Programming and Software Development
COMP90041

Lecture 10

Polymorphism, Abstract Classes, and Interfaces

Dr. Thomas Christy
thomas.christy@unimelb.edu.au

Slides and materials were developed by Prof Peter Schachte

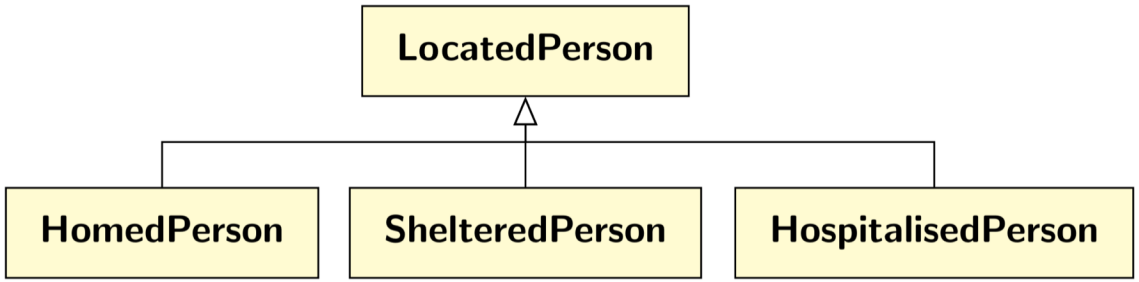
Polymorphism

- Polymorphism refers to the ability to write a single piece of code that handles multiple types of data
- Comes from Greek for many forms or many shapes
- This has two major benefits:
  - It avoids code duplication, making code more succinct
  - It makes code more flexible, allowing code in one part of a program to change without requiring changes elsewhere
- Java supports three kinds of polymorphism:
  - Ad hoc polymorphism — allowing a method to be applied to arguments of different types (overloading)
  - Subtype polymorphism — allowing subtypes to be used in place of supertypes (Liskov substitution; overriding)
  - Parametric polymorphism — allowing types to have parameters (Generics; to be discussed later)

Polymorphism

- `System.out.print` and friends demonstrate both ad hoc and subtype polymorphism:
  - `System.out.print` itself is heavily overloaded to handle all primitive types
  - It handles subclasses of `Object` by calling `toString`
  - `toString` uses overriding to print any object in an appropriate way
- This means code can use `System.out.print` to print anything
- Changing the type of a variable doesn't change how you print it

Abstract Class



- The way we contact a `LocatedPerson` to report finding a missing person varies
- Each subclass will have its own `contactPerson` method:
  - `HomedPerson` will contact them at home
  - `ShelteredPerson` will contact them through the shelter
  - `HospitalisedPerson` will contact them at the hospital
- `LocatedPerson` is an abstraction of all of these

Abstract Methods

- To know how to contact a `LocatedPerson`, we need to know which kind of `LocatedPerson`
- No general way to define `contactPerson` method for `LocatedPerson` class
- But every descendent class of `LocatedPerson` must define `contactPerson` method
- Solution: `LocatedPerson` should define an abstract `contactPerson` method
- An abstract method is a method with a header but no definition
- It specifies a method that must be defined by subclasses

Abstract Methods

- Form:
  `vis abstract type method(params...);`
- Normal method declaration, with the `abstract` keyword, and a semicolon in place of method body
- If abstract method were actually used, there would be no body to execute
- If instance were created, abstract method could be used
- So it must not be possible to make an instance of a class with abstract methods

Abstract Classes

- A class with any abstract methods must be declared to be abstract
- Form: `vis abstract class name { ... }`
- An abstract class is an abstraction of a closely related set of classes
- It serves as a (super-)type for all of these classes, so it is a valid type for variables and parameters
- But you cannot create an instance of an abstract class, only of its concrete subclasses
- Any class not declared `abstract` is a concrete class

Abstract Classses and Inheritance

- Any class that `extends` an abstract class must do one of two things:
  - Implement (override) all its abstract methods; or
  - Be declared `abstract` itself
- An abstract class can implement some but not all of its abstract base class's methods, and may add its own abstract and concrete methods
- An abstract class can extend a concrete class and add some concrete and abstract methods
- A class declared `abstract` is not required to have abstract methods, but it usually does
- A concrete class can extend an abstract base class and add concrete methods, but must override all abstract methods

Abstract `LocatedPerson` Class

```
public abstract class LocatedPerson extends Person {
    public LocatedPerson(int age, String name) {
        super(age, name);
    }
    public abstract String contactInstructions();
}
```

- Now you can't do `new LocatedPerson(...)`
- OK to have variables of type `LocatedPerson`
- Abstract classes can have instance variables and methods
- They can and should have constructors, used by constructor chaining

Concrete `HomedPerson` class

```
public class HomedPerson extends LocatedPerson {
    private String phoneNumber;
    public HomedPerson(int age, String name,
            String phoneNumber) {
        super(age, name);
        this.phoneNumber = phoneNumber;
    }
    public String contactInstructions() {
        return "Returned home; ring " + phoneNumber;
    }
    ⋮ // getter and setter
}
```

Concrete `HospitalisedPerson` class

```
public class HospitalisedPerson extends LocatedPerson {
    private Hospital hospital;
    private String roomNumber;
    public HospitalisedPerson(int age, String name,
            Hospital hospital, String roomNumber) {
        super(age, name);
        this.hospital = hospital;
        this.roomNumber = roomNumber;
    }
    public String contactInstructions() {
        return "Contact at " + hospital +
                " room " + roomNumber;
    }
    ⋮ // getters and setters
}
```

Using LocatedPerson

Now each LostPerson can keep track of who to notify when they are found by adding an instance variable:

```
private LocatedPerson[] seekers = new LocatedPerson[0];
```

This can be used to give notification instructions:

```
public void contactSeekers() {
    for (LocatedPerson p : seekers) {
        System.out.println("Contact " + p);
        System.out.println(p.contactInstructions());
    }
}
```

This is safe because LocatedPerson cannot be instantiated

Interfaces

- Abstract classes allow a number of closely related classes to implement common methods
- A Java interface allows unrelated classes to implement common methods
- Defines a capability some classes have
- An interface is even more abstract than an abstract class:
  - Interfaces cannot have instance or class variables
  - Interfaces cannot have non-abstract or static methods[1]
- An interface specifies only abstract methods (and possibly constants)
- Like an abstract class, an interface is a type

[1] these restrictions are relaxed in Java 8

Use Case

- Imagine a dungeon text adventure game
- Some objects, like a `Scroll`, can be inspected more closely to reveal more detail
- Some objects, like a `Sack`, can be opened to reveal contents
- Some objects can be inspected and opened, such as a `CarvedBox`
- If `Inspectable` and `Openable` were abstract classes, which should `CarvedBox` be derived from?
- A Java class cannot `extend` multiple classes
- Not all `Inspectable` things are `Openable`, and vice versa, so neither can be derived from the other
- Interfaces allow multiple interface inheritance

Defining an Interface

- Form: `public interface name { ...}`
- Like a class, define it in a file by itself
- If you leave off `public`, the interface will have package visibility — `private` and `protected` don't make sense
- The body should contain only abstract method declarations and constant definitions
- All members are implicitly `public`; can omit
- All methods are implicitly `abstract`; can omit
- All variables are implicitly `static final`; can omit
- Method bodies are omitted; just use a semicolon

Using an Interface

- A class can declare that it implements an interface
- Form:
  `public class name implements iface {...}`
- This promises that class *name* defines all interface *iface*'s abstract methods
- (or the class must be declared abstract)
- Like deriving an abstract base class
- Use interface name as a variable or parameter type

Using an Interface

- A class may implement multiple interfaces by following `implements` with multiple interface names separated by commas
- A class may be derived from a base class and implement interfaces by preceding `implements` with `extends` *`BaseClass`*
- Use *`var`* `instanceof` *`Interface`* in code to check if value of *`var`* implements *`Interface`*
- If so, then cast *`var`* to *`Interface`*, so you can use *`Interface`* methods

Abstract Base Class

- A `DungeonItem` is the root of the hierarchy of things that can appear in the dungeon
- All that unites every class of dungeon item is that it has a description
- It is more flexible to implement this with a method rather than an instance variable
  - Can implement it with an instance variable and a getter
  - Or can piece together description from the object
  - Or even have a method that returns a constant string if every instance of a class always has the same description

```
public abstract class DungeonItem {
    public abstract String getDescription();
}
```

Interfaces

```
public interface Inspectable {
    String detailIntro = "Looking closer, you see ";
    String getDetailedDescription();
}
```

- Note `Inspectable` interface defines a constant
- Classes that implement `Inspectable` can use it

```
public interface Openable {
    boolean isOpen();
    void open();
    void close();
}
```

- Note `public`, `static final`, and `abstract` are omitted

Scroll

```
public class Scroll extends DungeonItem
                    implements Inspectable {
    private final String scrollText;
    public Scroll(String scrollText) {
        this.scrollText = scrollText;
    }
    public String getDescription() {
        return "a faded scroll, covered with barely" +
               " decypherable ornate lettering";
    }
    public String getDetailedDescription() {
        return detailIntro +
               "the following words:\n" + scrollText;
    }
}
```

Sack

```
public class Sack extends DungeonItem
                  implements Openable {
    String detail;
    DungeonItem[] contents = new DungeonItem[0];
    private boolean tied = false;
    public Sack(String detail) {
        this.detail = detail;
    }
    public boolean isOpen() { return !tied; }
    public void open() { tied = false; }
    public void close() { tied = true; }
```

Sack (Continued)

```
public void addItem(DungeonItem item) {
    if (item instanceof Openable) {
        ((Openable)item).close();
    }
    DungeonItem[] newContents =
            new DungeonItem[contents.length+1];
    for (int i = 0; i<contents.length; ++i) {
        newContents[i] = contents[i];
    }
    newContents[contents.length] = item;
    contents = newContents;
}
```

Sack (Continued)

```
    public String getDescription() {
        String descr = "a " + detail + " sack ";
        if (tied) {
            descr += "tied shut with a bit of twine";
        } else {
            descr += "containing" +
                    (contents.length > 0 ? ":" :
                    " nothing at all");
            for (DungeonItem item : contents) {
                descr += "\n\t"+item.getDescription();
            }
        }
        return descr;
    }
}
```

CarvedBox

```
public class CarvedBox extends DungeonItem
                    implements Openable, Inspectable {
    private final String inscription;
    private DungeonItem contents = null;
    private boolean open = false;
    public CarvedBox(String inscription) {
        this.inscription = inscription;
    }
    public String getDetailedDescription() {
        if (open) return "close the box to inspect it";
        return detailIntro + " the inscription '" +
                inscription + "'";
    }
    public boolean isOpen() { return open; }
    public void open() { open = true; }
    public void close() { open = false; }
```

CarvedBox (Continued)

```
    public String getDescription() {
        String descr = "an ornately carved wooden box";
        if (open) {
            descr += " standing open to reveal";
            if (contents == null) {
                descr += " that it is empty";
            } else {
                descr += ":\n    " +
                         contents.getDescription();
            }
        } else {
            descr += " with fine runes on the cover";
        }
        return descr;
    }
}
```

DungeonTest

```
public class DungeonTest {
    // main method (next slide)
    private static void showMe(DungeonItem item,
                               String name) {
        System.out.println(name + ": " +
            item.getDescription());
        if (item instanceof Inspectable) {
            Inspectable ins = (Inspectable)item;
            System.out.println("Detail: " +
                ins.getDetailedDescription());
        }
    }
}
```

DungeonTest (Main method)

```
public static void main(String[] args) {
    Scroll scroll = new Scroll("Please Turn Over");
    showMe(scroll, "Scroll");
    CarvedBox box = new CarvedBox("Pandora beware");
    showMe(box, "Box");
    box.open();  System.out.println("Opened it");
    showMe(box, "Box");
    box.addItem(scroll); System.out.println("Added scrol
    showMe(box, "Box");
    Sack bag = new Sack("brown felt");
    showMe(bag, "Sack");
    bag.addItem(box); System.out.println("Added box");
    showMe(bag, "Sack");
    bag.close(); System.out.println("Closed it");
    showMe(bag, "Sack");
}
```

Extending Interfaces

- An interface can be declared to `extend` one or more interfaces
- Form:

  `public interface` *name*
  `extends` $iface_1, iface_2, \ldots$ `{ ...}`
- The interface *name* is the union of all the abstract methods and constants in the specified interfaces
- . . . plus the abstract methods and constants in the body

Conflicting Interfaces

- An interface cannot extend two interfaces that specify the same abstract method with different return types
- Or two interfaces with different meanings for the same signature
- Similarly a class can't implement two conflicting interfaces
- Same problem inheriting from a (possibly abstract) class with methods that conflict with a method in an interface the class implements
- Sorry, just can't combine those interfaces/classes

Java Standard Interfaces

- The `Comparable` interface specifies classes that allow two objects to be compared for less or greater
- `Comparable` interface specifies only one method:

```
int compareTo(Object o)
```

- `o1.compareTo(o2)` should return:
  - A negative number if `o1` $<$ `o2`
  - Zero if `o1` $=$ `o2`
  - A positive number if `o1` $>$ `o2`
- If class `C implements Comparable`, and `a` is an array of `C`, then `Arrays.sort(a)` will sort `a`
- You'll need to `import java.util.Arrays;`

Guidance

- Use non-abstract class inheritance if there's a meaningful class with a few variations with different ways to do something
- Use an abstract class when you have a limited variety of types each with its own way of doing something, but no "generic" version
- Use an interface when many unrelated classes should support some operations
- Use interfaces when some classes may support several unrelated operations
- But interfaces can't provide method definitions or instance variables

Summary

- 3 kinds of polymorphism: ad hoc (overloading), subtype (inheritance/overriding), and parametric
- Abstract methods have only method headers without bodies
- Classes with abstract methods must be declared abstract
- Concrete subclasses of abstract classes must specify the definitions
- Interface includes only abstract methods and constants
- Class can `implement` one or more interfaces by giving definitions for abstract methods

THE UNIVERSITY OF
MELBOURNE