Inheritance

Yangtao Ge

June 18, 2019

1 Classes, Superclasses and Subclasses

These relationships are usually related to an '*is-a*' relationship. (i.e. Manager is an employee)

1.1 Defining Subclasses

Basic Grammar:

```
public class Manager extends Employee{
    ...
}
```

Some feature of using '**extends**':

- Subclasses $\xrightarrow{extends}$ Superclasses
- Subclasses have *more* functionalities than their Superclasses
- Subclasses can use method and field from *Superclasses*
  (i.e. new avaliable = original + extended)
- point out the differences between original and new classes (***Factoring***)

1.2 Overriding Methods

Definition: In a subclasses, it has the same **method (name + parameter)** as its superclass, the functionalities are different from its original one.
it is different from '*overloading*' (same name + different parameter)
e.g.

```
public double getSalary(){
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```

Some notices:

- we have no access to '*salary*' (private field in Employee)
- we cannot plainly use '*getSalary()*' (it means call it self – infinit loop)
- '*super*' here is not a reference to an object,
  it is just for invoking superclass method

1.3 Subclass Constructors

The *second* way of using 'super' is build contructors:

```
public Manager(String name, double salary,
               int year, int month, int day){
    super(name, salary, year, month, day);
    bonus = 0;
}
```

N.B. When *super* Constructor is not used, no-argument Constructor is applied

Summary of 'this' and 'super':

- **this**:
  - denote a reference to the implicit parameter
  - call another constructor of the same class

- **super**:
  - invoke the super class method
  - invoke a superclass constructor
- when both used for 'constructor': be careful about *this class(this)* or *superclass(super)*

****polymorphism** means an object variable can refer to *multiple* actual types.(*Ref: p.217 & subsection 1.5*)

1.4 Inheritance Hierarchies

Inheritance can have **more than one** layer,
e.g. Employee ← Manager ← Executive (this path is '*inheritance chain*')

Ref: pp.216-217, especially Figure 5.1

1.5 Polymorphism

polymorphism means an object variable can refer to *multiple* actual types.
(i.e. every Manager is an employee, but not every employee is a Manager)
e.g.1 (*Every Manager is employee*)

```
Employee e;
e = new Employee(...); //Employee Object expected
e = new Manager(...); // Manager is a subclass of Employee
```

e.g.2 (*Not every employee is a Manager*)

```
Manager boss = new Manager(...);
Employee[] stafflist = new Employee[3];
stafflist[0] = boss;

boss.setBonus(5000); // correct
staff[0].setBonus(5000); //ERROR
```

1.6 Understanding Method calls

Understand what happends:

1. know all possible candidates for the method to be called
2. knows the name and parameter types of the method
3. decide '*static binding*' or '*dynamic binding*'
   - **static**: has modifier 'private', 'static', 'final' or a constructor
   - **dynamic**: depends on the actual type of the '**implicit** parameter'
4. running the 'static' or 'dynamic':
   - static: run the method call indicated by step 3.
   - dynamic: call the verision of the method that is appropriate for the '*actual*' type by '**method table**' (method table list all *method signatures* and *actual methods to be called*)
     e.g. '*e.getSalary()*'
     (a) fetches the method table for the actual type of '*e*'
     (b) lookup for defining classs of the signatures '*getSalary()*'
     (c) get the correct method

i.e. **Method table** is:
Exp: For actual type 'Manage' the following signatures are corresponding to the catual method call.
Manager:

- getName() → Employee.getName()
- getSalary() → Employee.getSalary()
- getHireDay() → Employee.getHireDay()
- raiseSalary(double) → Employee.raiseSalary()
- setBonus(double) → Manager.setBonus()

1.7 Preventing Inheritance: *Final* Classes and Method