COMP90041
Programming and Software Development
2020 - Semester 1
Lab 8 - Week 9

Yuhao(Beacon) Song

yuhsong1@unimelb.edu.au

Introduction

- Timetable
  - Tue(11) 14:15-15:15 (Melbourne Time) Join URL: https://unimelb.zoom.us/j/490084146
  - Tue(07) 16:15-17.15 (Melbourne Time) Join URL: https://unimelb.zoom.us/j/291505765

- Contact
  - yuhsong1@unimelb.edu.au
  - yuhsong@student.unimelb.edu.au
  - GitHub: https://github.com/Beaconsyh08/COMP90041_Programming_and_Software_Development_Tutorials.git

Outline

- Lecture Review
- Exercise
- Project C

Object-Oriented Programming (OOP)

- There are three main programming mechanisms that constitute **OOP**
  - **Encapsulation**
    - This is the practice of keeping fields within a class **private**, then providing access to them via **public methods**. It's a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide.
  - **Inheritance**
    - This is a special feature of OOP in Java. It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.
  - **Polymorphism**
    - This Java OOP concept lets programmers use the same word to mean different things in different contexts. One form of polymorphism in Java is **method overloading**. The other form is **method overriding**.

Polymorphism

- **Polymorphism** is the ability to associate many meanings **to one method name**

- **Recap**
- **Overloading**: same class or could happen in derived class and base class; different **signature** and same **name**.
- **Overriding**: derived class and base class; same **signature** and **returned type**.

Late Binding/Dynamic Binding

- The process of associating a method **definition** with a method **invocation** is called **binding**
- If the method definition is associated with its invocation when the code is compiled, that is called **early binding or static binding**
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called **late binding or dynamic binding**
- Java uses **late binding** for almost all methods
  - Except private, final, and static methods: these method cannot be overridden and the type of the class is determined at the compile time.

Upcasting and Downcasting

- **Upcasting** is casting to a supertype
  - Dog dog = new Dog();
  - Animal upcastedAnimal = (Animal) dog;
- **Downcasting** is casting to a subtype
  - Animal animal = new **Dog**();
  - Dog downcastedDog = (Dog) animal;
- **ClassCastException – Run Time**
  - Animal animal = new **Animal**();
  - Dog downcastedDog = (Dog) animal;
  - animal's **runtime type** is Animal, and so when you tell the runtime to perform the cast it sees that animal is not really a Dog.
- **Upcasting** is always allowed, but **Downcasting** need to be done by the programmer manually

Abstract Method

- An **abstract method** has a heading, but no method body(no implementation)
  - public **abstract** int sumOfTwo(int n1, int n2);
  - public **abstract** int sumOfThree(int n1, int n2, int n3);

- The body of the method is defined in the derived classes
  - public int sumOfTwo(int num1, int num2){return num1+num2;}
  - public int sumOfThree(int num1, int num2, int num3){return num1+num2+num3;}

- An abstract method is like a placeholder for a method that will be fully defined in a **descendent class**. It has a complete method heading with modifier **abstract**

Abstract Classes

- If a **class** has an abstract method it should be declared **abstract**, the vice versa is not true, which means an **abstract class** doesn't need to have an abstract method compulsory.
- If a **regular class** extends an **abstract class**, then the class must have to implement **all** the abstract methods of abstract parent class or it has to be declared **abstract** as well.
- A class that has no abstract methods is called a **concrete class**
- **Cannot be private**, for abstract class and abstract method

You Cannot Create Instances of an Abstract Class

- An **abstract class** can only be used to derive more specialized classes
  - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker (Example from lecture)
- An **abstract class constructor** cannot be used to create an object of the abstract class
  - However, a derived class constructor will include an invocation of the abstract class constructor in the form of super
  - The constructor in an abstract class is only used by the constructor of its derived classes

Interfaces

- An **interface** is something like an **extreme case** of an **abstract class**, but not a class
- The syntax for defining an interface is use word interface
- It contains method headings and constant definitions **only**, and no instance variables nor any **complete** method definitions
- An interface and **all of** its method headings should be declared **public**
- When a class implements an interface, it must implement **all the methods** in the interface
- Any variables defined in an interface must be public, static, and final. Because this is understood, Java allows these modifiers to be omitted

Implementing Interfaces

- The classes may implement one or more interfaces, If more than one interface is implemented, each is listed, separated by commas
- It must include the phrase *implements Interface_Name* at the start of the class definition, if the class extends another class, implements go after extends.

- A **concrete class** must give definitions for **all** the method headings given in the abstract class and the interface
- A **abstract class** could also implements interface. Any method headings given in the interface that are not given definitions are automatically made into abstract methods

Derived Interfaces

- Like classes, an interface may be derived from a base interface
  - This is called extending the interface
  - The derived interface must include the phrase ***extends*** *BaseInterfaceName*
- A **concrete class** that implements a derived interface must have definitions for **any methods** in the derived interface as well as any methods in the base interface

Abstract class vs Interface

	Abstract class	Interface
Type of methods	abstract and non-abstract	only abstract
Final Variables	final or non-final	by default final
Type of variables	final, non-final, static and non-static	only static and final
Implementation	can provide the implementation of interface	can't provide the implementation of abstract class.
Inheritance vs Abstraction	extends	implements
Multiple implementation	extend another Java class and implement multiple Java interfaces	extend another Java interface only
Accessibility of Data Members	private, protected, etc	public by default

Chap8_Question5

5. Consider a graphics system that has classes for various figures—say, rectangles, boxes, triangles, circles, and so on. For example, a rectangle might have data members' height, width, and center point, while a box and circle might have only a center point and an edge length or radius, respectively. In a well-designed system, these would be derived from a common class, `Figure`. You are to implement such a system.

   The class `Figure` is the base class. You should add only `Rectangle` and `Triangle` classes derived from `Figure`. Each class has stubs for methods `erase` and `draw`. Each of these methods outputs a message telling the name of the class and what method has been called. Because these are just stubs, they do nothing more than output this message. The method `center` calls the `erase` and `draw` methods to erase and redraw the figure at the center. Because you have only stubs for `erase` and `draw`, center will not do any "centering" but will call the methods `erase` and `draw`, which will allow you to see which versions of `draw` and `center` it calls. Also, add an output message in the method `center` that announces that `center` is being called. The methods should take no arguments. Also, define a demonstration program for your classes.

   For a real example, you would have to replace the definition of each of these methods with code to do the actual drawing. You will be asked to do this in Programming Project 8.6.

Chap8_Question6

6. Flesh out Programming Project 8.5. Give new definitions for the various constructors and methods `center`, `draw`, and `erase` of the class `Figure`; `draw` and `erase` of the class `Triangle`; and `draw` and `erase` of the class `Rectangle`. Use character graphics; that is, the various `draw` methods will place regular keyboard characters on the screen in the desired shape. Use the character `'*'` for all the character graphics. That way, the `draw` methods actually draw figures on the screen by placing the character `'*'` at suitable locations on the screen. For the `erase` methods, you can simply clear the screen (by outputting blank lines or by doing something more sophisticated). There are a lot of details in this project, and you will have to decide on some of them on your own.

Chap8_Question7

7. Define a class named `MultiItemSale` that represents a sale of multiple items of type `Sale` given in Display 8.1 (or of the types of any of its descendent classes). The class `MultiItemSale` will have an instance variable whose type is `Sale[]`, which will be used as a partially filled array. There will also be another instance variable of type `int` that keeps track of how much of this array is currently used. The exact details on methods and other instance variables, if any, are up to you. Use this class in a program that obtains information for items of type `Sale` and of type `DiscountSale` (Display 8.2) and that computes the total bill for the list of items sold.