Inheritance

Yangtao Ge

June 24, 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

Definition: Classes that cannot be extended are called *final* classes. (i.e. Preventing others from forming a 'subclass' of one of your classes)
e.g.

```
public final class Executive extends Manager{
    ...
}
```

Definition: Method that cannot be overriden are called *final* methods.

All methods in a '*final* class' are automatically *final*

The **Only** good reason to make classes/methods *final* is:
To make its semantics cannot be changed in a subclass

Definition: If a method is not overriden, and it is short, then a compiler can optimize the method call. It is called *inlining* (*Ref: pp.222-223 & COMP0012 Compiler*)

1.8 Casting

Definition: The process of **forcing** a conversion from one type to another.
e.g.

```
double x = x = 3.406;
int nx = (int) x;
```

Some features to consider about:

- Cast only within an inheritance hierarchy
- Use '*instanceof*' to check before casting from a superclass to a subclass
  e.g.

```
if (staff[1] instanceof Manager){
    boss = (Manager) staff[1];
}
```

The reason for doing a cast is use a object in its full capacity (i.e. use special method '*setBouns()*')

1.9 Abstract Classes

Definition: *Abstract* (class): is a superclass that cannot be instantiated and is used to state or define general characteristics

Some features to remember:

- using '*abstract*' → don't need to implement the method at all
  e.g.

```
public abstract class Person{
    private String name;
    public Person(String name){
        this.name = name;
    }
    // just a signature
    public abstract String getDescription();

    public String getName(){
        return name;
    }
}
```

- when extending a abstract class – 2 choices:
  - left methods undefined → tag the subclass '*abstract*'
  - define all method → it is a normal subclass(concrete one)
- Class can be tagged as '*abstract*' with no abstract methods
- Abstract class **cannot** be instantiated → no objects can be created
- Abstract class **can** be *object variables*, but need to refer to an object of *concrete subclass*
  e.g.

```
Person p = new Student('Yangtao.G','Comp_Sci');
```

Ref: pp.227-229 & Chapter 6 Interfaces

1.10 Protected Access

When to use Protected:

- field: two cases
  - restrict a method to subclasses only
  - allow subclass methods to access a superclass (less common)

  N.B. Protected field is accessible by any class in the same package, so be cautious when using item
- method: the subclasses can be trusted to use the method correctly.(more common to use)

Summary of four access modifiers:

- private: in Class only
- public: by the world, everywhere
- protect: in the package and all subclasses
- 'no modifier': default is accessible in the package

2 *Object:* The Cosmic Superclass

Object method: Only cover equals, hashcode, toString Methods, In Chapter 12 concurrency, more method will be covered

2.1 Variables of Type 'object'

Variables '*object*' can refer to any type e.g.

```
Object obj = new Employee(``Harry Porter'', 35000);
```

Only value of *primitive types* are not objects

All array types are classes extended from 'Object' class

2.2 The '*equals*' Method

Definition: *'equals'* method tests whether one **object** is equal to another. Differents between *equals* (equal) and == (identical):

- ==: checks if both objects point to the same memory location
- **equals()**: evaluates to the comparision of value in the object

N.B. identical → equal (True); equal → identical (False)

This method usually needs implementing to do state-based equality testing (whether they are in the same state)
e.g.

```
public class Employee{
    ...
    public boolean equals(Object otherObject){
        // whether the objects are identical
        if (this == otherObject) return true;
        // explicit para is null
        if (otherObject == null) return false;
        // class don't match
        if (getClass() != otherObject.getClass){
            return false;
        }
        // After checking 3 conditions above
        // it must be a non-null Employee
        Employee other = (Employee) otherObject;
        // whether the fields have identical values
        return name.equal(other.name)
            && salary == other.salary
            && Objects.equals(hireDay, other.hireDay);
    }
}
```

When defining 'equals' method for a subclass, we need to call 'equals' on its *'superclass'*
e.g.

```
public class Manager extends Employee{
    ...
    public boolean equals(Object otherObject){
        //check belongs to the same class
        if(!super.equals(otherObject)) return false;
        Manager other = (Manager) otherObject;
        return bonus == other.bonus;
    }
}
```

2.3 Equality Testing and Inheritance

The requirement for '*equals* method':

- **Reflexive**: for non-null reference x, *x.equals(x)* return true
- **Symmetric**: for ref x & y, *x.equals(y)* $\Leftrightarrow$ *y.equals(x)*
- **Transitive**: for ref x, y & z, *x.equals(y)* $\wedge$ *y.equals(z)* $\rightarrow$ *x.equals(z)*
- **consistent**: x, y not change $\rightarrow$ *x.equals(y)* remains the same
- for any non-null ref x, *x.equal(null)* is false

Recipe for writting the perfect '*equals* method':

1. name the explicit parameter '*otherObject*' (cast it to '*other*' later)
2. test itdentity:
   ```
   if (this == otherObject) return true;
   ```
3. test whether it is null:
   ```
   if (otherObject == null) return false;
   ```
4. Compare the classes of *this* and *otherObject*:
   - If the semantics of 'equals' can change in subclasses: Using **getClass()**

```
if(getClass() != otherObject.getClass()) return false;
```

- If the same semantics holds for all subclasses: Using **instanceof**

```
if(!(otherObject instanceof ClassName)) return false;
```

5. Cast *otherObject* to a variabe of current class type:

```
ClassName other = (ClassName) otherObject;
```

6. Compare the fields:

   - Use '==' for primitive type fields
   - Use 'Object.equals' for object fields

```
return field1 == othe.field1
    && Objecs.equals(field2, other.field2)
```

Common mistake: Cannot declares the method para as 'explicit para' (e.g. Employee). protect from it by adding '**@Override**'

2.4 The *hashCode* Method

Definition: A hash code is an integer that derived from an object.

Two types of *hashcode* method:

- default hashcode: derives the hash code from the object's memory address
- redefined hashcode: has it own algorithm. e.g. Strings derives from its *content*

Usually, 'equals' method is related to 'hashcode' method (redefine together):

- *x.equal(y)* $\rightarrow$ *x.hashCode() == y.hashCode()*
- define equal on employee ID $\rightarrow$ hashcode method only hashes employee ID

An examples of 'hashcode' method:
e.g.1

```
public int hashCode(){
    return 7 * Objects.hashCode(name)
        + 11 * Double.hashCode(salary)
        + 13 * Objects.hashCode(hireDay);
}
```

e.g.2 use '*hash()* method'

```
public int hashCode(){
    return Objects.hash(name, salary, hireDay);
}
```

2.5 The *toString* Method

Ref:pp.241-243

A normal *toSrting* method:

```
public String toString(){
    return getClass().getName()
        + "[name=" + name
        + ", salary=" + salary
        + ", hireDay=" + hireDay
        + "]";
}
```

Every object can be printed because of 'toString' method

It is useful for logging in Chapter 7.

3 Generic Array Lists

In Java we can set the size of an array at **running time**. Also, we could use *generic type* e.g. *ArrayList* has ability to adjust the capacity

** A generic type is a generic class or interface that is parameterized over types; usually using diamond notation (<Type>)
Ref: Chapter 8 & Java Doc

3.1 Declaring Array Lists

Three ways to init a Array List:

- *ArrayList<Employee> staff = new ArrayList<Employee>()*
- *var staff = new ArrayList<Employee>()* (After Java 10)
- *ArrayList<Employee> staff = new ArrayList<>()*

Differents between 'ArrayList' and 'array':

- array: *new Employee[100]* → the size of the array is 100 (cannot be changed)
- ArrayList: *new ArrayList<>(100)* The capacity of ArrayList is 100, but the size is 'zero'
  - extendable: *new ArrayList<>(100)* or *staff.ensureCapacity(100)*
  - unextendable: *staff.trimToSize(100)*

3.2 Accessing Array List Elements

Things to Notice:

- Array: When init for a certain size → all the slot are avaliable
- ArrayList: When init for a certain capacity → nothing is in the ArrayList

A way of convience:

- Make an ArrayList and all all Elements:

```
var list = new ArrayList<X>();
while(...){
    x = ...;
    list.add(x);
}
```

- use *toArray()* method to copy the elements into an array:

```
var a = new X[list.size()];
list.toArray();
```

We can use enhance loop to traverse the contents of an ArrayList:

```
for(Employee e : staff){
    do something
}
```

3.3 Compatibility between Typed and Raw ArrayLists

The limitation of generic types in Java:
Compiler translates all typed ArrayLists into raw ArrayList object ***after checking that type rules are not violated***

4 Object Wrappers and Autoboxing

Definition: Converting a primitive type to an Object is called Wrapping, these classes are called Wrappers. e.g. *int* → *Integer*

Some feature of Wrappers:

- eight of them: Integer, Long, Float, Double, Short, Byte, Character, Boolean
- First 6 of them are subclass of *number*
- They are *immutable* – cannot change the value once they are constructed

- They are final – cannot subclass them.

Definition: automatically box a primitive type to a Wrapper. e.g.

```
list.add(3)
list.add(Integer.valueOf(3)) //they are the same
```

Wrapper class reference can be *null* but primitive type cannot be a null reference. Hence, it is possible to have a *NullPointerException*

To convert string to an integer:

```
int x = Integer.parseInt(str);
```

5 Methods with a Variable Number of Parameters

It is an application of *autoboxing*.
e.g.

```
public static double max(double... values){
    double largest = Double.NEGATIVE_INFINITY;
    for (double v: values){
        if (v > largest){
            largest = v;
        }
    }
    return largest;
}

double m = max(3.1, 40.4, -5);
```

In this case, compiler passes a new double[]3.1, 40.4, -5 to the max function we defined.

We can say that "varages" methods usually pass an *array* of objects into the argument.

6 Enumeration Classes

Define a Enum type:

```
public enum Size{
    SMALL(``S''), MEDIUM(``M''),
    LARGE(``L''), EXTRA_LARGE(``XL'');
    private String abbreviation;

    private Size(String abbreviation){
        this.abbreviation = abbreviation;
    }
    public String getAbbreviation(){
        return abbreviation;
    }
}
```

Some Notices:

- The constructor of it must be *private* – cannot change it contents
- All self-defined enum type are the subclasses of "*Enum*"
- "*toString*": size.SMALL.toSrting() returns "SMALL"
- "*valueOf*": size s = Enum.valueOf(Size.class, "SMALL") – set s to size.SMALL

7 Reflection

Definition: A program that can analyze the capabilities of classes is called *reflective*.
We can use *Reflection* to:

- Analyze the capabilities of classes at running time
- Inspect object at running time
- Implement generic array manipulation code

- As funtion pointer in C++ (*Method* object)

Come to it later in the seconde reading