Abstract classes, Interfaces & Comparators

- See also Java Precisely, section “Classes” and “Interfaces”
- Read more about Java interfaces here:
  `http://docs.oracle.com/javase/tutorial/java/concepts/interface.html`

This lecture

- Abstract classes and Interfaces
  - Difference between **extends** and **implements**
  - Multiple inheritance
- Comparator and Comparable
  - Comparing objects
  - How to use these two interfaces.

What is an abstract class?

```
public abstract class AbstractPoint  {
    int x, y;
    public AbstractPoint () { }; //ok but should not have c
    public abstract void setPoint(int a, int b);
    public int getX () {return x;}
    public int getY() { return y;}
}
// Using the class
AbstractPoint p; //OK! Can declare a variable
p = new AbstractPoint(); //Compile-time error
```

- An abstract class should have at least one abstract method.
- It is better not to have a constructor in an abstract class.

Abstract class - Inheritance

```
public class MyPoint extends AbstractPoint {
    public MyPoint () { setPoint(0,0);} //Better design
    public MyPoint (int a, int b) {setPoint(a,b);}
    public void setPoint (int a, int b){
          x=a; y=b;
    }
    public double size() {
        return 0.0;
    }
}

MyPoint p = new MyPoint(); //Now it is OK
```

- Abstract class need be extended to create objects.
- Abstract classes do not allow for **multiple inheritance**, i.e., extending more than one (abstract) class.
- How do we get multiple inheritance in Java?

Interface - What is a Java interface?

```
public interface Point {
    void setPoint(int a, int b);
    int getX();
    int getY();
}
```

- An interface contains only methods signatures and/or constants.
- An interface needs be implemented. How?

Interface – Implementation

```
public abstract class AbstractPoint implements Point {
    protected int x, y;
    public void setPoint(int a, int b);
    public int getX () {return x;}
    public int getY() { return y;}
}
```

- An interface supplies a specification of methods and requires another class to implement those methods.
- An interface can be implemented by a class or an abstract class.
- In Java, **multiple inheritance** is achieved using interfaces.

Interface – multiple inheritance

```
public class CompPoint implements Point, Comparable {
    int x, y;
    CompPoint (int a, int b) { setPoint(a, b);} //constr
    public void setPoint (int a, int b) {x=a; y=b;}
    public int getX() { return x;}
    public int getY() { return y;}
    // this method belongs to the interface Comparable
    public int compareTo(Object o) {
            CompPoint obj = (CompPoint)o;
            if (x != obj.x) {
                return x < obj.x ? -1 : 1;
            }
            return y < obj.y ? -1 : (y == obj.y ? 0 : 1);
    } }
```

- Comparable is a Java interface from java.lang.*
- The class CompPoint inherits methods from Point and Comparable.

Another way: extend AbstractPoint and implement Comparable

```
public class CompPoint2 extends AbstractPoint
                        implements  Comparable {
  CompPoint2 (int a, int b) { //constructor
      setPoint(a, b);
   }
  public void setPoint (int a, int b) {x=a; y=b;}
  // this method belongs to the interface Comparable
  public int compareTo(Object o) {
        CompPoint2 obj = (CompPoint2)o;
        if (x != obj.x) {
            return x < obj.x ? -1 : 1;
        }
        return y < obj.y ? -1 : (y == obj.y ? 0 : 1);
    }
}
```

Quick Quiz 1

Which of the following is correct?

1. An interface can extend one or more interfaces
2. An interface can extend a single class
3. A class can extend more than 2 abstract classes.
4. An abstract class cannot have method implementations with bodies.

What is the point of interfaces?

Compare this

```
// version 1
class Student { ... } // has a getName() method
class Teacher { ... } // has a getName() method
Object[] arr = [...]; // all students and teachers
for(int i=0; i<arr.length; i++) {
  print(arr[i].getName()); } // error!
```

with

```
// version 2
interface Namable { String getName(); }
class Student implements Namable { ... }
class Teacher implements Namable { ... }
Namable[] arr = [...]; // all students and teachers
for(int i=0; i<arr.length; i++) {
  print(arr[i].getName()); } // OK!
```

The comparator interface, comparable

- Look at the Java API, `Comparator` and `Comparable`

Comparator: `int compare(T o1, T o2)`

Comparable: `int compareTo(T o)`

The int that is returned means:

> *(Comparator) Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.*
>
> *(Comparable) Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.*

Difference between comparator and comparable

What is the difference between Comparator and Comparable?

If c is a **comparator**, you compare two objects by typing `c.compare(o1, o2)`. If a class (such as String) supports the compareTo method, you can type e.g. `s1.compareTo(s2)` to compare the strings s1 and s2.

With `c.compare(o1, o2)` there are 3 objects: c, o1, o2

With `s1.compareTo(s2)` there are 2 objects: s1 and s2

In the String class, compareTo compares strings in dictionary order. YOU CAN NOT CHANGE THIS! With comparators, the comparator c can be whatever you want: you can compare strings in dictionary order, or reverse dictionary order, or whatever.

Comparators are much more flexible!

Comparator and Comparable – Example

```
public class MyCompPoint extends MyPoint implements
                                    Comparable, Comparator {
   public int compareTo(Object o) {
     MyCompPoint obj = (MyCompPoint)o;
     if (x != obj.x) {
        return x < obj.x ? -1 : 1;
      }
      return y < obj.y ? -1 : (y == obj.y ? 0 : 1);
    }
   public int compare(Object o1, Object o2) {
      MyCompPoint obj1 = (MyCompPoint)o1;
      MyCompPoint obj2 = (MyCompPoint)o2;
      if (obj1.x != obj2.x) {
         return obj1.x-obj2.x;
       }
      return obj1.y-obj2.y;
   }
```

Demo - Application to Sorting

```
MyCompPoint p1 = new MyCompPoint(2, 3);
MyCompPoint p2 = new MyCompPoint(2, 0);
MyCompPoint p3 = new MyCompPoint(0,2);
if p1.compareTo(p2) System.out.println("P1 before P2");
else System.out.println("P1 after P2");
List<MyCompPoint> lp = new ArrayList<MyCompPoint>();
lp.add(p1); lp.add(p2); lp.add(p3);
Collections.sort(lp); //will use compareTo
MyCompPoint c = new MyCompPoint();
Collections.sort(lp, c);//will use compare
```

- See the API documentation for Collections.sort()
- `http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html` - ordering tutorial
- Play around with the provided Java code; more on sorting with Tom

Comparator - A local use

```
public class MyCompPoint2 extends MyPoint {
      // implement a comparator within the class
      Comparator c = new Comparator() {
          public int compare(Object o1, Object o2) {
           // code for this method goes here
          }
       };
    // using the comparator
    public void mySort(List l) {
       Collections.sort(l, c);
    }
}
```

- Again `Comparator` is much more flexible! You may also write a class `MyPointComparator` that implements the interface Comparator for objects of type MyPoint. **See an example from the provided Java code**.

Quick Quiz 2

Which of the following is NOT correct?

1. A class can implement more than 2 interfaces
2. Comparator is a class while Comparable is an interface.
3. An interface can have public static final attributes.
4. Comparable provides a natural ordering of the objects of a given class.
5. An abstract class can have method signatures without bodies.

Solutions to the Quizzes thus far

- L1Q1: 3
- L1Q2: 3
- L1Q3: 4
- L2Q1: 4
- L2Q2: 3
- L3Q1: 1
- L3Q2: 2

NB: L1Q1 means Quiz 1 in Lecture 1, etc.

Some exercises on comparators

Task 1: Comparator to compare strings ignoring their first character (you can use `lexcompare(s1,s2)`, which returns true if s1 is less than s2 in dictionary order)

Task 2: Comparator to compare names (first last) based on the dictionary order of the last name (you need to discard the first word and any spaces following the first word);

```
Comparator c = new Comparator() {
        public int compare(Object o1, Object o2) {
            String s1 = ((String)o1);
            String s2 = ((String)o2);
      ...
        }
    };
```

Solutions, task 1

```
/*
Task 1: Comparator to compare strings ignoring their first
(you can use lexcompare(s1,s2), which returns true if s1 is
in dictionary order)
*/
Comparator c = new Comparator() {
        public int compare(Object o1, Object o2) {
            String s1 = ((String)o1);
            String s2 = ((String)o2);
            // FIXME are s1 and s2 non-empty?
            s1=s1.substring(1);
            s2=s2.substring(1);
            if(lexcompare(s1,s2)) return -1;
            else if(s1.equals(s2)) return 0;
            else return 1;
        }
    };
```

Solutions, task 2

```
/*
Task 2: Comparator to compare names (first last) based on the
dictionary order of the last name (you need to discard the first word
and any spaces following the first word);
*/
String discard_spaces(String s) {
while(true) {
    if(s.equals("")) return s;
    if(!s.charAt(0)==' ') return s;
    s=s.substring(1);
}
}

String discard(String s) {
while(true) {
    if(s.equals("")) return ""; // FIXME what to do here?
    if(s.charAt(0) == ' ') return discard_spaces(s);
    s=s.substring(1);
}
}

Comparator c = new Comparator() {
    public int compare(Object o1, Object o2) {
        String s1 = ((String)o1);
        String s2 = ((String)o2);
        s1=discard(s1);
        s2=discard(s2);
        if(lexcompare(s1,s2)) return -1;
        else if(s1.equals(s2)) return 0;
        else return 1;
    }
};
```