORD","ANALYST","1981-12-
03",3000.00f,"RESEARCH");

arr[ 13 ] = new Employee(7934,"MILLER","CLERK","1982-01-
23",1300.00f,"ACCOUNTING");
```

- Comparable is an interface declared in java.lang package.
- "int compareTo(T other)" is a method of java.lang.Comparable I/F.
- If we want to sort array of instances of same class then class must implementation java.lang.Comparable interface.
- return type of compareTo method is integer.
    1. If state of current instance is less than specified instance then we should return any -ve number(-1).
    2. If state of current instance is greater than specified instance then we should return any +ve number(+1).
    3. If state of current instance is equal to specified instance then we should return 0.
- Comparator is interface declared in java.util package.
- "int compare(T o1,T o2)" is a method of java.util.Comparator interface.
- If we want to sort array of instances of different class then we should implement java.util.Comparator interface.
- return type of compare method is integer.