Interfaces

CPSC 219: Introduction to Computer Science for Multidisciplinary Studies II
Fall 2023

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Thursday, 25 October 2023

UNIVERSITY OF CALGARY

Interfaces

- An interface is a description of a list of things to fulfill

```
public interface Brain {
    public void talk(String name);

    public String getIdea();

    public void doProgramming();
}
```

Interfaces

- An interface is a description of a list of things to fulfill
- If a class **implements** an interface it is declaring it fulfills this functionality

```java
public interface Brain {
    public void talk(String name);

    public String getIdea();

    public void doProgramming();
}
```

```java
public class RealBrain implements Brain {

    @Override
    public void talk(String name) {
        // TODO Auto-generated method stub
    }

    @Override
    public String getIdea() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void doProgramming() {
        // TODO Auto-generated method stub
    }

}
```

Interfaces

- In fact in Java these are all implicitly abstract

```
public interface Brain {
    public void talk(String name);

    public String getIdea();

    public void doProgramming();
}
```

```
public abstract interface Brain2 {
    public abstract void talk(String name);

    public abstract String getIdea();

    public abstract void doProgramming();
}
```

Interfaces

- The interface and the methods must all be public

```
public interface Brain {
    public void talk(String name);

    public String getIdea();

    public void doProgramming();
}
```

```
private interface Brain3 {
    private void talk(String name);

    public String getIdea();

    public void doProgramming();
}
```

Interfaces

- Every instance member is a constant (even if you don't put in final or static)
- Class member variables must be given a value (since they are constants)

```
public interface Brain {
    public void talk(String name);

    public String getIdea();

    public void doProgramming();
}
```

```
public interface Brain4 {

    public int x;
    public int y = 2;
    public static int z = 3;
    public final int a = 4;
    public static final int b = 5;
    int c = 6;
    final int d = 7;

    public void talk(String name);

    public String getIdea();

    public void doProgramming();
}
```

Interfaces

- You can have concrete methods, only if they are static

```
public interface Brain {
    public void talk(String name);

    public String getIdea();

    public void doProgramming();
}
```

```
public interface Brain5 {
    public static void talk(String name) {
        System.out.println(name);
    }

    public String getIdea();

    public void doProgramming();
}
```

Interfaces

- You can implement multiple interfaces

```
public interface Brain {
    public void talk(String name);

    public String getIdea();

    public void doProgramming();
}

public interface Leg {
    public void kick();
}
```

```
public class RealBrain implements Brain, Leg {

    @Override
    public void talk(String name) {
        // TODO Auto-generated method stub
    }

    @Override
    public String getIdea() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void doProgramming() {
        // TODO Auto-generated method stub
    }

    @Override
    public void kick() {
        // TODO Auto-generated method stub

    }

}
```

Interfaces

- Interfaces can extend interfaces

```
public interface Brain {
    public void talk(String name);

    public String getIdea();

    public void doProgramming();
}

public interface BigBrain extends Brain {
    public void thinkBigger();
}
```

```
public class RealBrain implements BigBrain {

    @Override
    public void talk(String name) {
    }

    @Override
    public String getIdea() {
        return null;
    }

    @Override
    public void doProgramming() {
    }

    @Override
    public void thinkBigger() {
    }
}
```

Interfaces

- What is the purpose of this tool?

Interfaces

- What is the purpose of this tool?
- Code-reuse? (not really as static/constants methods aren’t real purpose)
- Shared state? (there are no instance variables)
- Shared behaviour? (we don’t inherit concrete methods (except static))

Interfaces

- What is the purpose of this tool?
- Code-reuse? (not really as static/constants methods aren't real purpose)
- Shared state? (there are no instance variables)
- Shared behaviour? (we don't inherit concrete methods (except static))
- **Shared functionality – We know that the class must fulfill the (abstract) methods outlined. A lot of shared API concepts work this way.**
- **Multiple functionalities – We can guarantee functionality of different interfaces.**

Interfaces

- **Inheritance and polymorphism comes with tight coupling**
  - **Coupling was to our benefit in reducing code for state/behaviour**
- But just because things are capable of the same functions does not mean that they have any state/behaviour that is shared
- Through interfaces we can outline and guarantee (but not share) behaviour, without coupling
  - Yes each implementing class is coupled tightly to the interface, but not to other classes.

Example Interface vs Extends

- We can create for ourselves an interface, we want everything that uses this interface to fulfill a draw function

```
public interface Drawable {
    public void draw();
}
```

- Done correctly we should have comments that explicitly describe what the implementing class should be fulfilling

```
public interface Drawable {
    /**
     * Will print to System.out.println a visualization of the implementing Class
     */
    public void draw();
}
```

Example Interface vs Extends

- We can make abstract class for a Shape and declare that it should implement the draw function and getArea functions

```
import java.util.ArrayList;

public abstract class Shape implements Drawable {
    protected ArrayList<Point> points = new ArrayList<>();

    public abstract double getArea();
}
```

Example Interface vs Extends

```
public class Triangle extends Shape {

    public Triangle(Point p1, Point p2, Point p3) {
        points.add(p1);
        points.add(p2);
        points.add(p3);
    }

    @Override
    public void draw() {
        System.out.println("\u25B3");
    }

    @Override
    public double getArea() {
        Point p1 = points.get(0);
        Point p2 = points.get(1);
        Point p3 = points.get(2);
        return Math.abs((p1.x * (p2.y - p3.y) + p2.x * (p3.y - p1.y) + p3.x * (p1.y - p2.y)) / 2.0);
    }

}
```

Example Interface vs Extends

```
public class Circle extends Shape {
    private double r;

    public Circle(Point centre, double r) {
        points.add(centre);
        this.r = r;
    }

    @Override
    public void draw() {
        System.out.println("\u25CB");
    }

    @Override
    public double getArea() {
        return Math.PI * r * r;
    }

}
```

Example Interface vs Extends

- We can also make a non-shape class implement the Drawable interface.
- There is no coupling here to any Shape class, only to the Draw interface

```
import java.util.ArrayList;

public class List implements Drawable {

    private ArrayList<String> items;

    public List() {
        items = new ArrayList<>();
    }

    public void add(String item) {
        items.add(item);
    }

    @Override
    public void draw() {
        System.out.println("LIST");
        System.out.println("----");
        for (String item : items) {
            System.out.println(item);
        }
    }

}
```

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        List list = new List();
        list.add("item1");
        list.add("item2");
        Triangle tri = new Triangle(new Point(0,0), new Point(0,4), new Point(5,0));
        Circle circle = new Circle(new Point(0,0), 5);
        ArrayList<Drawable> draw = new ArrayList<>();
        draw.add(list);
        draw.add(tri);
        draw.add(circle);
        System.out.println("Drawables: Draw");
        for(Drawable d: draw) {
            d.draw();
        }
        System.out.println("");
        System.out.println("Shapes: Draw and area");
        ArrayList<Shape> shapes = new ArrayList<>();
        shapes.add(tri);
        shapes.add(circle);
        for(Shape s: shapes) {
            s.draw();
            System.out.println(s.getArea());
        }
    }

}
```

```
Drawables: Draw
LIST
---
item1
item2
△
○

Shapes: Draw and area
△
10.0
○
78.53981633974483
```

Existing Interfaces

- My favourite interface is Comparable
- Makes objects sortable
- https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
- Usage:

```
public class Person implements Comparable<Person> {
```

- You must fulfill that the class has

```
@Override
public int compareTo(Person other) {
```

  - returns -1 -- if **this Person** should be before **other Person**
  - returns 0 -- if **this Person** is equal to **other Person**
  - returns -1 -- if **this Person** should be after **other Person**

Existing Interfaces

- We can make our running example of a Person class to sort by the String ordering that exists for name
- String already implements Comparable<String>

```
public class Person implements Comparable<Person> {

    private String name;
    private int id;
    public Person(String name, int id) {
        this.name = name;
        this.id = id;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    @Override
    public String toString() {
        return String.format("Person(%s,%s)", name, id);
    }

    @Override
    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }

}
```

Existing Interfaces

- [Person(Bob,99), Person(Carol,1), Person(Alex,33)]
- We use Collections.sort(list)
- [Person(Alex,33), Person(Bob,99), Person(Carol,1)]

```
public static void main(String[] args) {
    Person p1 = new Person("Bob", 99);
    Person p2 = new Person("Carol", 1);
    Person p3 = new Person("Alex", 33);
    ArrayList<Person> list = new ArrayList<>();
    list.add(p1);
    list.add(p2);
    list.add(p3);
    System.out.println(list);
    Collections.sort(list);
    System.out.println(list);
}
```

Existing Interfaces

- Or we can switch it up to sort by name, but if the names are equal than we will sort by id

```
@Override
public int compareTo(Person2 other) {
    int result = this.name.compareTo(other.name);
    if(result != 0) {
        return result;
    }
    if (this.id < other.id) {
        return -1;
    }
    if (this.id > other.id) {
        return 1;
    }
    return 0;
}
```

Existing Interfaces

- [Person(Bob,99), Person(Bob,1), Person(Alex,33)]
- We use Collections.sort(list)
- [Person(Alex,33), Person(Bob,1), Person(Bob,99)]

```
public static void main(String[] args) {
    Person2 p1 = new Person2("Bob", 99);
    Person2 p2 = new Person2("Bob", 1);
    Person2 p3 = new Person2("Alex", 33);
    ArrayList<Person2> list = new ArrayList<>();
    list.add(p1);
    list.add(p2);
    list.add(p3);
    System.out.println(list);
    Collections.sort(list);
    System.out.println(list);
}
```

Existing Interfaces

- However, we can have only **one** natural ordering per class
- But we can make separated sorters (Comparators)
- implements Comparator<Class>
- Has to fulfill
  - int compare(Class c1, Class c2)
  - With same return as previously but note **this** is c1, and parameter is c2

```java
import java.util.Comparator;

public class PersonComparator implements Comparator<Person> {

    @Override
    public int compare(Person p1, Person p2) {
        int result = p1.getName().compareTo(p2.getName());
        if(result != 0) {
            return result;
        }
        if (p1.getId() < p2.getId()) {
            return -1;
        }
        if (p1.getId() > p2.getId()) {
            return 1;
        }
        return 0;
    }

}
```

Existing Interfaces

- However, we can have **one** natural ordering per class
- But we can make separated sorters (Comparators)
- implements Comparator<Class>
- You now need accessors (you can't seen any private class instance members)

```java
import java.util.Comparator;

public class PersonComparator implements Comparator<Person> {

    @Override
    public int compare(Person p1, Person p2) {
        int result = p1.getName().compareTo(p2.getName());
        if(result != 0) {
            return result;
        }
        if (p1.getId() < p2.getId()) {
            return -1;
        }
        if (p1.getId() > p2.getId()) {
            return 1;
        }
        return 0;
    }

}
```

Existing Interfaces

- [Person(Bob,99), Person(Bob,1), Person(Alex,33)]
- Collections.sort(list) -> natural
- [Person(Alex,33), Person(Bob,99), Person(Bob,1)]
- Collections.sort(list, comparator)
- [Person(Alex,33), Person(Bob,1), Person(Bob,99)]

```
public static void main(String[] args) {
    Person p1 = new Person("Bob", 99);
    Person p2 = new Person("Bob", 1);
    Person p3 = new Person("Alex", 33);
    ArrayList<Person> list = new ArrayList<>();
    list.add(p1);
    list.add(p2);
    list.add(p3);
    System.out.println(list);
    Collections.sort(list);
    System.out.println(list);
    Collections.sort(list, new PersonComparator());
    System.out.println(list);
}
```

Interfaces

- Shared functionality – We know that the class must fulfill the (abstract) methods outlined.
- An API lets anyone code that decides it wants to fulfill those guarantees to jump on board (like Comparable) and other code built around this interface becomes available.
- Why is this like an API? (API -> Application Programming Interface)
  - For android google used Oracle's API's (but wrote own code underneath)
  - Copyrightable?
  - https://en.wikipedia.org/wiki/Oracle_America,_Inc._v._Google,_Inc.
- Multiple functionalities – We can guarantee functionality of different interfaces.

Java Collections

- The Java Collections Framework is a collection of interfaces and classes which helps in storing and processing the data efficiently.
- This framework has several useful classes which have tons of useful functions which makes a programmer task super easy.
- Certain behaviour in these Collections 'comes for free' if we fulfill **equals**, **hashCode,** and more-so the interfaces **Comparable/Comparator**
- You've been using one since 1501 ➔ **ArrayList**
  - This is a Collection that fulfills the **List** interface
- We'll also look at classes that fulfill the **Set**, **Queue**, and **Map** interfaces as well

Java Collection Interface

public interface **Collection**<E> extends Iterable<E>

- interface **Iterable**<E> lets us make loops like **for(E object: iterable_E_object){ … }**

1.	boolean add(E e)	add item E to collection (Boolean for success)
2.	boolean remove(Object o)	remove Object o from collection (Boolean for success)
3.	boolean contains(Object o)	is Object o in collection
4.	void clear()	empty the collection
5.	boolean isEmpty()	is collection empty
6.	int size()	how many items in collection

Java Collection Interface

public interface **Collection**<E> extends Iterable<E>

- interface **Iterable**<E> lets us make loops like **for(E object: iterable_E_object){ … }**

1. boolean add(E e) — add item E to collection (Boolean for success)
2. boolean remove(Object o) — remove Object o from collection (Boolean for success)
3. boolean contains(Object o) — is Object o in collection
4. void clear() — empty the collection
5. boolean isEmpty() — is collection empty
6. int size() — how many items in collection

RELY on equals(Object o) to be overridden if want to compare Object's based on contents

Java List Interface

public interface List<E> extends Collection<E>

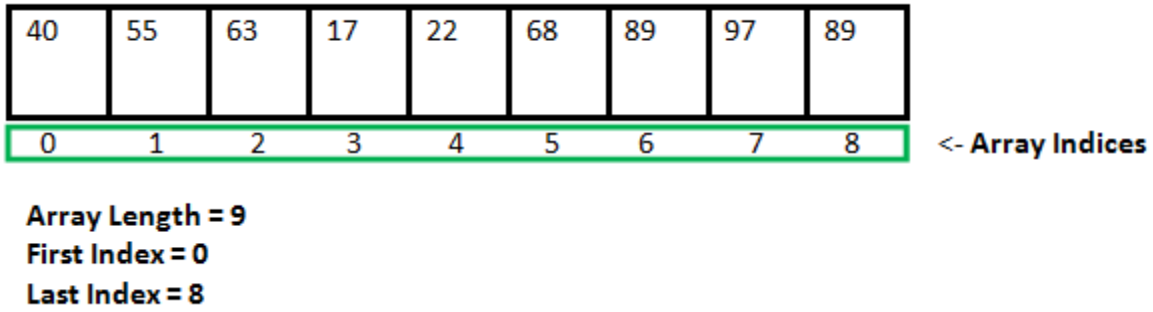
Store things in a sequential list (indexed from 0 to size()-1), no empty locations size()==stored items

1. boolean add(int index, E element) — put item in index and shift everything in way up indices
2. E get(int index) — get item at index
3. E remove(int index) — remove item E at index
4. int indexOf(Object o) — what is index of Object O
5. int lastIndexOf(Object o) — what is last index of Object O

Java ArrayList/LinkedList

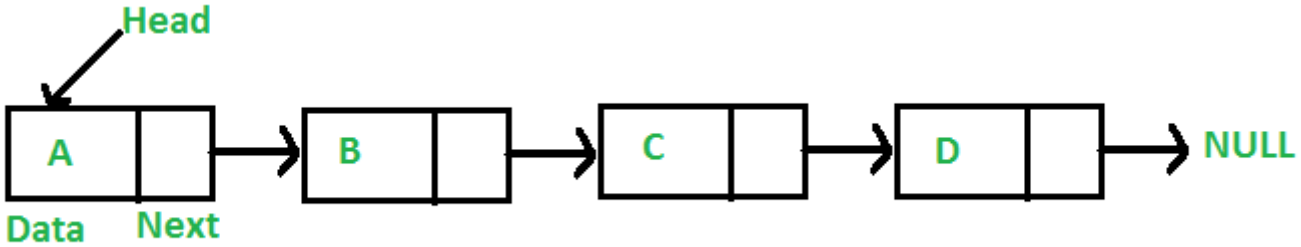
public class ArrayList<E> extends AbstractList<E> implements List<E>

- Uses a hidden Array (E[]) that is managed in the private implementation



public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>

- Uses a object structure of linked Node objects where each stores and E data and points to next Node (Java's is Doubly-Linked ➔ A Node points to next and previous Nodes)



Java Set Interface

public interface Set<E> extends Collection<E>

Store things (without indices), only store one of each item (based on equals(Object o))

Some implementing classes

- HashSet (storage in hidden array E[]),
- TreeSet (storage in hidden tree structure)

The key for this interface is if you attempt to add something that is already in the class implementing Set, that the set is unchanged

Java Map Interface

public interface Map<K,V>

- NOT A COLLECTION, You store **values** of type V, according to some **key** of type K

1.	V put(K key, V Value)	put value of type V at storage spot of key of type K
2.	V get(Object key)	get value of type V stored at given key to type K
3.	V remove(Object key)	remove Object o from map (V value removed)
4.	void clear()	empty the map
5.	boolean isEmpty()	is map empty
6.	int size()	how many items in collection
7.	Set<K> keyset()	set of keys of type K
8.	Collection<V> values()	collection of V values stored in map

Java Map Interface

- A Map makes use of equals(Object o) to determine if a key K has been used in the map
- Often a map will use the key's hashCode() first, and only if two hashCodes are equal will it then double-check deeper by comparing using equals()
- Maps work like Python's dictionaries (but with a lot more syntax than python requires)
- Common maps are TreeMap, and HashMap

Onward to … JavaFX

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY