Interfaces

Software Engineering
CITS1220
Lecture Outline

- Interfaces
- Abstract Classes
- Java Examples

Reference reading for this lecture is

- Chapter 10, focus on Section 10.6 Interfaces
- Barnes and Kolling, Objects First with Java
Software Reuse

A main goal of OOP is reuse of software and the concept of software components.

Techniques for software reuse include:
- Cut-n-paste (not recommended!)
- Aggregation (see lecture)
- Inheritance (see lecture)
- Interfaces
Reuse of specification

- *Aggregation* is based on reuse of *implementation* - using existing code
- *Interfaces* are based on the reuse of *specification*, but not implementation
- *Inheritance combines* the reuse of implementation and specification
  - So far we have only seen implementation
Design patterns

- Reuse *design* as well as implementation and specification
- Not covered in this unit, but very important in practical OOP
  - See the book *Design Patterns* by Gamma, Helm, Johnson, Vlissides (the Gang of Four)
Interfaces

- An interface is “pure specification” - in other words a list of method signatures
  - Method signature just specifies the method’s return type, name and parameters
- A class that supplies an actual implementation for all of the methods is said to implement the interface
Using interfaces

- To use a pre-existing interface is easy
  - Look up the required methods
  - Declare that your class implements the interface
  - Implement them
The class libraries define this interface as consisting of the single method

```java
public void actionPerformed(ActionEvent e)
```

Essentially it means “if you implement this method then you can call yourself an ActionListener“
Be an `ActionListener`

```java
import java.awt.event.*;
public class MyGUI implements ActionListener
{

    void actionPerformed(ActionEvent e) {
        // Some code here
    }
}
```
Why?

Why would a class want to be an ActionListener?

- So that it can be used in methods whose parameter is of type ActionListener
- For example, JButton has a method
  ```java
  public void addActionListener(ActionListener al)
  ```

The magic of polymorphism means that a MyGUI object can be used for al
Reuse - even in advance

- When an action happens on the JButton it notifies all its registered listeners.
- It does that by calling their `actionPerformed` method.
- It knows that they have an `actionPerformed` method because they have claimed to be an ActionListener.
Some important interfaces

- Large range (dozens) of listeners
  - `MouseListener, KeyListener, MenuListener, ItemListener`
- `Comparable` and `java.util.Comparator`
  - Important for ordering and sorting objects
- `Iterable` and `java.util.Iterator`
  - Important for listing objects in a collection
Iterating

- Often need to examine every object in a list or collection one at a time
  - Called “iterating through the collection”
- The **Iterator** interface provides a unified way of dealing with *any* sort of collection
The class

- Uses generics, so E is any specific type
- Interfaces can be inherited, thus providing superinterfaces and subinterfaces
Its methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext()</td>
<td>Returns true if the iteration has more elements.</td>
</tr>
<tr>
<td>next()</td>
<td>Returns the next element in the iteration.</td>
</tr>
<tr>
<td>remove()</td>
<td>Removes from the underlying collection the last element returned by the iterator (optional operation).</td>
</tr>
</tbody>
</table>
Getting an **Iterator**

- All the Java collection classes have a method called `iterator()` that returns an **Iterator**
  - More precisely, the method returns an object of some class that implements the **Iterator** interface
  - The exact class of the returned object is not important
void showContents(ArrayList<String> a) {
    Iterator<String> it = a.iterator();
    while (it.hasNext()) {
        String s = it.next();
        System.out.println(s);
    }
}

The while-loop keeps calling next on the iterator, until there are no more elements left as each has been processed.
A quick way to iterate

- Java has a special loop known as the “foreach” loop for iterating through arrays
- Suppose `a` is of type `int[]`

```java
    for (int el : a) {
        // process el
    }
```
This works on ArrayLists

```java
void showContents(ArrayList<String> a) {
    for (String s : a) {
        System.out.println(s);
    }
}
```

What else does this “foreach” loop work on?
Anything `Iterable`

- Objects of *any class* that implements the `Iterable` interface can be used
- The `Iterable` interface specifies just one method
  
  ```java
  Iterator<T> iterator()
  ```
Creating your own interface

- Simply declare like a class, but using the word interface
  - No constructors and no method bodies

```java
public interface StoredValue {
    public int getCredit();
    public void topUp(int amount);
}
```
Abstract Classes

- Abstract classes are the mechanism for reusing both implementation and specification
- An abstract class contains a mixture of
  - Instance and class variables
  - Methods with implementations
  - Method signatures with no implementation
Declaring abstract classes

```java
public abstract class Avatar {

    private int age;
    private int xpos, ypos;

    public void moveTo(int x, int y) {
        xpos = x;
        ypos = y;
    }

    public abstract void talk();
}
```

Implemented method same for all Avatars

Abstract method “under construction”
Using abstract classes

- Subclass the abstract class and implement the abstract methods

```java
public class Gnome extends Avatar {
    public void talk() {
        System.out.println("Grr, I’m an ugly gnome");
    }
}

public class Wizard extends Avatar {
    public void talk() {
        System.out.println("Abracadabra, I’m a wizard");
    }
}
```
Notes on Abstract classes

- No constructors, no objects allowed
- If a class has an abstract method it must be declared to be an abstract class
- If a subclass of an abstract class does not override all abstract methods then it too must be declared to be abstract
- A class with all methods abstract is essentially identical to an interface
Multiple Inheritance

What if a gnome gains wizardly powers?

class GnomeWizard extends Gnome, Wizard

This is called *multiple inheritance* and it is *not allowed* in Java, though it is permitted in C++
The diamond problem

Which implementation of `talk()` should the GnomeWizard inherit?
Java’s solution

- Java prohibits multiple inheritance of implementation, but permits multiple inheritance of specification

- A class can *implement* many interfaces

  ```java
  public class Gnome extends Avatar
      implements Talkative
  public class GnomeWizard extends Gnome
      implements Talkative, SpellThrower
  ```