Chapter 4

Object-Oriented Programming (1)

References :

[1] รังสิพรรณ มฤคทัต, กระบวนทัศน์ในการเขียนโปรแกรม (บทที่ 4)

[2] Tucker & Noonan, Programming Languages: Principles and Paradigms (Chapter 13)

[3] Dietel & Dietel, Java: How to program (Chapters 8-9)

[4] Oracle, Java Documentation

Chapter Objectives

At the end of this chapter, you should be able to:

- Explain the visibilities of variables/methods in classes
- Compare static and non-static members
- Explain inheritance, overriding, and constructor chain
- Explain polymorphism
- Hand trace Java programs containing basic OOP elements (classes with inheritance & polymorphism)
- Design Java classes for given problems
- Write Java programs in proper OOP style

Object – Oriented Programming

- OO program simulates activities of a collection of interacting machines or objects
  - Objects store data & provide means to manipulate data
  - They communicate with each other by passing messages
- In pure OO language
  - Everything – even primitive data such as int – is object
  - All computation is done by sending a message to object to invoke its methods, and waiting for response (i.e. result)
- Fundamental : encapsulation, inheritance, polymorphism

Imperative *vs.* Object-Oriented

- Imperative program
  - Action-oriented / top-down
  - Object is a passive component, manipulated by operation or function, e.g.

```
result = compute(obj, 100);        // obj as function's argument
```

- OOP
  - Data-oriented / bottom-up
  - Object is an active component, transforming itself, e.g.

```
result = obj.compute(100);         // message to obj = method call
                                   // message content = parameter
                                   // response = returned result
```

Java Object & Class

- Class = type declaration which encapsulates:
  - Member variables
  - Member functions or methods
  - Constructors (no destructor in Java)
  - There may be inner-classes within a class
- **MyClass obj = new MyClass (…)**
  - (1) declare reference variable, (2) create object in heap
  - obj **instanceof** MyClass → true if obj is an instance of MyClass; false otherwise
  - obj.**getClass().getName()** → return class name as string

Encapsulation & Information Hiding

- Encapsulation
  - Wrap logically related variables & methods together
  - Facilitate information hiding, but encapsulation alone does not guarantee proper information hiding
- Information hiding
  - Limit access to details of work or some data ➔ prevent them from being updated/changed unexpectedly
  - Use visibility rules for levels of protection
    - All members (variables and methods) must be assigned visibility levels

Visibility Rules in Java

Accessible to	public	protected	default	private
Same class Same package	✓	✓	✓	✓
Subclass Same package	✓	✓	✓	✗
Other class Same package	✓	✓	✓	✗
Subclass Other package	✓	✓	✗	✗
Other Class Other package	✓	✗	✗	✗

If no visibility is specified ➔ default visibility

Example : visibility

- Folder (i.e. package) 1

```
class A {
      ???  void f()  { … }
}
class B extends A { … }
class C { … }
```

- Folder (i.e. package) 2

```
class D extends A { … }
class E { … }
```

Who can call method f() if

- **???** = private ➔ only methods in A
- **???** = public ➔ everyone (methods in A, B, C, D, E)
- **???** = protected ➔ methods in A, B, C, D
- **???** = default ➔ methods in A, B, C

Java Package

- Hierarchical organization of classes
  - After compilation, we get 1 class file per class
  - Every class must be a member of a package. If no package is specified, it belongs to default package (current working dir)
  - There may be subpackages within a package
- Using a class within a package
  - Full path name : PackageName.ClassName
    JVM uses CLASSPATH to search for this package
  - Import PackageName.* and use class without specifying package name (similar to using namespace in C++)

Some Standard Packages

java.lang	Core language classes (imported automatically), e.g. Object, Math, String, Thread, Throwable
java.io	Stream and file I/O classes
java.util	Utility classes, e.g. Date, Random, ArrayList
java.util.concurrent	Classes to support multithreaded programming, e.g. Semaphore, CyclicBarrier
java.awt javax.swing	Classes for graphics & GUI

Good OOP Practices (1)

- Avoid public member variables. Instead, use setter/getter methods to access private variables
  - setVar(…) assign a value to Var
  - getVar() return Var
- Let object update its own states, leave results or new states inside the object, use getter methods to get results
- Each method should do a single activity & be independent of others. Method params should be few
  - Passing a whole object as argument is better than passing too many individual variables
  - For complicate task, have a public method calling multiple private methods. Hide details in private methods

Lifetime

- Class
  - When program execution starts, class with main method is loaded into code segment memory (no object creation)
  - If there are references to other classes, they will be loaded into the memory
  - All loaded classes remain in the memory until the program terminates
- Object
  - Lifetime starts when object is allocated on heap and its address is held by reference variable
  - Automatic de-allocation, when no reference to object

    E.g. reference var is assigned a new address; exit from current execution block

Member Variables

- Declared inside class but outside methods
- Static or class variable : tied with class
  - Loaded with class into code segment memory & remain there until the program terminates
  - 1 copy of static variable per class
  - Accessed via class or object
  - Typically used to hold constant values
- Non-static or instance variable : tied with object
  - Created when the object is allocated on heap & assigned a default value (false / 0) upon creation
  - 1 copy of non-static variable per object
  - Accessed via object only

Member Methods

- Both static and non-static methods are loaded with class into code segment memory
- Static or class method
  - Accessed via class or object
  - Can use only static vars, or call only static methods
- Non-static or instance method
  - Despite loaded into memory, it is not readily accessible
  - Must allocate an object first & access it via object
  - All objects of the same class share the same copy of non-static method, but each has its own pointer to the method

Method Overloading

- Implement similar tasks using the same method name
  - Overloaded methods are distinguished by their parameters
  - But they cannot be distinguished by return type
  - For example, overloading
    - ✓ int action(String x) AND int action(String x, int y)
    - ✗ int action(String x) AND void action(String x)

- C disallows overloading
- C++ allows both function and operator overloading
- Java allows only method overloading

Usage of Class

- Entity class : simulate real-world object
  - Member vars represent object's attributes or states
  - Member methods represent object's activities that change the object's current states
- Abstract class
  - Object cannot be initialized directly
  - Used as fundamental class from which other classes inherit
- Boundary class
  - Wrap multiple utilities or service functions together
  - Member methods are mostly static
  - Object can be initialized, but it is usually not needed because most members can be accessed via class
  - E.g. Math, Integer, Double

Constructor, Destructor

- Java
  - No destructor – garbage collection works automatically
  - Constructor
    - Overloading is allowed
    - Cannot return anything
    - If a class has no constructor, compiler will add empty constructor automatically ➔ public classname() { }
    - Member vars are initialized with default values
- Client cannot create object from
  - Abstract class
  - Class that has only private constructors (but object can still be created inside the class)

Static block

- Each class has a static block which is executed once the class is loaded. So a static block is executed once per program execution
- Can be thought of as a “class constructor”
- If a class has no static block ➔ an empty block is inserted

```
static {
   // can use only static vars, or call static methods
   ...
}
```

- But it is not a method ∴ cannot take any argument

Good OOP Practices (2)

- Non-static method is better for reusing
  - Can create many objects with their own copies of non-static variables
  - Then, use objects to call non-static methods
- What if we use static methods ?
  - Static methods can only access static methods/variables (in the same class)
  - All objects created from the same class share the same copy of static variables

Incremental development

- If class has already been in use
  - Keep update inside class/method
  - Method's header should remain unchanged
- If we want existing method to have different parameters
  - Use overloading instead of changing old method
- If we want to add more details to existing class
  - Use inheritance instead of changing old class

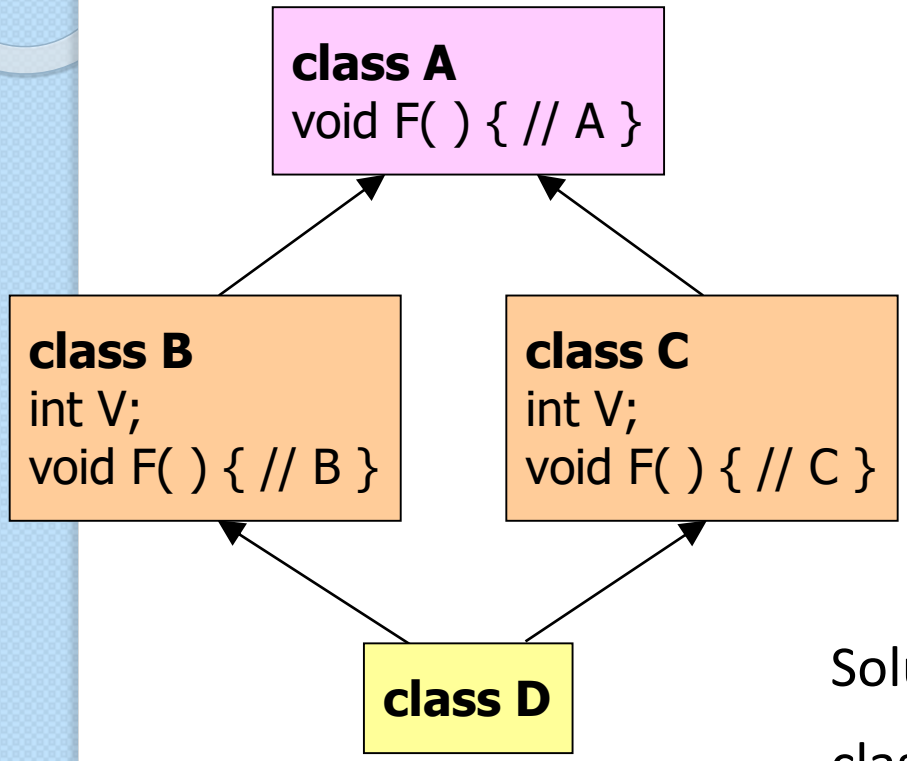
Inheritance

- Make new class by inheriting variables and methods from the old one
  - All components, with all visibility (including private), are inherited. But using them in the new class depends on their visibility rules
  - Exception : constructors are not inherited
- Java Inheritance
  - Multiple inheritance is not allowed
  - Inherited members in subclass have the same visibility as those in superclass

  class <subclass> extends <superclass> { … }

Problems with Multiple Inheritance

- C++



Collision in class D

- Classes B and C have variables of the same name
- Classes B and C both override function F( ) in class A

Solution

class B : virtual public A { … }
class C : virtual public A { … }
class D : public B, public C { … }

Shadowing / hiding

- If subclass has the same variable as superclass (either same or different type), the one in superclass will be shadowed
- Variables in subclass and superclass may have any visibility
- Either of them can be static or non-static

```
class A {
  int x = 10;
  void f()      { … }
}
class B extends A {
  double x = 200.5;          //  shadows x in class A
  void f()      { … }        //  overrides f() in class A
  void g()      {
    x = x * 2;               //  means x in this class
    f();                     //  means f() in this class
  }
}
```

- Overriding has conditions as follows:
  - Method in subclass has the same header (same name, parameters, return type) as the one superclass
  - Subclass's method has at least equal or broader visibility (public > protected > default > private)
  - Static method can only override static method
  - And non-static method can only override non-static one
- Prefix final
  - Final class cannot be inherited
  - Final method cannot be overridden
  - Final variable can be assigned a value only once
    i.e. const variable in C/C++

Class java.lang.Object

- Classes in java.lang are imported automatically
- The root of all classes in Java
  - If a class does not inherit from anyone, compiler will make it inherit from java.lang.Object
- Some basic methods
  - public final native Class getClass()
  - public String toString()

Built-In Reference

- **this** : access members in the current class
- **super** : access members in superclass that are shadowed or overridden
  - Nested references (super.super.method()) is not allowed
- Both can be used to access the following
  - Variable : both static and non-static
  - Method : non-static only
    - Both references cannot be used inside static method
    - To access static method in superclass, use **class.method**
- To avoid confusion, shadowing is discouraged
- But overriding is necessary for polymorphism

Example : this *vs.* super

Top of stack →

```
class C extends B {
   int x = 30;
   void f()    { }
   void gC()  { … }
}
```

```
class B extends A {
   int x = 20;
   void f()    { }
   void gB()  { … }
}
```

```
// base class
class A {
   int x = 10;
   void f()    { }
   void gA()  { … }
}
```

In method gC, if we use:
this.x and this.f() = C's members
super.x and super.f() = B's members

In method gB, if we use:
this.x and this.f() = B's members
super.x and super.f() = A's members

In method gA, if we use:
this.x and this.f() = A's member
super.x and super.f() = members of java.lang.Object which do not exist

Constructor Sequence

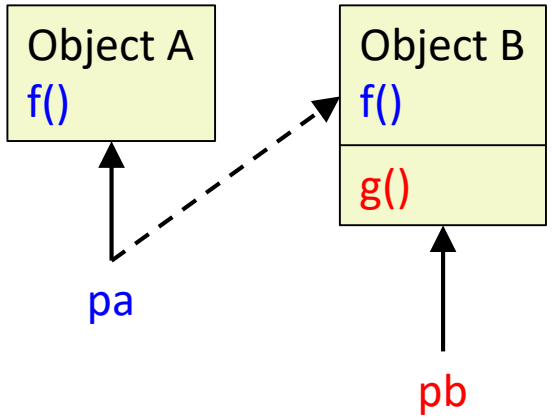
- If a class has multiple constructors, one constructor may call another, using **this(…)** with appropriate parameters
- When an object is initialized
  - Constructors in superclasses along the lineage are called, starting from ancestor's to descendant's
  - If superclass has multiple constructors ➔ subclass should explicitly specify which constructor should be called, i.e. **super(…)**

- Each constructor must have this(…) or super(…)
  - But not both
  - Must be the first instruction in constructor
  - From the current constructor, if we follow chain of this(…), it must eventually reach any super(…)
- If a constructor has neither this(…) nor super(…)
  - Compiler will add **super()** automatically
- So when we write class without inheritance & constructor

```
class A extends java.lang.Object  {    // added by Java
    public A( )     { super( ); }          // added by Java
    …
}
```

References to Parent – Child Objects

- class A { // has f() }
  class B extends A { // has g() }
  A pa = new A();
  B pb = new B();



- Parent's reference (pa) can point to child (B)
  - But it sees only B's members that are inherited from A
  - Allowed pa = pb; pa = new B(); pa.f(); ((B) pa).g();
  - Not allowed pa.g();
- But child's reference (pb) cannot point to parent (A)
  - Not allowed pb = pa; pb = new A();

Reference from java.lang.Object can point to object of any class

Polymorphism

- Late binding of a call to different implementations, depending on type of the calling object
  - All objects' classes must inherit from the same base class
  - All of them override method in base class
  - Method must be called via pointer or reference whose type is declared as that of the base class

```
class Anyshape                          { public void draw() { … } }
class MyCircle      extends Anyshape    { public void draw() { … } }
class MyRectangle   extends Anyshape    { public void draw() { … } }
class MyTriangle    extends Anyshape    { public void draw() { … } }
```

```
// calling polymorphic method via different objects

Anyshape [ ] parent  =  new Anyshape[3];
parent[0]  =  new MyCircle();
parent[1]  =  new MyRectangle();
parent[2]  =  new MyTriangle();

for (int i = 0; i < 3; i++)
    parent[ i ].draw()
```

Actual type of the calling object is known at runtime

Polymorphic method = method that can be applied to any one of related types

Overloading *vs.* Polymorphism

- Overloaded methods
  - Rely on static binding
  - The version of method to be called is known at compile time (from arguments in the call instruction)
- Polymorphic methods
  - Rely on dynamic binding
  - The correct version is known at runtime, depending on the actual type of the calling object