ITI 1121. Introduction to Computing II

Interface: abstract data types (ADT) and their implementations

by

Marcel Turcotte

Version January 19, 2020

Preamble

Preamble

Overview

Overview

Interface: abstract data types (ADT) and their implementations

Class declarations are one of Java's mechanisms to create new data types. In this case, we say that it is a concrete data type. In this module, we discuss the concept of interface that will allow the definition of abstract types of data.

General objective:

- This week, you will be able to declare an abstract data type through an interface.

Preamble

Learning objectives

Learning objectives

- **Explain** in your own words the concept of interface.
- **Declare** an interface
- **Implement** an interface

Lectures:

- Pages 2–7 of E. Koffman and P. Wolfgang.

Preamble

Plan

Plan

Summary

Defining a new type

- A **class declaration** defines a new data type

Defining a new type

- The following declaration defines a new type

```
public class Point {

}
```

Placed in a file named **Point.java**, this class declaration can be compiled.

```
> javac Point.java
```

Defining a new type

- We can declare a variable of type **Point**.

```
public class Test {
    public static void main(String[] args) {
        Point p;
    }
}
```

Defining a new type

- We can create an object of the class **Point**.

```
public class Test {
    public static void main(String[] args) {
        Point p;
        p = new Point();
    }
}
```

- It can be done because there exists a **default constructor** .
- Of course, these objects have no variables or methods, for now.

```
public class Point {
    private double x;
    private double y;
    public Point(double xInit, double yInit) {
        x = xInit;
        y = yInit;
    }
    public double getX() {
        return x;
    }
    // ...
    public void translate(double deltaX, double deltaY) {
        x = x + deltaX;
        y = y + deltaY;
    }
}
```

- The course Web site has a **complete implementation**.

Using this new type type

```
public class Test {
    public static void main(String[] args) {

        Point p1, p2;

        p1 = new Point(10, 20);
        p2 = new Point(522, 43);

        if (p1.getX() < p2.getX()) {
            p1.translate(p2.getX() - p1.getX(), 0.0);
        }

        if (p1.getY() < p2.getY()) {
            p1.translate(0.0, p2.getY() - p1.getY());
        }

    }
}
```

Definition: concrete data type

- A class declaration defines a **concrete data type**.
- We say **concrete** because the data representation and the methods are present.

Interface

Introduction

- Let's revisit the two implementations of the class **Pair** introduced in the last lecture.

```java
public class PairVar {

    private int first;
    private int second;

    public PairVar(int firstInit, int secondInit) {
        first = firstInit;
        second = secondInit;
    }
    public int getFirst() {
        return first;
    }
    public int getSecond() {
        return second;
    }
    public void setFirst(int value) {
        first = value;
    }
    public void setSecond(int value) {
        second = value;
    }
}
```

```
public class PairArray {

    private int[] elems;

    public PairArray(int first, int second) {
        elems = new int[2];
        elems[0] = first;
        elems[1] = second;
    }
    public int getFirst() {
        return elems[0];
    }
    public int getSecond() {
        return elems[1];
    }
    public void setFirst(int value) {
        elems[0] = value;
    }
    public void setSecond(int value) {
        elems[1]= value;
    }
}
```

Introduction

- **PairVar** and **PairArray** are **two implementations of the same concept**, a pair of integer values.
- Java provides us with a mechanism to formalize this idea.

Interface Pair

```
public interface Pair {

    public abstract int getFirst();

    public abstract int getSecond();

    public abstract void setFirst(int first);

    public abstract void setSecond(int first);

}
```

Interface Pair

- The declaration of an interface is similar to that of a class.
- It starts with the keyword **interface**, rather than **class**, followed by the name of an identifier (the name of the interface), followed by the body of the interface.

```
public interface Pair {
    public abstract int getFirst();
    public abstract int getSecond();
    public abstract void setFirst(int first);
    public abstract void setSecond(int first);
}
```

This declaration is saved into a file that has the same name as the interface, with extension .java. When compiled, it produces a **.class** file.

Interface

An **interface** contains:

- Constants;
- Abstract methods.

Interface

- An **abstract** method has no implementation.
- Since all the methods of the interface must be abstract and public, **public abstract** is implied and can be omitted.

```
public interface Pair {
    int getFirst();
    int getSecond();
    void setFirst(int first);
    void setSecond(int first);
}
```

The **interface** is a contract. A list of methods to be implemented.

Abstract data type

- We have just defined an **abstract data type (ADT)**. We specified the **operations**, but not the **implementation**!

```
public interface Pair {
    int getFirst();
    int getSecond();
    void setFirst(int first);
    void setSecond(int first);
}
```

Abstract data type

- **What's** the point?

We can declare a variable of type **Pair**!

```
Pair p;
```

The operations **p.getFirst()**, **p.getSecond()**, **p.setFirst(10)**, and **p.setSecond(55)** are all valid, from the point of view of the type system. 1

Abstract data type

- But what do you put in the reference variable **p**?

```
Pair p;
```

Java: implements

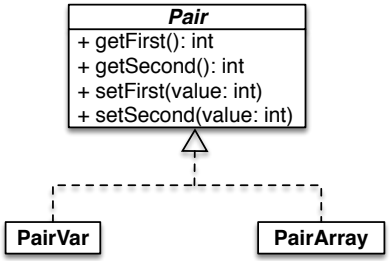
- Here is a new keyword, **implements**. We say that the class **PairVar implements** the interface **Pair**.

```
public class PairVar implements Pair {

    private int first;
    private int second;

    public PairVar(int firstInit, int secondInit) {
        first = firstInit;
        second = secondInit;
    }
    public int getFirst() {
        return first;
    }
    // ...
}
```

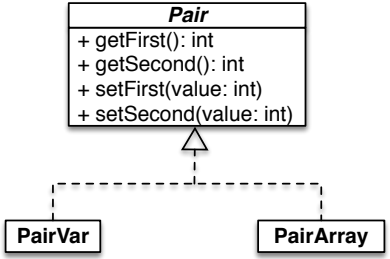
```java
public class PairVar implements Pair {

    private int first;
    private int second;

    public PairVar(int firstInit, int secondInit) {
        first = firstInit;
        second = secondInit;
    }
    public int getFirst() {
        return first;
    }
    public int getSecond() {
        return second;
    }
    public void setFirst(int value) {
        first = value;
    }
    public void setSecond(int value) {
        second = value;
    }
}
```

```java
public class PairArray implements Pair {

    private int[] elems;

    public PairArray(int first, int second) {
        elems = new int[2];
        elems[0] = first;
        elems[1] = second;
    }
    public int getFirst() {
        return elems[0];
    }
    public int getSecond() {
        return elems[1];
    }
    public void setFirst(int value) {
        elems[0] = value;
    }
    public void setSecond(int value) {
        elems[1]= value;
    }
}
```

UML: Pair


Pair
+ getFirst(): int
+ getSecond(): int
+ setFirst(value: int)
+ setSecond(value: int)
PairVar
PairArray

What's the point?




```
Pair p;

p = new PairVar(10, 20);

p = new PairArray(10, 20);
```

new Pair() does not make any sense.

Comparable

Complete example

We would like to write a **sort** method.

- Compare how you would write a sort method for objects of the class **Person** and a sort method for objects of the class **Time**.
- What are the **resemblances** and **differences**?

Sort algorithm

- The **sort algorithm** is the same, regardless of the elements of the array.
- What changes is the way you **compare the elements**!

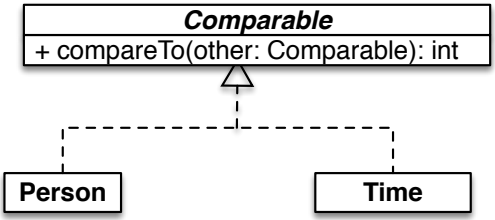
Comparable

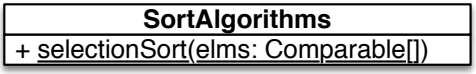
Let's define the interface **Comparable**:

```
public interface Comparable {
    public int compareTo(Comparable other);
}
```

An object is "**Comparable**" if it has a **compareTo** method!

UML: Comparable





SortAlgorithms.selectionSort

```
public class SortAlgorithms {
    public static void selectionSort(Comparable[] a) {
        for ( int i = 0; i < a.length; i++ ) {
            int min = i;
            // find the smallest element in the unsorted
            // portion of the array
            for ( int j = i+1; j < a.length; j++ )
                if ( a[j].compareTo( a[ min ] ) < 0 )
                    min = j;
            // exchange that element and that of i
            Comparable tmp = a[ min ];
            a[min] = a[i];
            a[i] = tmp;
        }
    }
}
```

Time

```
public class Time implements Comparable {

    private int timeInSeconds;

    public int compareTo(Comparable obj) {

        Time other = (Time) obj;
        int result;
        if (timeInseconds < other.timeInSeconds) {
            result = -1;
        } else if (timeInSeconds == other.timeInSeconds) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

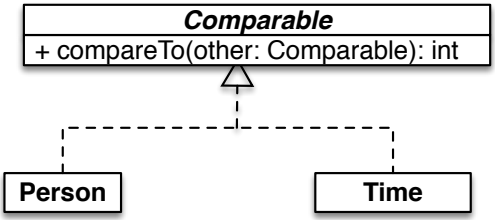
Person

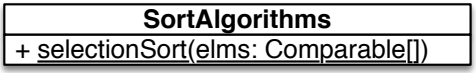
```
public class Person implements Comparable {

    private int id;
    private String name;

    public int compareTo(Comparable obj) {
        Person other = (Person) obj;
        int result;
        if (id < other.id) {
            result = -1;
        } else if (id == other.id) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

Person (2)

```
public class Person implements Comparable {

    private int id;
    private String name;

    public int compareTo(Comparable obj) {

        Person other = (Person) obj;

        return name.compareTo(other.name);

    }
}
```

UML: Comparable





Call-back function

Problem: uoAlert

- **Problem:** The University of Ottawa wishes to replace the the computer system it uses for the **management of warning messages**.
  - The system must transmit the alert messages to different clients, including a **mobile phone application** and a **Web application**.

uoAlert: AlertListener

- As a software developer, you have the idea of creating the interface **AlertListener**.
  - Any client who wishes to receive alert messages must implement the interface **AlertListener**.
  - The server has a method **register** whose parameter is of type **AlertListener**.
  - Any customer who wishes to receive alert messages must register with the server.

```
public interface AlertListener {
    void processAlert(String message);
}
```

uoAlert: PhoneApp

- The application **PhoneApp** implements the interface **AlertListener**.

```
public class PhoneApp implements AlertListener {

    public void processAlert(String message) {
        System.out.println("PhoneApp: " + message);
    }

    // The other methods of PhoneApp would be here!

}
```

uoAlert: WebApp

- The application **WebApp** implements the interface **AlertListener**.

```
public class WebApp implements AlertListener {

    public void processAlert(String message) {
        System.out.println("WebApp: " + message);
    }

    // The other methods of WebApp would be here!

}
```

uoAlert: AlertServer

```
public class AlertServer {

    private AlertListener[] clients;
    private int numberOfClients;

    public AlertServer(int capacity) {
        clients = new AlertListener[capacity];
        numberOfClients = 0;
    }

    public void register(AlertListener client) {
        clients[numberOfClients] = client;
        numberOfClients++;
    }

    public void broadcast(String message) {
        for (int i=0; i<numberOfClients; i++) {
            clients[i].processAlert(message);
        }
    }
}
```

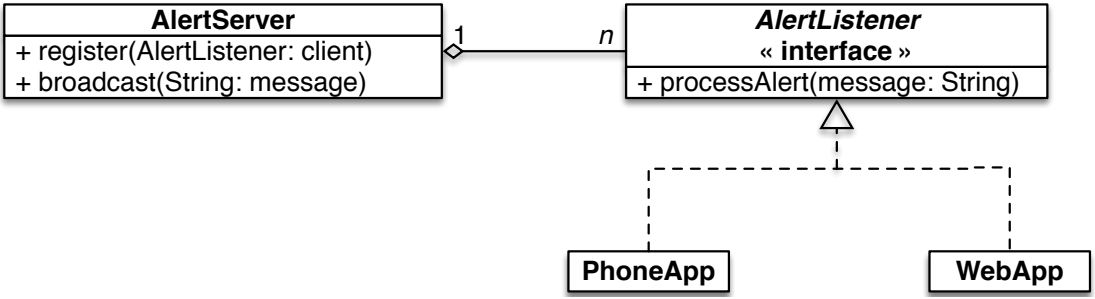
uoAlert: Run

```
public class Run {
    public static void main(String[] args) {
        AlertServer server;
        server = new AlertServer(2);

        PhoneApp phone;
        phone = new PhoneApp();

        WebApp web;
        web = new WebApp();

        server.register(phone);
        server.register(web);

        server.broadcast("Test!");
    }

}
```

uoAlert: Run

```
> java Run
PhoneApp: Test!
WebApp: Test!
```

uoAlert: UML


AlertServer
+ register(AlertListener: client)
+ broadcast(String: message)
1
n
AlertListener
« interface »
+ processAlert(message: String)
PhoneApp
WebApp

Call-back

- The method **processAlert** is a **call-back function**.
- A client registers with the server (service).
  - Only the clients having a method called **processAlert** can register with the server.
- Later, when an alert is generated, the server informs the clients.
  (calls their method **processAlert**)

Prologue

Summary

- An **interface** is a formalism to create an **abstract data type**.
- The declaration of an **interface** is similar to that of a class.
  - However, an interface contains only abstract methods.
- The **keyword** implements is used to indicate the fact that a class implements all the methods of a give interface.

Next module

- **Inheritance**

References I


E. B. Koffman and Wolfgang P. A. T.
Data Structures: Abstraction and Design Using Java.
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa