Interfaces, Lambda Expressions, and Inner Classes

Yangtao Ge

June 26, 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

2.1 Why Lambdas?

Definition: *Lambda expression* is a block of code that you can pass around so it can be executed **later, once or multiple times**

Java will not pass the actual code block around. Lambda expression provides a tag (API) to access them.

2.2 The Syntax of Lambda Expression

Three types of lambda expression:

- simple return:

```
(String first, String second)
    -> first.length()-second.length()
```

- code block:

```
(String first, String second) -> {
    if (first.length() < second.length()) return -1;
    else if (first.length() < second.length()) return -1;
    else return 0;
}
```

- no parameters:

```
() -> {
    for (int i = 100; i>= 0; i--) System.out.println(i);
}
```

i.e. We never specify the '*Return type*', it is inferred from the context.

2.3 Function Interface

Definition: *functional interface* is an **interface** that we can supply a lambda expression whenever an object of an interface with a single abstract method is expected. (i.e. it is implement an interface by 'lambda expression')

Some applications of functional interface:

- **Predicate**:

```
public interface Predicate<T>{
    boolean test(T t);
}
```

The 'ArrayList' class has a *removeIf* method whose para is a 'Predicate'

```
list.removeIf(e -> e == null)
```

- **Supplier**: used for lazy evalutation

  ```
  public interface Supplier<T>{
      T get();
  }
  ```

  The 'requireNonNullOrElseGet' method only calls the supplier when value is needed:

  ```
  LocalDate hireDay = Object.requireNonNullOrElseGet(day,
      () -> new LocalDate(1970, 1, 1))
  ```

2.4 Method Reference

Def: *method reference* directs the compiler to produce an instance of a functional interface , overriding the single abstract method of the interface to call the given method.
e.g.

```
var timer = new timer(1000, event -> System.out.println(event));
```

is the same as:

```
var timer = new timer(1000, System.out::println);
```

here, ***System.out::println*** is a 'method reference'

Three variants of Method reference (target::methodName):

- *object::instanceMethod*: e.g. *System.out::println*

  ```
  x -> System.out.println(x)
  ```

- *Class::instanceMethod* + 2 variants: e.g *String::compareToIgnoreCase*

  ```
  (x, y) -> x.compareToIgnoreCase(y);
  x -> this.equals(x);
  ```

  - *this::instanceMethod*

- *super::instanceMethod*

- *Class::staticMethod*: e.g. *Math::pow*

```
(x, y) -> Math.pow(x, y)
```

2.5 Constructor Reference

Almost the same as 'method reference', except name of the method is *new* e.g.

```
Person[] people = stream.toArray(Person[]::new)
```

i.e. *int[]::new* is the same as '*x -> new int[x]*'

2.6 Variable Scope

How to construct a lambda expression (three ingredients):

1. A block of code
2. parameters
3. Value for the '*free*' variables (i.e. val which is not parameter and not defined inside code)

e.g.

```
public static void repeatMessage (String text, int delay){
    ActionListener listener = event -> {
        System.out.println(text);
        Toolkit.getDefaultToolKit().beep();
    };
    new Timer(delay, listener).start();
}

repeatMessage(``Hello'', 1000);
```

N.B. Here,

- 'text'is a **free** variable.
- 'Hello' is a **captured** variable.

Some Notice of the scope in lambda expression:

- we can't mutate captured variable (increment)
- we can't refer to a variable that is mutated outside (for loop)
- captured var must be 'effectively final'
  (i.e. it means never assigning a new value after it has been initialized)
- it has the same scope as a nested block – can't have a para in lambda which has the same name as a local val

2.7 Processing Lambda Expression

The point of using lambdas is **deferred execution** (i.e. execute it later)

Some cases for executing code later:

- Running code in separate thread
- Running code multiple times
- Running code at right point in an algorithm
- Running code when something happen
- Running code when necessary

To accept the lambda, we need to pick a **functional interface** e.g.:

```
public interface IntConsumer{
    // this is a functional interface
    void accept(int val);
}

public static void repeat(int n, IntConsumer action){
    // this is the method to accept lambdas
```

```
    for (int i = 0; i < n; i++){
        action.accept(i);
    }
}

// this is how to call it
repeat(10, i -> System.out.println(``CountDown: '' + (9 - i)))
```

i.e. *Ref:*

- **Function Interface** are on *p.337*
- **Function Interface** for primitive types are on *p.338*

2.8 More about Comparators

Some common method in Comparator:

- plain sort an array:

```
Array.sort(people,
    Comparing(Person::getName));
```

- two-level sort an array:

```
Array.sort(people,
    Comparator.comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

- user-defined sort:

```
Array.sort(people,
    Comparator.comparing(Person::getName,
    (s,t) -> Integer.compare(s.length(), t.length())))
```

- including *null* in Key function:

```
Array.sort(people,
    comparing(Person::getMiddleName,
        nullsFirst(natureOrder())))
```

Ref: pp.339 - 340 & Java doc 'Comparator'

3 Inner Class

Two reasons for construct an inner class:

- hidden from other classes in the same package
- access the data from the scope where they are defined (its mother class)

3.1 Use of an Inner Class to Access Object State

An example of inner class:

```
public class TalkingClock{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep){...}
    public void start(){...}

    public class TimePrinter implements ActionListener{
        //this is an inner class
    }
}
```

Some Notices to this example:

- **NOT** every 'TalkingClock' has a 'TimePrinter' instance field.
- inner class('TimePrinter') has access to:
  - its own data fields
  - those of the outer object creating it
- Inner class can be **private**, normal class can only be public

3.2 Special Syntax Rules for inner classes

Some new notations:

- Outer class reference: "*OuterClass.**this***"
- Inner object Constructor: "*outerObject.**new** InnerClass (constr para)*"
- Inner class reference: "*OuterClass.InnerClass*"

3.3 Are Inner Classes Useful, Necessary, or Secure?

Important Fact: inner classes are a phenomenon of the '*compiler*', not the '*virtual machine*'

Explaination: If we define a regular class the same as the inner class, we will get the class with exact same functionalities but with some *tags*
e.g.

```
public class innerClass.TalkingClock$TimePrinter
    implements java.awt.event.ActionListener{
        final innerClass.TalkingClock this$0;
        public innerClass.TalkingClock$TimePrinter
            (innerClass.TalkingClock);
        public void actionPerformed
            (java.awt.even.ActionListener)
    }
```

'*this$0*' is produced by the compiler and run that in the virtual machine

Necessity:
Inner class is powerful than regular class because they have more **access privileges**.

Security:
When it is a public accessiable inner class, it is possible to hack the class by adding class into the same package

3.4 Local Inner Class

Definition: Class defined inside local Methods

Their accssors are never declared, and their scope is always restricted to the block where they are declared

3.5 Accessing Variables from Outer Methods

Local inner classes are powerful than other normal inner class: **they can access local variables**

Noticed that: those local varibales must be ***effectively*** *final*

Ref: For details, pp.350 - 351

3.6 Anonymous Inner Classes

Definition: make only a single objecgt of this class, and don't give the class a name. This is called '*Anonymous Inner Classes*'
e.g.

```
public void start(int interval, boolean beep){
    var listener = new ActionListener(){
        public void actionPerformed(ActionEvent event){
            ...
        }
    };

    var timer = new timer(interval, listener);
    timer.start();
}
```

General Syntax is:
new *SuperType(construction parameters)*{*inner class method and data*}

Some notice of Anonymous inner classes:

- *superType* can be an interface as well as a class
- no constructors (class has no name)
- used for event listeners and other call backs, but better to use lambdas

3.7 Static Inner Classes

When to use: want to hide one class inside another, don't want the inner class to have reference to the outer class object.
e.g. Self Defined data structure:

```
class ArrayAlg{
    // self-defined data structure
    public static class pair{
        private double first;
        private double second;

        public Pair(double first, double second){
            this.first = first;
            this.second = second;
        }
        // getter method
        ...
    }

    // static method using it
    public static Pair minmax(double[] values){
        ...
        return new Pair(min, max)
    }
}
```

some Notices of Static inner class:

- only inner class can be declared as *static*
- the example requires *static* inner classs(Pair) because the inner class object is constructed inside a *static* method (minmax)
- static inner classes can have static fields and Methods
- inner classes inside an interface are automatically *static* and *public*

4 Service Loaders

Materials for the second reading

5 Proxies

Materials for the second reading