Interfaces, Lambda Expressions, and Inner Classes

Yangtao Ge

June 24, 2019

1 Interfaces

1.1 The Interface Concept

Definition: ***Interface*** is not a class but a set of *requirements* for the classes that we want to conform to the interface.
e.g.

```
public interface Comparable<T>{
    int compareTo(T other); // para has type T
}
```

Some Notice to Interface:

- all methods of an interface are automatically *public* – we don't add *public* in the signiture
- interfaces can define constants
- interfaces cannot have instance fields
- method are never implemented in interface (we could now, but it's **BAD**)

To make a class implement an interface:

1. declare that your class intends to *implement* the given interface
2. supply definitions for <u>all</u> methods in the interface

e.g.

```
class Employee implements Comparable<Employee>{
    public int compareTo(Employee other){
        return Double.compare(salary, other.salary);
    }
    ...
}
```

N.B. Try to use *Generic type*, less using 'type cast'.

Regarding to *compareTo* method:

- how to compare:
  - substraction: if we know the maximum bounday is less than 'maximum of Integer'.
  - compareTo: don't care
- inheritance Problems: (solve like *equal()* method in Chapter 5)
  - different notations of comparison: add a same class test

```
if(getClass() != other.getClass){
    throw new ClassCastException;
}
```

  - common algorithm: provide a single *compareTo* method, and declare it as *final*

1.2 Properties of Interfaces

Some properties of Interface:

- Interfaces are not classes – can't do '*x = new Comparable(...);*'
- we can declare <u>interface variables</u>

  - can do '*Comparable x;*'
  - can do '*x = new Employee(...);*' (since 'Employee' implements Comparable)

- we can check whether an object implements an interface by '*instanceof*' keyword

- we can extend Interfaces

```
public interface Moveable{
    void move (double x, double y);
}
public interface Powered extends Moveable{
    double milesPerGallon();
}
```

- we can add *constants* in the interface. This mathod is automatically '**public static final**'

```
public interface Powered extends Moveable{
    double milesPerGallon();
    // public static final constant
    double SPEED_LIMIT = 95;
}
```

- classes can implement *multiple* interfaces – we can do '*class Employee implements Person, Comparable*' (but one class can only have one superclass)

1.3 Interfaces and Abstract Classes

Key: A class can only extend a single class, but can implement several interfaces.

We can think it as:

- abstract classes: tends to stress what it is (inheritance – 'is-a relationship')
- interface: tends to illustrate what can it do (properties).

N.B. Remember it by – **things can only belong to one class, but it can have several properties.**

1.4 Static and Private Methods

We can add '*static* method' since Java 8, and '*private* method' since Java 9. This is not very useful.

Ref: p.306

1.5 Default Methods

some useful situation for *default* modifier:

- implement 'iterator': providing an exception

```
public interface Iterator<E>{
    boolean hasNext();
    E next();
    default void remove(){
        throw new UnsupportedOperationException(``remove'')
    }
}
```

- implement 'collection': call other methods

```
public interface Collection{
    int size(); // an abstract method
    default boolean isEmpty(){
        return size() == 0;
    }
}
```

- *interface evolution* for adding class in the future.

1.6 Resolving Default Method Conflicts

Two basic rules for resolving default method conflicts:

- **Class win rule**: if a superclass provides a conrete method, default methods with the same name and parameter types are ignored

```
class Student extends Person implements Named{
    ... // use getName in Person class only
}
```

- **Interfaces clash rule**: if an interface provide default method, another interface contains a method with the same name and parameter (types default or not), then you must resolve the conflict by overriding the method

```
class Student implements Person, Named{
    public String getName(){
        //choose to use getName in Person
        return Person.super.getName();
    }
}
```

1.7 Interfaces and Callbacks

Definition: *callback* pattern means when you specify the action that should happen whenever a particular event happens. e.g. *ActionListener* in java swing

Usually, we will predefine how a method work of a method, then call it whenever we want it.

Ref: pp.310 - 312 & COMP 0004 Java Coursework Part2

1.8 The *Comparator* Interface

What is is like:

```
public interface comparator<T>{
    int compare(T first, T second);
}
```

Using user-defined comparator:

```
public class LengthComparator implements Comparator<String>{
    public int compare(String first, String second){
        return first.length() - second.length();
    }
}

String friends = {``Peter'', ``Paul'', ``Mary''};
Array.sort(friends, new LengthComparator());
```

Ref: pp.323 - 314 & p.322

1.9 Object Cloning

Difference between 'copy' and 'clone':

- copy (=): make a copy of variable holding an object reference – change to either variable also affects the other
  i.e. original → Employee ← copy
- clone (clone()): identical to original but whose state can diverge over time
  i.e. original → Employee1; cloned → Employee2

clone method is '*protected*' so that it can only clone itself. i.e. Employee's clone can clone Employee only

Two types of Clone:

- **Shallow copy**: (default) just copy the object only, don't care what are inside it. – only valid for immutable objects

```
class Employee implements Cloneable{
    //public access, changfe return type
    public Employee clone() throws CloneNotSupportedException{
        return (Employee) super.clone();
    }
 }
```

- **Deep copy**: (redefined) clone the instance fields piece by piece – for mutable objects

```
class Employee implements Cloneable{
    ...
    public Employee clone() throws CloneNotSupportedException{
        //call Object.clone
        Employee cloned = (Employee)super.clone
        //clone mutable fields
        clone.hireDay = (Date)hireDay.clone
    }
}
```

i.e. *super* means Cloneable interface

Condition to use clone:

- it can be cloned:
  - default clone is good enough (shallow clone)
  - default clone can be patched up by calling *clone* on mutable sub-objects
- it cannot be cloned

When it is cloneable:

1. implement the *Cloneable* interface
2. redefine the *clone* method with the *public* accessor

2 Lambda Expression