Interface

Objectives

- Analyse when to use interface to write Java programs,
- Use standard interfaces in JDK to solve a problem

Table of Content

Definition

- Like class, interface is also used to define a new type.
- An interface is a special type of construct that may have
  - some constants : **`static and final`**
  - some methods listed but with no implementations: **`abstract`**
- All the members of interface are **`public`**
- The modifiers (**`public, abstract static and final)`** need not be explicitly specified.
- Interface cannot be instantiated.
- They don't automatically inherit from **`Object`** class.
- They cannot have any java statements except declarations.
- A class can inherit from more than one interface.

Syntax

Interface definition:

```
interface interface_name
{
[datatype variable_name=value; ]
[returntype method_name();]
}
```

Class implementing interface:

```
class class_name [extends class] implements
   interface_name_1 [, interface_name_2 …
   interface_name_n]
{
// implements methods in the interface_name
}
```

Like classes, interfaces can also be defined inside packages. Therefore, they have public or package access.

Interface Example

```
public interface Shape {
   double PI=3.14;
   void area();
}
class  Circle implements Shape {
   private double radius;
   Circle(double r){radius=r;}
@Override
public void area(){
      System.out.println(PI* radius* radius);
}
public static void main(String a[]){
   Shape s= new Circle(10);//or Circle c= new Circle (10);
   s.area();
   System.out.println(Shape.PI+ " "+ Circle.PI);
}}
```

Compiler automatically inserts **`public, static and final`**

Compiler automatically inserts **`public and abstract`**

?

*If **`public`** is omitted, a compilation error occurs. Why?*

instanceof

```
1. Circle c= new Circle();
   System.out.println(c instanceof Shape );// prints true
2. Student s= new Student("Mary");
   System.out.println(s instanceof Shape );//prints false
   System.out.println(s instanceof Circle);
// gives compilation error
3. class Square implements Shape{}
   Square sq= new Square();
   System.out.println(sq instanceof Circle);
   // gives compilation error
```

Interface and casting

- A class implementing an interface is automatically converted to the interface type.

  ```
  Shape s= new Circle(10);//
  ```

- To convert an interface reference back to the original class type requires explicit casting
- `Circle c =(Circle)s;`
- Any reference can be converted to interface type by explicit casting.

  ```
  Student s= new Student("Meera");
  Shape s1=(Shape) s; //no error
  ```

  But `Circle c=(Circle)s; // error`

extending interface

```
interface X {void x();}
interface Y extends X{void y();}
class Z implements Y{
   public void x(){}
   public void y(){}
}
```

?

What is the difference between abstract class and interface?

Tell me why?

We can incorporate the same logic using abstract class.

```
public abstract class Shape {
double PI=3.14;
void area();}
```

Then why do we need interfaces?

Yes abstract classes can be used instead of interfaces. But one drawback of abstract class is once a class extends abstract class, it cannot extend from any other class since Java does not support multiple inheritance.

That is, if you replace Shape interface with the above class, then Circle class cannot inherit from any other class.

If you have to reuse some implementation, then use inheritance via class.

But what other use does inheritance have?

Hint: What does polymorphism offer?

Uses

- Interfaces are used to
  1. share constants
  2. set standards/define contracts
  3. tag a class, so objects of its class can represent another type
  4. overcome the issues that arise because Java does not support multiple inheritance.

Shared constants

```
public interface TeachingStaff {
    int RETIREMENT_AGE=60;
    int MAX_SUBJECTS=2;
 }

class AnyClass{
      void f(){
      System.out.println(TeachingStaff.RETIREMENT_AGE);
      }
}
```

Exercise

- *A scientific application needs to use several constants through out the application. It was decided to create an interface called PhysicalConstants. The constants that needs to be set are*
  - *Speed of light in vacuum (C):299 792 458 m/s*
  - *Gravitational constant (G): 6.674 28×10^{-11} m^3 kg^{-1} s^{-2}*
  - *Standard Gravitational Acceleration(g) : 9.806 65 m/s2*
- *Use these constants for a class that has following functions.*
  - *E= MC2*
  - *F=G (m1 m2)/r2*
  - *d=0.5 gt*
  - *Make sure that constants names are used without repeating interface names with constants(Hint : use static imports).   (30 mins)*

Set standards/ define contracts

I am not sure if I will fit into those non-standard holders !

Most devices have adopted me as standard. I am cool!

- JSE has many classes with many common programming utility methods.
- Classes that needs to make use of those methods must abide by some rules (or contracts).
- This is because the utility methods sometimes may depend on the methods defined in classes that use them to implement certain parts of functionality. The contracts are imposed through interfaces.
- One such interface is **`Comparable`** and the class that uses this interface is **`Arrays.`**

Arrays

java.util.Arrays

`int binarySearch(<primitiveType> a[], <primitiveType> key)`
`void sort(<primitiveType> [] a)`
`int binarySearch(Object[] a,Object key)`
`void sort(Object[] a)`

- **`Arrays`** class in JSE has utility methods such as sorting, searching implemented efficiently **.**
- All the methods in this class are **`static`**.
- **`binarySearch`** requires the array be sorted prior to making this call. If it is not sorted, the results are undefined. The search location begins from 0.

Comparable

```
java.util.Arrays
int binarySearch(<primitiveType> a, <primitiveType> key)
void sort(<primitiveType> [] a)
int binarySearch(Object[] a,Object key)
void sort(Object[] a)
```

```
<<interface>>
java.lang.Comparable<T>
int compareTo(T o)
```

Replace T refers to any reference type

- **`Arrays`** class requires objects of **`Comparable`** type to be passed to its methods so that utilities like sort and search could be used. Otherwise an exception will be thrown by the methods like sort and search at runtime.

Activity

- Have you come across **`int compareTo(T o)`** methods in any predefined classes ?
- List all the predefined classes that you have learnt so far and determine if their objects are **`Comparable`**.

Implementing `Comparable`

```
public class Student implements Comparable<Student>{
 public int compareTo(Student o) {
   return (int)(regNo - o.regNo);
   }
…
}
```

Exercise

Given a unsorted list of 10 athletes nominated for the coaching class, provide a way for the coach to search for the athlete name and provide grades. Finally print the list of athletes' names with their grade in the sorted order of their names.

Hint: Use **`Arrays`** *class methods.*

(30 mins)

Activity

- Check out what happens if the class using **`Arrays sort`** and **`binarySearch`** methods does not implement Comparable.
- Find out the other methods that **`Arrays`** class implement. Does all the methods need **`Comparable`** objects?

Arrays and **Comparator**

- For the objects of a class that need to be sorted in more than one way, **`Arrays`** class provides another sort method.

  ```
  public static void sort(Object[] a, Comparator c)
  ```

- This method expects the comparison method to be sent through **`Comparator`** object.
- **`java.util.Comparator<T>`**
  - **`int     compare(T o1, T o2)`**
  - **`boolean equals(Object obj)`**

- The **`equals`** is implemented in the **`Object`** class, so there will be no compilation error if this method is not included in the class.
- However, **`Comparator`** explicitly specifies it in the class so that this is implemented appropriately.

Example: Comparator

```
import java.util.Arrays;
import java.util.Comparator;
import student.Student;
public class NameSortStudent implements
Comparator<Student> {

public int compare(Student o1, Student o2){
return o1.getName().compareTo(o2.getName());
}
public static void main(String[] args) {
Student s[]= new Student[3];
s[0]=new Student("Rani");
s[1]= new Student("Sani");
s[2]= new Student("Ravi");
Arrays.sort(s,new NameSortStudent());
for(Student s1:s)
System.out.println(s1.getName());
}}
```

Exercise

- *Provide a way to print list of athletes sorted by the grade as well.*

(30 mins)

Marker class

- Interfaces that do not have any methods or constants are called Marker interfaces.
- Marker interfaces are used to tag a class so that the class is of the interface type.
- Some examples of marker interface in JSE are:

a) **`Cloneable`**

b) **`Serializable`**

c) **`Remote`**

*Read and understand the need of **`Cloneable`** interface.*

Exercise

- *Shape abstract class has 2 abstract methods*
  - *area()*
  - *volume()*
- *Classes Cube, Rectangle, Triangle and Sphere are created . For Rectangle, Triangle volume returns -1.*
- *Shapes that implement volume must be of type Spatial which is a marker interface.*
- *The user enters 5 shape objects which is stored in an array.*
- *Finally, all the Shape objects are printed. Only for Shape object of Spatial type, volume is printed.*

Hint:

Triangle Area = 1/2 of the base X the height,

Rectangle Area: l X w

Sphere Area= $4pr^2$ *, Volume =* $4/3\ pr^3$

Cube Area = 2lw + 2lh + 2wh Volume = l X w X h

Where l is length, w is width and h is height   (30 mins)

Summary

- An interface is a special type of construct that may have some constants and some methods listed but with no implementations.
- All the members of interface are public and Interface cannot be instantiated.
- A class implementing an interface is automatically converted to the interface type.
- Arrays class requires objects of Comparable type to be passed to its methods so that utilities like sort and search could be used.
- Interfaces that do not have any methods or constants are called Marker interfaces.
- Some examples of marker interface in JSE are Cloneable, Serializable, Remote.