



Advanced Programming Interfaces

What is an interface?

Webster: “a place at which *independent and often unrelated systems* meet and communicate with each other“, respecting a set rules .



Contract, Protocol

- An interface describes a model, a contract.
- A class may implement that model, adhering to the contract and strictly respecting its specifications.



Defining an Interface

```
[public] interface InterfaceName
    [extends SuperInterface1, SuperInterface2...] {
        /* Constant declarations
           Abstract methods
           Default methods
           Static methods */
    }
```

Examples:

```
public interface Student {
    int MAX_GRADE = 5;
    int getExamGrade();
}
//public static final int ...
//public abstract int ...

public interface AutoCloseable {
    void close();
}

public interface Observer {
    update(Observable o, Object arg);
}
```

Implementing an Interface

```
class ClassName implements Interface1, Interface2, ... {  
    /* A concrete class that implements an interface  
       must specify code for all abstract methods  
       declared by the interface */  
}
```

Examples:

```
public class InfoStudent implements Student {  
    public int getExamGrade {  
        return MAX_GRADE;  
    }  
}
```

```
public class EagleEye implements Observer { ... }
```

```
public class FileReader implements AutoCloseable { ... }
```

```
public class Connection implements AutoCloseable { ... }
```

Interface – Reference Type

We may say that an object is of type X, where X is an interface, if that object is an instance of a class implementing the interface X.

```
Student student = new InfoStudent();
```

```
Student student = new Student();
```

```
Observer observer = new EagleEye();
```

```
Observer = new Observer();
```

```
AutoCloseable reader = new FileReader("fis.txt");
```

```
AutoCloseable reader = new AutoCloseable();
```

Multiple Implementations

```
public interface Matrix {
    void set(int row, int col, double value);
    double get(int row, int col);
    Matrix add(Matrix m);
    Matrix mul(Matrix m);
    ...
}

public class DefaultMatrixImpl implements Matrix {
    private double[][] data;
    public DefaultMatrixImpl(int rows, int cols) { ... }
    ...
}

public class SparseMatrixImpl implements Matrix {
    private int[] row;
    private int[] col;
    private double[] data;
    public SparseMatrixImpl(int rows, int cols) { ... }
    ...
}

public static void main ( String args []){
    Matrix a = new DefaultMatrixImpl(10, 10); a.set(0,0, 123); ...
    Matrix b = new SparseMatrixImpl (10, 10); b.set(9,9, 456); ...
    Matrix c = a.add(b);
}
```

Interfaces and Abstract Classes

- Extending an abstract class imposes a strong relationship among two classes.
- Implementing an interface is much lighter: it only specifies that a class is respecting a certain contract, making sure it conforms to some specifications.
- Interfaces and abstract classes do not exclude each other, they are used together in many situations:
 - ♦ **List** → **the contract**
 - ♦ **AbstractList** → **the common behaviour**
 - ♦ **LinkedList,**
ArrayList → **specific behaviour**

Evolving Interfaces

- Add another abstract method to an interface → all classes that implement it will break.

```
public interface Matrix {  
    ...  
    void reset(); //we add a new abstract method to the interface  
}
```

- Anticipate all uses for your interface and specify it completely from the beginning!
- Create a new interface, extending the old one
 - Old classes will use the old interface or upgrade to the new interface.
- Use **default methods**.

Default and Static Methods

- **Default methods** allow extending interfaces without having breaking implementations.

```
public interface Matrix {  
    ...  
    default void reset() {  
        for (int i = 0; i < rowCount(); i++) {  
            for (int j = 0; j < columnCount(); j++) {  
                set(i, j, 0);  
            }  
        }  
    }  
}
```

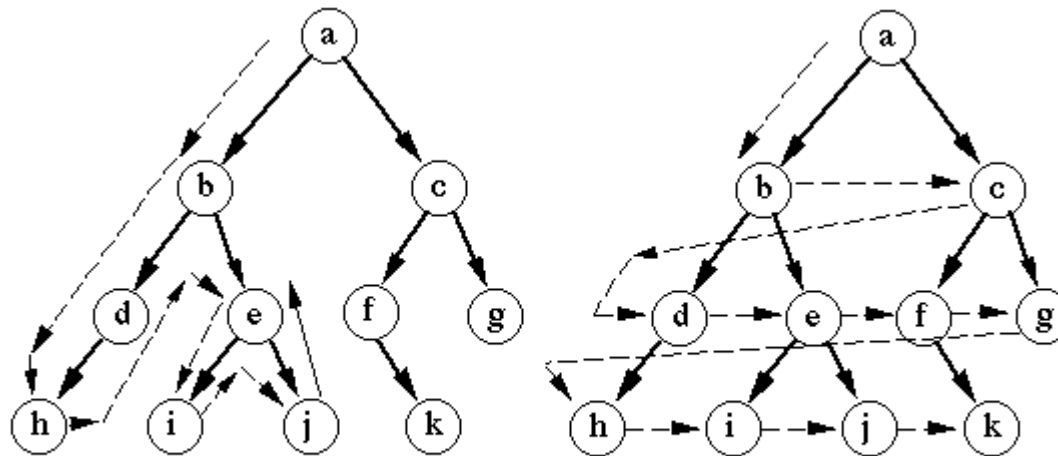
- Classes may override default methods.
- **Static methods** are similar to default methods except that we can't override them.

```
static void log(String str) {  
    System.out.println("Logging: " + str);  
}
```

Callback Methods

A **callback** is something that you pass to a method as an argument (some piece of code).

Example: *When a graph exploration algorithm reaches a node, we want to execute some kind of processing for that specific node.*



Depth-first search

Breadth-first search

Implementing *Callback* Methods

```
public interface NodeProcessing {  
    public void execute(Node u);  
}
```

```
public class Graph {  
    ...  
    public void explore(NodeProcessing f) {  
        ...  
        if (exploration reached the node v) {  
            f.execute(v);  
        }  
        ...  
    }  
}
```

//Implement various processing types

```
class PrintRo implements NodeProcessing {
```

```
    public void execute(Node v) {  
        System.out.println("Nodul curent este: " + v);  
    }  
}
```

```
class PrintEn implements NodeProcessing {
```

```
    public void execute(Node v) {  
        System.out.println("The current node is: " + v);  
    }  
}
```

```
Graph g = new Graph();
```

```
...
```

```
g.explore(new PrintRo());
```

```
g.explore(new PrintEn());
```

FilenameFilter Interface

```
//Listing the files from a folder
import java.io.*;

public class ListFiles {

    public static void main ( String [] args ) {
        File folder = new File(".");
        String[] list = folder.list(new MyFilter("mp3"));
        for (int i = 0; i < list.length ; i ++ ) {
            System.out.println(list[i]);
        }
    }
}

class MyFilter implements FilenameFilter {
    String extension;

    public MyFilter (String extension) {
        this.extension = extension;
    }

    public boolean accept(File dir, String name) {
        return name.endsWith("." + extension);
    }
}
```

Anonymous Classes

Anonymous Class = Inner class used to create an object of a specific type.

```
method(new SomeInterface() {  
    // Implement interface methods  
});
```

Example:

```
folder.list(new FilenameFilter() {  
    // Anonymous class  
    public boolean accept (File dir, String name) {  
        return ( name.endsWith(".txt") );  
    }  
});
```

Compile: OuterClass\$1.class, OuterClass\$2.class, ...



Comparing Objects

```
class Person {
    private int code;
    private String name;
    public Person (int code, String name ) {
        this.code = code;
        this.name = name ;
    }
    ...
    public String toString () {
        return code + " \t " + name;
    }
}

class Sorting {
    public static void main ( String args []) {
        Person persons[] = new Person[4];
        persons[0] = new Person (3, " Ionescu ");
        persons[1] = new Person (1, " Vasilescu ");
        persons[2] = new Person (2, " Georgescu ");
        persons[3] = new Person (4, " Popescu ");
        java.util.Arrays.sort(persons);
        System.out.println ("The persons were sorted...");
        for (int i = 0; i < persons.length ; i++)
            System.out.println (persons[i]);
    }
}
```



The *Comparable* Interface

Imposes a total ordering on the objects of a class (*natural*)

```
public interface Comparable {  
    int compareTo(Object o);  
}
```

```
class Person implements Comparable {
```

```
...
```

```
public int compareTo (Object other) {  
    //returns: 0 if this==other, <0 if this<other, >0 if this>other
```

```
    if (other == null )  
        throw new NullPointerException();  
    if (!( other instanceof Persoana ))  
        throw new ClassCastException ("Uncomparable objects!");
```

```
    Person pers = (Person) other;  
    return (this.code - pers.code);
```

```
}
```

```
}
```


The *Comparator* Interface

A comparison function, which imposes a total ordering on some collection of objects

```
import java.util.*;
class Sorting {
    public static void main ( String args []) {
        Person persons[] = new Person [4];
        persons[0] = new Person (3, " Ionescu ");
        persons[1] = new Person (1, " Vasilescu ");
        persons[2] = new Person (2, " Georgescu ");
        persons[3] = new Person (4, " Popescu ");
        Arrays.sort (persons, new Comparator () {
            public int compare ( Object o1 , Object o2) {
                Person p1 = (Person)o1;
                Person p2 = (Person)o2;
                return p1.getName().compareTo(p2.getName());
            }
        });
        System.out.println("The persons are orderd by name:");
        for (int i = 0; i < persns.length; i++)
            System.out.println (persons[i]);
    }
}
```

Functional Interfaces

- A functional interface is any interface that **contains only one abstract method**.
- Instead of using an anonymous class, you use a *lambda expression*, omitting the name of the interface and the name of the method.

```
Arrays.sort(p, (Object o1, Object o2) -> {  
    Person p1 = (Person) o1;  
    Person p2 = (Person) o2;  
    return p1.getName().compareTo(p2.getName());  
});
```

```
@FunctionalInterface  
public interface Comparator { ...}
```

Method References

- Sometimes a lambda expression does nothing but call an existing method.

```
class Person {  
    ...  
    public static int compareByAge(Person a, Person b) {  
        return a.birthday.compareTo(b.birthday);  
    }  
}
```

```
Arrays.sort(persons, (a, b) -> Person.compareByAge(a,b));
```

- In those cases, it's often clearer to refer to the existing method by name:

```
Arrays.sort(persons, Person::compareByAge);
```

Marker Interfaces

- Interfaces that do not define any method.

```
interface Serializable {}
```

```
interface Cloneable {}
```

- Their role is to associate some *metadata* to a class, that will be useful at runtime.

```
class Person
```

```
    implements Serializable, Cloneable { ... }
```

- A modern alternative: *annotations*

```
@Entity(table="persons")
```

```
class Person { ... }
```

Conclusions

- An interface defines a set of specifications
- They are essential in separating the model from the implementation.
- An interface is a common denominator between unrelated classes.
- Can be used as Reference Data Types
- Can be used to implement *Callbacks*