Interface

Naveen Kumar K S

Adith.naveen@gmail.com

http://naveenks.com

Definition

- 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.

Interface as type

- Like class, interface is also used to define a new type.
- A class can inherit from an interface. A class inheriting from an interface is of interface type.
- Also a class can inherit from more than one interface (but not from more than one class)
- Any class that inherits from an interface must provide implementation for all the methods of the interface if it has to be a concrete class.
- Hence interface serves as a mechanism to bind a class contractually to implement methods specified in the interface.
- Usually project design at a very high level define interfaces to create a framework.

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.

Example

```
public interface Shape {
double PI=3.14;
void area();
}
class  Circle implements Shape {
private double radius;
Circle(double r){radius=r;}

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();}
}
```

Points to note

```
public interface Shape {
double PI=3.14;
//Compiler automatically inserts public, static and final
void area();
// Compiler automatically inserts public and abstract
}
class  Circle implements Shape {
private double radius;
Circle(double r){radius=r;}
public void area(){System.out.println(PI* radius* radius);
}
public static void main(String a[]){
Shape s= new Shape(); // compilation error
System.out.println(Shape.PI+ " "+ Circle.PI);
// both prints 3.14
}}
```

If **public** is omitted a compilation error occurs.

Why?

instanceof

- **instanceof** returns **true** or **false** when interface names are used except in case of **final** classes. In case of **final** class, it results in compilation error.
- It never gives a compilation error.

```
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 implementating an interface is automatically converted to the interface type.
- This is very obvious and we used it in the example as well.

```
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

- Interfaces themselves can inherit from one another using extend clause.
- In such cases, the implementing class must implement all the methods of the interface including those from the inherited interface.

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

- For example, when we come to collection framework we will find that root of the framework has `Collection` interface and then we have a list of interfaces that inherit from `Collection` like `List, Set` etc.
- These type of scenarios are common in framework implementation where top of the hierarchy consists significantly of interfaces. This will be more clearer once we understand the use of interface.

Test your understanding

- What is the difference between abstract class and interface?

- Abstract class can have any type of member variables
- Abstract class can even have all methods implemented or can have partial implementation or no implementation at all.
- Interface does not fall into Object hierarchy.
- Class can inherit only from one abstract class but many interfaces.

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 interface can be used instead of abstract classes. 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. used to share constants
  2. used to set standards/define contracts
  3. used just to 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

- A large application may have lot of constants that will be used by many or all parts of the application. Java does not provide facility to have global variables. But this disadvantage can be overcome by using interface that serves as a convenient place to put all the shared constants.

```
public interface TeachingStaff {
    int RETIREMENT_AGE=60;
    int MAX_SUBJECTS=2;        }
class AnyClass{
void f(){
System.out.println(TeachingStaff.RETIREMENT_AGE); }}
```

or

```
import static teacher.TeachingStaff.*;
class AnyClass{
void f(){System.out.println(MAX_SUBJECTS);}
}
```

Set standards/ define contracts

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

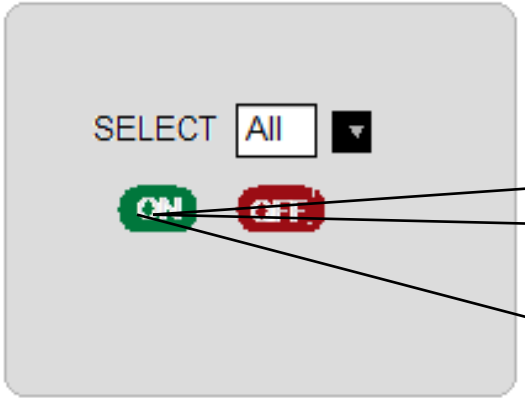
Most devices have adopted me as standard. I am cool!

- When a large project is designed there are many teams with each team focusing on some aspect of the application
- At some point the applications of each of these teams will have to be integrated to work together.
- Therefore it is important that all the teams have a common agreement on how they expose parts of their application so that applications can interact with each other.
- Thus, interfaces are created as a standard for the application and teams to work with this as baseline.

Scenario for defining contracts

Designing a remote control

Need one remote control to start and stop many devices



Team B: makers of lamp devices

Lamp

Night Lamp

AC

…

Team C: makers of AC and other devices

Team A: makers of remote control

Design –Step 1

Classes identified by Teams

Team A

```
class RemoteControl{
public void pressOn(){}
public void pressOff(){}
}
```

Team B

```
class Lamp{
public void on(){}
public void off(){}
}
class  NightLamp extends Lamp{
public void on(){}
public void off(){}}
```

Team C

```
class AC{
public void switchOn(){}
public void switchOff(){}
}
```

Team A: **`pressOn()`** and **`pressOff()`**
which devices. Where are the devices.
I need a common class to represent all these devices.
This device can be passed a parameter to **`pressOn()`** and **`pressOff()`**

Design -Step 2

Team C : let us have an abstract class called Device from which all the other devices will inherit.

Team B: I cannot extends from Device, because my NightLamp already extends from Lamp… java does not support multiple inheritance!

Team C: Ok then we can make Device an interface!

```
abstract class Device{}
interface Device{}
class Lamp implements Device{...}
class AC implements Device{...}
class NightLamp extends Lamp implements Device {...}
```

Team A: Ok this is fine with us. Now I can represent an array of Devices

```
class RemoteControl{
Device devices[];
public void pressOn(){
for(Device d:devices)
d.???
}
}
public void pressOff(){
for(Device d:devices)
d.???
}
}
```

Team A: But now I run into another problem.
Which method on device do I call?
Because while team B is using **on** and **off** as methods, team C is using **switchOn** and **switchOff.**
I think there must be a CONTRACT between **RemoteControl** class and other **Devices**.
And the contract is this. ONLY those devices which provide methods named **on()** and **off()** can be switched on or off by the remote control.

Design -Step 3

Team C: Well then what we can do is have **`on`** and **`off`** methods declared in the **`Device`** interface.
Team B: Yes, this will ensure that we implement **`on`** and **`off`** methods.
Team A: That solves the matter.

```
interface Device{
void on();
void off();
}
```

```
class RemoteControl{
Device d[];
public void pressOn(){
for(Device d:devices)
d.on();
}
public void pressOff(){
for(Device d:devices)
d.off();
}}
class Lamp implements Device{
public void on(){}
public void off(){}
}
```

```
interface Device{
void on();
void off();
}
class AC implements
   Device{
public void switchOn(){}
public void switchOff(){}
}
class  NightLamp extends
Lamp{
public void on(){}
public void off(){}
}
```

Team C: I have a new device which got added -geyser.

Team A: No problem.. Just implement Device interface

Team C: Interface saved us. It helps application scale and extend !

An example from JSE on contracts

- JSE has many classes that does lot of common programming utility methods.
- But for our classes to make use of those methods, we must abide by some rules (or contracts).
- This is because the utility methods need to call back the methods defined in our classes to implement certain parts of functionality.
- Let us understand this by taking an example.
- **`Comparable`** class in **`java.util`** package is a good example to understand this concept.

Comparable

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** like _____class.
- **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.
- How can we make use of these say for Student class?
- Let us understand the complexity it involves.

Student class

Can you sort my objects too. I don't want to remember the faster sorting method algorithm. I have to spend time to learn, implement test. Please sort my objects too.

Well, Circle also asked me the same question the other day. My dilemma is how will I know which of the fields of your class is to be used to compare? Let us do this, you provide me implementation for compare which I will call from my sorting algorithm .

`java.util.Arrays`

And how am I supposed to do that. I can pass student objects to your methods … how can I pass methods… we don't have function pointers in Java?

That is not a problem. I will have an arrangement with **`Comparable`** interface of JSE. It has a method called **`compareTo`** which returns an **`int`** . If you implement **`Comparable`** then I can call this method on your object when I want to compare.

Oh that's good idea. I shall implement **`compareTo`** method.

No ! you need to implement the **`Comparable`** interface and implement **`compareTo`** method. Because I need a guarantee that you have indeed implemented and so I am going to test if your class implements **`Comparable`**!

Yes, boss

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 with the type of class that you are going to use

- **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.
- So to make use of the methods in **Arrays** class, we must implement **Comparable** and provide implementation for **compareTo()**.

Example1: Comparable

```
import java.util.Arrays;
class  Circle implements Shape, Comparable<Circle> {
private double radius;
Circle(double r){radius=r;}
public void area(){
System.out.println(PI* radius* radius);}
public String toString(){
return "Circle with radius :"+radius;}
public int compareTo(Circle o){
return (int)(radius*100 -o.radius*100);
}
```

```
public static void main(String a[]){
Shape s[]= new Shape[3];
s[0]= new Circle(12);
s[1]= new Circle(10);
s[2]= new Circle(15);
Arrays.sort(s);
for(Shape s1:s)
System.out.println(s1);
}
}
```

Result:
Circle with radius :10.0
Circle with radius :12.0
Circle with radius :15.0

Interface objects only carrying method implementation

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

Hey buddy, you did fantastic job by providing a way to sort my Student objects.
But you know, sometimes other classes need to sort my objects by name! I can only provide one **`compareTo`** method… I am just not sure how to deal with this problem.

Oh! You just tell them that they need to provide me an object that implements **`Comparator`**. **`Comparator`** interface also has **`compare`** method. In this they can provide implementation for comparing 2 student object by name.

That is too good .. I don't have to implement anything and they can sort in any way they want

Arrays and Comparator

- `java.util.Arrays:`
  - `public static void sort(Object[] a, Comparator c)`
- `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());
}}
```

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`**

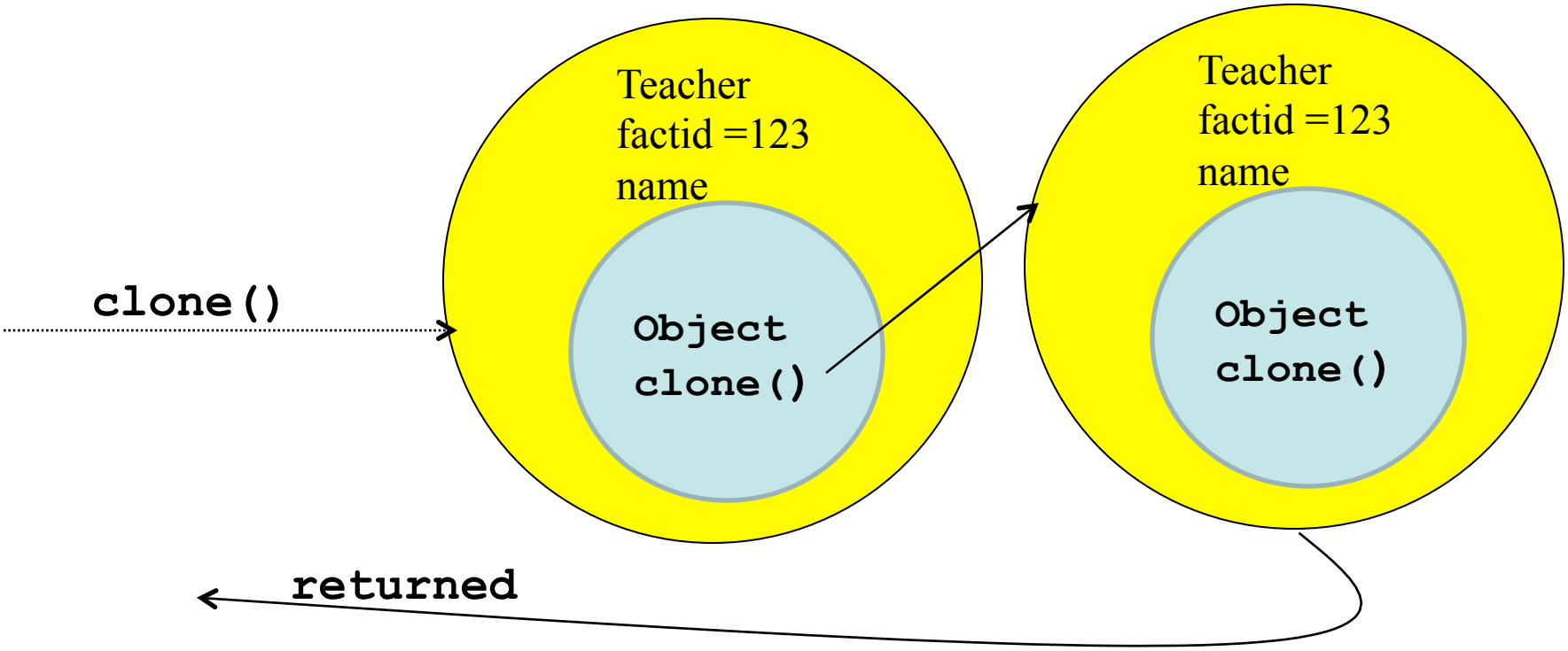
We will look at **`Cloneable`** *now.* **`Serializable`** *will be done in IO.*

`clone()` method of `Object` class

```
protected Object clone() throws
    CloneNotSupportedException{}
```

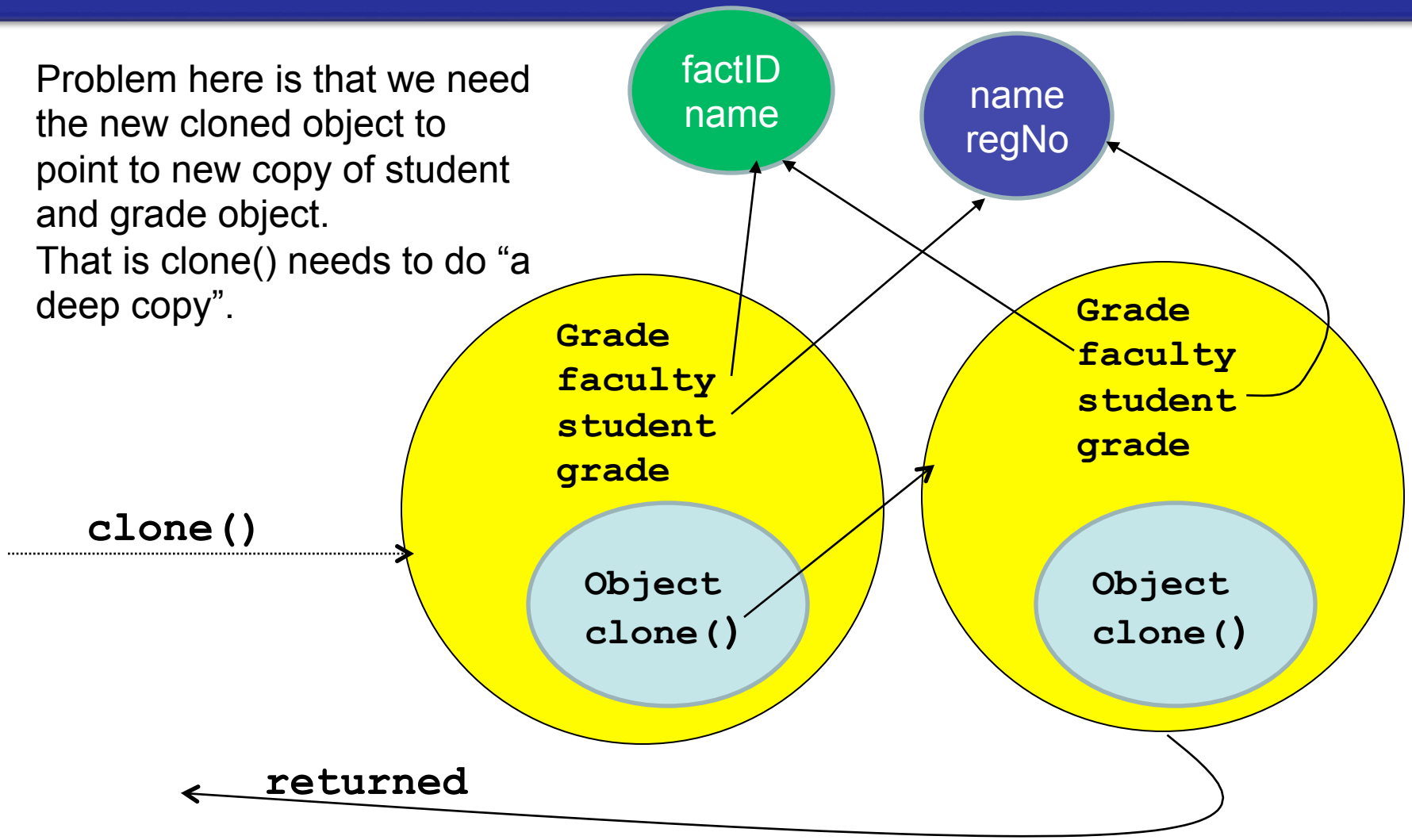
- **`clone()`** method in the **`Object`** class is intended to create copies of an object.
- **`clone()`** method works appropriately for other classes that do not contain any object references as their attributes because the **`Object`** class implementation is just a bit-wise-copy .
- If classes contains references then **`clone()`** method must be overridden.
- For example , the **`clone()`** method works fine for **`Student`** and **`Teacher`** object but is not logically correct for the **`Grade`** class.

Scenario 1 : `clone()`



Scenario 2 : `clone()`

Problem here is that we need the new cloned object to point to new copy of student and grade object.
That is clone() needs to do “a deep copy”.



clone()- protected method

- What we observer is that **`clone()`** works for the class contains only primitive fields or references to immutable objects needs to be overridden for others.
- That is the reason why **`Object`** class has made **`clone()`** **`protected`**. All classes that want to expose **`clone()`** must override the method to be **`public`**. This is true even for the classes for which the implementation of **`clone()`** by **`Object`** class is fine.

`Cloneable` interface

- **`clone()`** method throws **`CloneNotSupportedException`** if a class that implements **`clone()`** method is not of type **`Cloneable`**
- Classes overriding this method can choose to ignore this exception by removing it from declaration. However this is not recommended. By convention, classes overriding **`clone()`** must return object that is returned by calling **`super.clone`**. If this is done then **`x.clone().getClass() == x.getClass()`** will be true. If **`super.clone()`** is called then the method must either catch exception explicitly or must have this exception list . *(We will understand this in exception class)*
- Classes that are interested in calling **`clone()`** method can check if the object is **`Cloneable.`** If so , it is sure that the object has implemented the **`clone()`** method appropriately.

Example: clone()

```
public class Student extends general.Person
implements Cloneable{
public Object clone()throws
CloneNotSupportedException{
return super.clone();}
…
}
public class Teacher extends general.Person
implements Cloneable{
public Object clone()throws
CloneNotSupportedException{
return super.clone();}
…
}
```

```
public class Grade implements Cloneable{ {
    private Teacher faculty;
    private Student student;
    private String subjectCode;
    private String grade;

public Object clone()throws
CloneNotSupportedException{
Grade  g=(Grade)super.clone();

// explicitly deep copy
g.faculty= (Teacher)faculty.clone();
g.student = (Student ) student.clone();
return g;
}
}
…
}
```

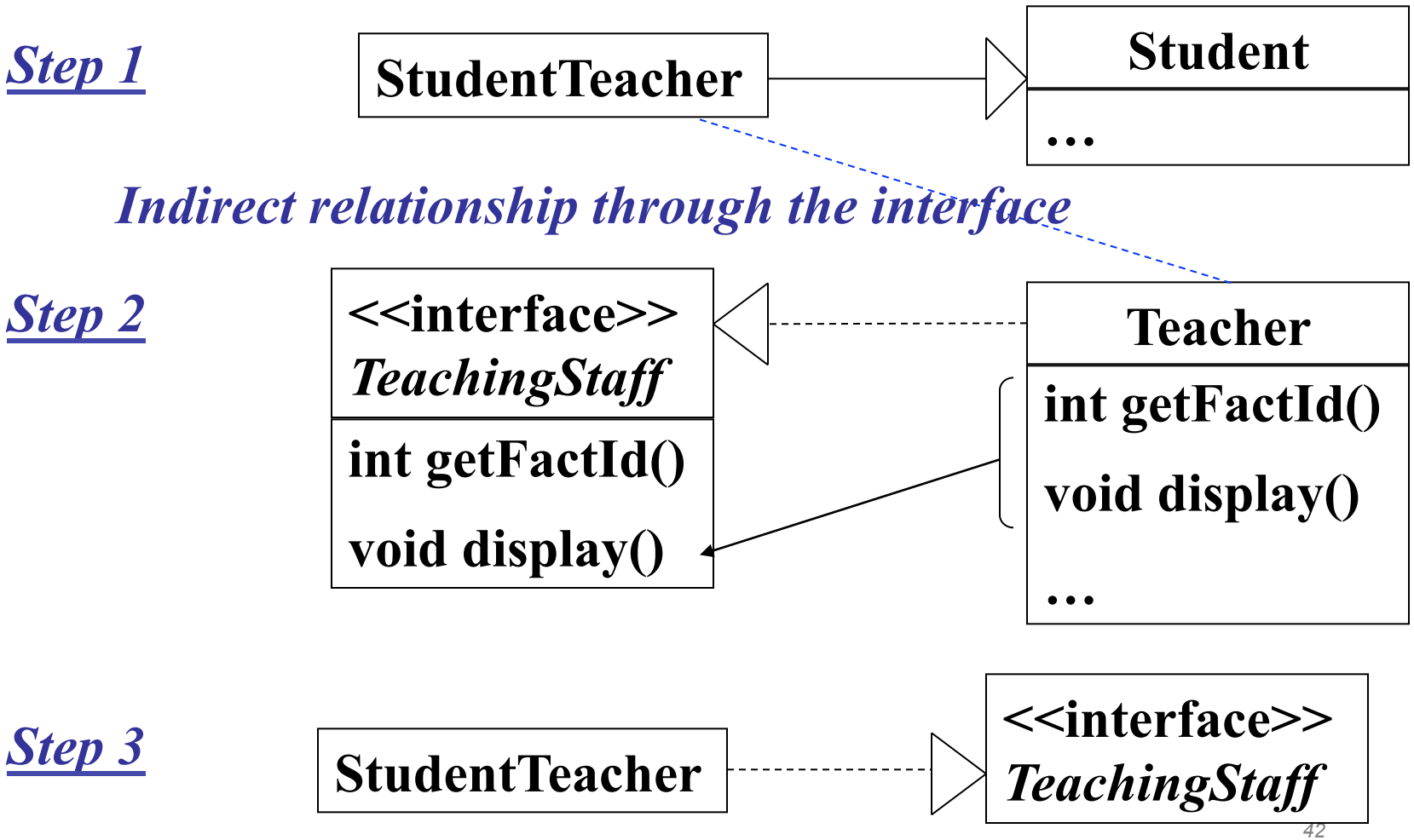
Tell me how?

- How can we represent a teacher who is also a student? Many universities offer programs that allow Ph.D students to teach while they pursue their doctoral programs. Java does not have multiple inheritance. So, we need a way around it.

- Though java does not support multiple inheritance using classes, it allows multiple inheritance using interfaces.
- Idea here is to
  - reuse the methods written Student class and Teacher class
  - Also represent the object as instance of both Student and Teacher when need be.
- Interface and composition could be used together to achieve this.

Scenario: Interface to implement multiple inheritance

- The problem that we have to represent a **`Teacher`** who is also a **`Student`** can be solved in the following way:
  1. **`StudentTeacher`** class will **`extend`** from **`Student`** class→ no problem.
  2. Since we cannot extend **`Teacher`** class, we will create an interface which will act as intermediary between **`StudentTeacher`** and **`Teacher`** class. We shall call this a **`TeachingStaff`** . This interface will have all the methods that are in **`Teacher`** class.
  3. We shall have both **`StudentTeacher`** and **`Teacher`** implement this interface. So any object that is of type **`TeachingStaff`** represents is a teacher and will have to provide implementation for all the methods.
  4. To make sure that we reuse the code written in the **`Teacher`** class in **`StudentTeacher`** also, we create a private **`Teacher`** object inside **`StudentTeacher class.`** The interface methods will indirectly call the methods on this teacher object.

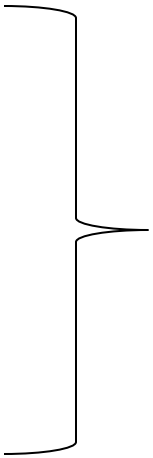

Step 1
StudentTeacher
Student
…
Indirect relationship through the interface
Step 2
<<interface>>
TeachingStaff
int getFactId()
void display()
Teacher
int getFactId()
void display()
…
Step 3
StudentTeacher
<<interface>>
TeachingStaff

Example :Interface to implement multiple inheritance

```
package teacher;
public interface TeachingStaff{
   // important methods of Teacher class}

package teacher;
public class Teacher implements TeachingStaff{
/* implement all the methods of TeachingStaff   */
                    }
package student;

import teacher.*;

public class StudentTeacher extends Student implements
   TeachingStaff{
/*implement all the methods of TeachingStaff*/
}
```

```
package teacher;
public interface TeachingStaff{
int getFactId();
String getName();
// and other methods of teacher class
}
-----------------------------------------
package teacher;
public class Teacher extends general.Person
   implements TeachingStaff{
 public int getFactId(){return factId;}

…
}
```

```
package student;

import teacher.*;

public class StudentTeacher extends Student implements
TeachingStaff{

private Teacher t;

public StudentTeacher(String name) {

super(name);  t= new Teacher(name);        }

public int getFactId()

{return t.getFactId();}

public String getName(){

return t.getName(); }

// and other methods

}
```



Implementing interface methods

```
import teacher.*;
import student.*;
import admin.*;

class Test{
static void listInvigilators(TeachingStaff s[]){
for(int i=0;i<s.length;i++){
if(!(s[i] instanceof StudentTeacher))
System.out.println(s[i].getName());
else
System.out.println("*"+s[i].getName());
}}
public static void main(String str[]){
TeachingStaff f[]=new TeachingStaff[3];
f[0]=new HOD("Ned","1.3.2006");
f[1]=new Teacher("Sam");
f[2]= new StudentTeacher("Rohit");
listInvigilators(f);
}}
```

Invigilators can be any type of teacher (either Teacher or StudentTeacher). `listInvigilators()` method uses `TeachingStaff` interface to hold references of `Teacher` and `StudentTeacher`.