

**COMP90041**  
Programming and Software Development  
2020 - Semester 2  
**Lab 7**

Yuhao(Beacon) Song  
[yuhsong1@unimelb.edu.au](mailto:yuhsong1@unimelb.edu.au)  
[yuhsong@student.unimelb.edu.au](mailto:yuhsong@student.unimelb.edu.au)

# Introduction

- ❖ **Timetable**

- ❖ Tue(18) 16:15-17:15 (Melbourne Time)
  - ❖ <https://unimelb.zoom.us/j/94854648719?pwd=WUY0NmR6MkI5UVZBUWhGNWFIU216Zz09>
- ❖ Wed(19) 17:15-18:15 (Melbourne Time)
  - ❖ <https://unimelb.zoom.us/j/93723434766?pwd=MGh4QkJuZnhJR21LZ0VqeXhJQU52UT09>

- ❖ **GitHub Page (Tutorial Materials, Solutions, Additional Resources)**

- ❖ <https://github.com/Beaconsyh08/COMP90041-2020SEM2>

- ❖ **PollEv**

- ❖ <https://pollev.com/yuhsong>

# Outline

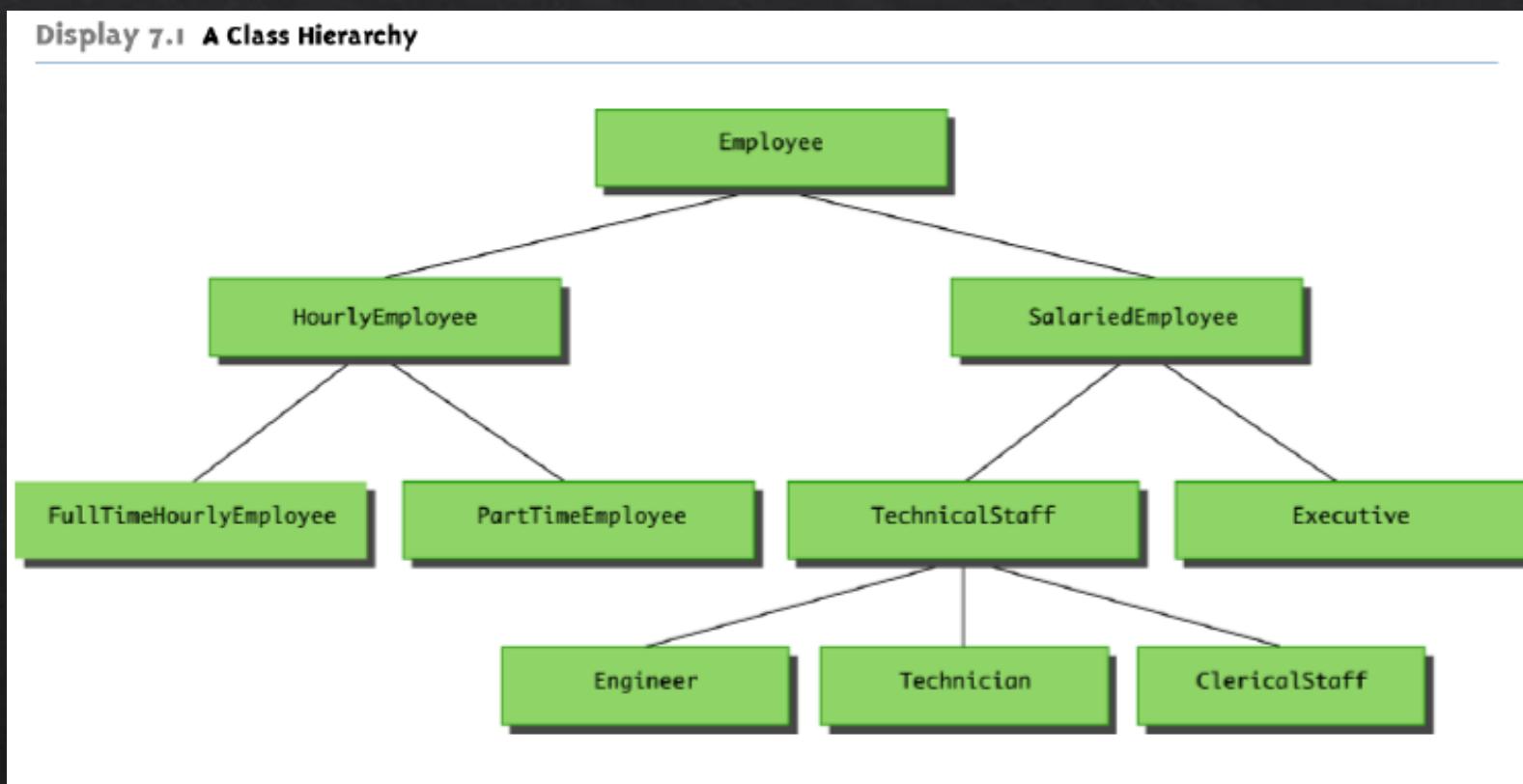
- ◆ Array Practice
- ◆ Lecture Review
- ◆ Demo

# 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 descendant 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



# Access Modifiers

|             | Class | Package | Subclass<br>(same pkg) | Subclass<br>(diff pkg) | World |
|-------------|-------|---------|------------------------|------------------------|-------|
| public      | +     | +       | +                      | +                      | +     |
| protected   | +     | +       | +                      | +                      |       |
| no modifier | +     | +       | +                      |                        |       |
| private     | +     |         |                        |                        |       |

+ : accessible      blank : not accessible

Display 7.9 Access Modifiers

```
package somePackage;
```

```
public class A
{
    public int v1;
    protected int v2;
    int v3;//package
        //access
    private int v4;
```

```
public class B
{
    can access v1.
    can access v2.
    can access v3.
    cannot access v4.
```

```
public class C
    extends A
{
    can access v1.
    can access v2.
    can access v3.
    cannot access v4.
```

```
public class D
    extends A
{
    can access v1.
    can access v2.
    cannot access v3.
    cannot access v4.
```

```
public class E
{
    can access v1.
    cannot access v2.
    cannot access v3.
    cannot access v4.
```

In this diagram, "access" means access directly, that is, access by name.

A line from one class to another means the lower class is a derived class of the higher class.

If the instance variables are replaced by methods, the same access rules apply.

# 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**

# 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)
```

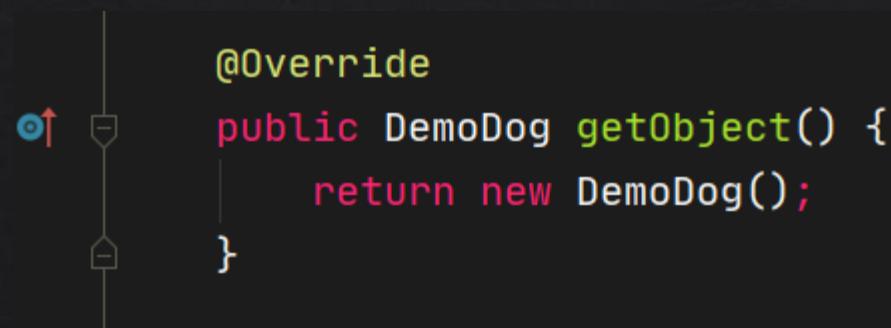
# Changing the Return Type

- ❖ 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**



```
class DemoAnimals {  
    public DemoAnimals get0bject(){  
        return new DemoAnimals( spec: "animals", name: "animal object", age: 10);  
    }  
}
```

The diagram shows a vertical inheritance relationship between two classes. At the top is a class icon with a blue arrow pointing down to a class definition. Below it is another class icon with a grey arrow pointing up to a class definition. This indicates that the class below is a subtype of the class above.



```
@Override  
class DemoDog extends DemoAnimals {  
    public DemoDog get0bject() {  
        return new DemoDog();  
    }  
}
```

The diagram shows a vertical inheritance relationship between two classes. At the top is a class icon with a blue arrow pointing up to a class definition. Below it is another class icon with a grey arrow pointing up to the same class definition. This indicates that the class below is a subtype of the class above.

# 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 → package, 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 (✗)**

# 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

```
▶ public class TestOverriding {  
▶     public static void main(String[] args) {  
▶         A a = new A();  
▶         a.p(10);  
▶         a.p(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);  
●     }  
● }
```

```
▶ public class TestOverloading {  
▶     public static void main(String[] args) {  
▶         C c = new C();  
▶         c.p(10);  
▶         c.p(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);  
●     }  
● }
```