COMP90041
Programming and Software Development
2020 - Semester 1
Lab 7 - Week 8

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
- Demo
- Project C

Inheritance

- Inheritance is one of the main techniques of **object-oriented programming (OOP)**
- Using this technique, a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods
- **General Class**: base/parent/superclass
- **Specialized Class**: derived/child/subclass
- The **derived class** inherits all the **non-private (methods**, **instance variables**, **static variables)** but can add more.
- The phrase **extends BaseClass** must be added to the derived class definition:

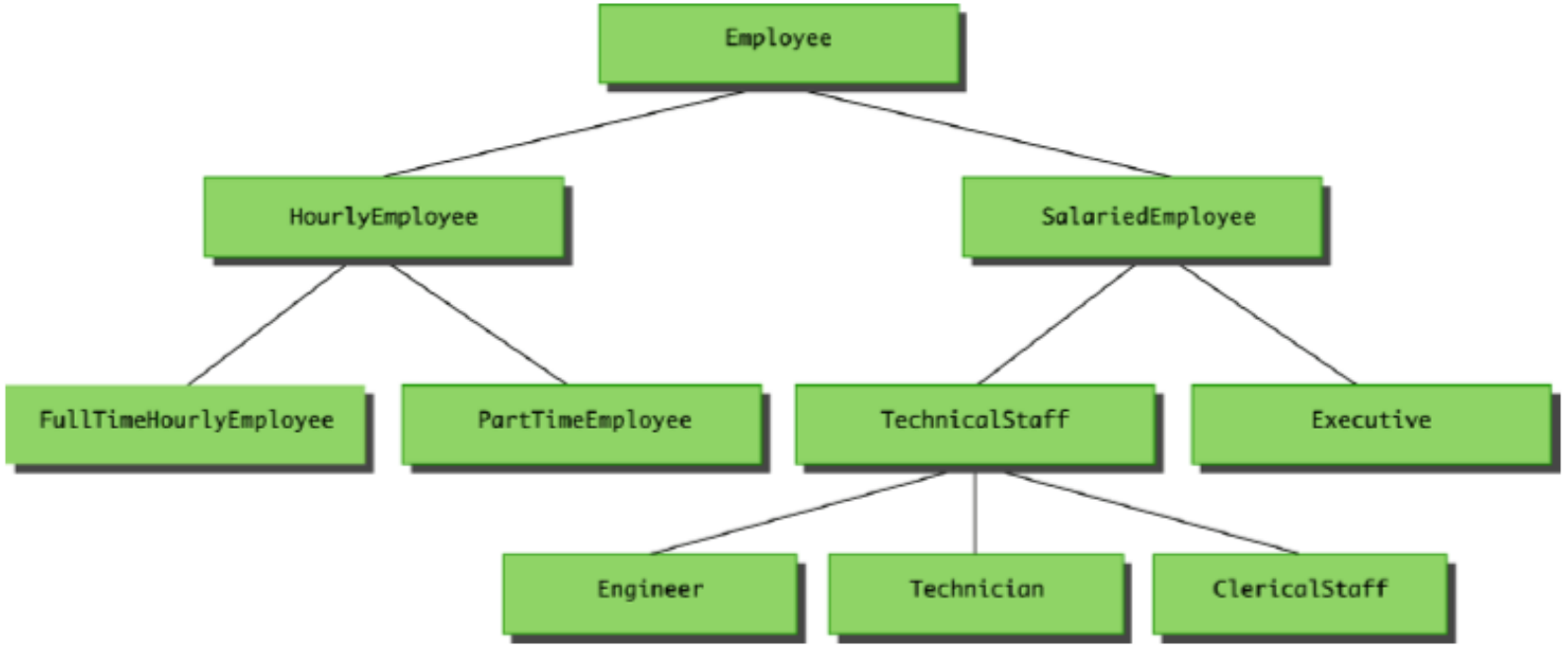
  *public class **subclass** **extends** **superclass***

- Inheritance is especially advantageous because it allows code to be **reused**, without having to copy it into the definitions of the derived classes

Ancestor and descendent Classes

- These relationships are often extended such that a class that is a parent of a parent . . . of another class is called an **ancestor class**
  - If class A is an ancestor of class B, then class B can be called a **descendent** of class A

Display 7.1 **A Class Hierarchy**



The this Constructor

- Within the definition of a constructor for a class, **this** can be used as a name for invoking another constructor in the same class
- Often, a no-argument constructor uses this to invoke an explicit-value constructor

```
    public Dog() {
        this( spec: "spec",  name: "python",  age: 5,  color: "red");  // invoke dog con 1
//          this("spec", "python", 5, "red", 10);  // invoke dog con 2
    }

    // Dog constructor 1
    public Dog(String spec, String name, int age, String color) {
        super(spec, name, age);  // invoke animals con
        this.color = color;
    }

    // Dog constructor 2
    public Dog(String spec, String name, int age, String color, int height) {
        super(spec, name, age);   // invoke animals con
        this.color = color;
        this.height = height;
    }
```

OVERLOADING

The super Constructor

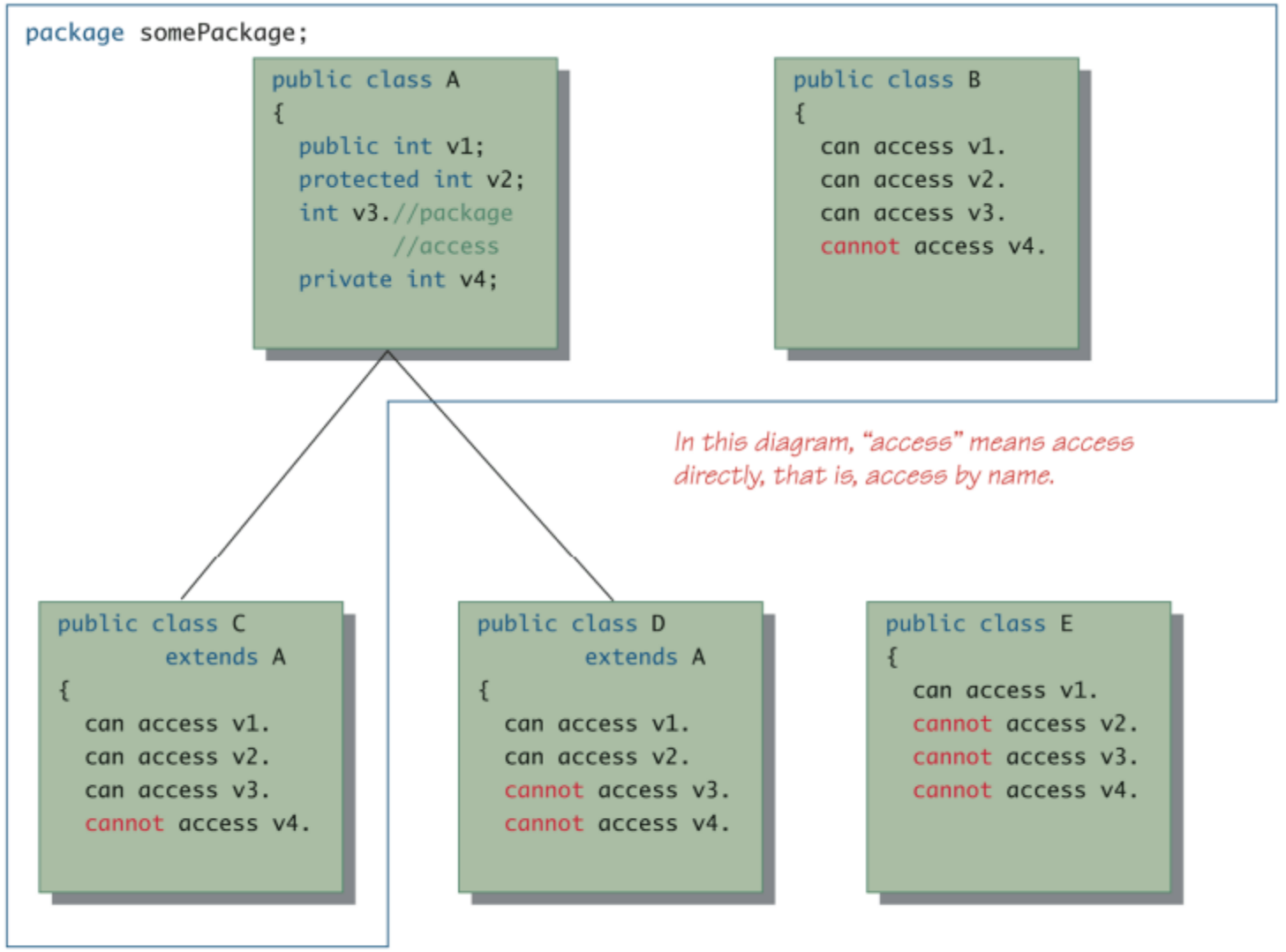
- A derived class uses a constructor from the **base** class to initialize all the data inherited from the **base** class
- A call to the base class constructor can never use the name of the base class, but uses the keyword **super** instead
  - A call to super must always be **the first action** taken in a constructor definition
  - If a derived class constructor does not include an invocation of **super**, then the **no-argument constructor** of the **base class** will automatically be invoked (super())
  - This can result in an error if the base class has not defined a no-argument constructor
  - Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to super should always be used

Access Modifiers

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
public	+	+	+	+	+
protected	+	+	+	+	
no modifier	+	+	+		
private	+				

+ : accessible blank : not accessible

Display 7.9 Access Modifiers



The Class Object

- In Java, every class is a descendent of the **class Object**
  - Every class has Object as its ancestor
  - Every object of every class is of type Object, as well as being of the type of its own class
- If a class is defined that is not explicitly a derived class of another class, it is still automatically a derived class of the class Object
- The class Object is in the package java.lang which is always imported automatically
- The class Object has some methods that every Java class inherits
  - For example, the equals and toString methods
- However, these inherited methods should be **overridden** with definitions more appropriate to a given class

Overriding a Method Definition

- **Override** → a new definition of the method
- The definition of an inherited method can be changed in the definition of a **derived** class so that it has a different meaning in the **derived** class
- The redefined method in the **derived** class should use same **method signature** and **returned type** with the method in the **super** class
- **Recap**
- **Signature**: the method name and the list of types for parameters →

Return Type Not Included!

```
public void setDate(int month, int day,int year)
public void setDate(String month, int day,int year)
public void setDate(int year)

setDate(int, int, int)
setDate(String, int, int)
setDate(int)
```

Changing the Return Type

- Ordinarily, the **type returned** may **not** be changed when **overriding** a method
- However, if it is a **class type**, then the returned type may be changed to that of **any descendent class** of the returned type → **covariant return type**

```
public DemoAnimals getObject(){
    return new DemoAnimals( spec: "animals",  name: "animal object",  age: 10);
}
```

```
@Override
public DemoDog getObject() {
    return new DemoDog();
}
```

Changing the Access Permission

- The access permission of an **overridden** method can be changed from **protected** in the base class to more permissive access in the derived class
- **protected → protected or public (√)**
- **Package → pacakage, protected or public**

- However, the access permission of an **overridden** method can **not** be changed from **public** in the base class to a more restricted access permission in the derived class
- **public → private (×)**

The final Modifier

- If the modifier **final** is placed before the definition of a **method**, then that method may not be redefined in a **derived class (not allowed overriding)**

- If the modifier **final** is placed before the definition of a **class**, then that class may not be used as a **base class** to derive other classes **(not allowed extends)**

Pitfall: Overriding Versus Overloading

- **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**.

- Note that when the derived class overloads the original method, it still inherits the original method from the base class as well

```java
public class TestOverriding {
    public static void main(String[] args) {
        A a = new A();
        a.p( i: 10);
        a.p( i: 10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // overrides the method in B
    @Override
    public void p(double i) {
        System.out.println(i);
    }
}
```

```java
public class TestOverloading {
    public static void main(String[] args) {
        C c = new C();
        c.p( i: 10);
        c.p( i: 10.0);
    }
}

class D {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class C extends D {
    // overloads the method in D
    public void p(int i) {
        System.out.println(i);
    }
}
```