香港浸會大學
HONG KONG BAPTIST UNIVERSITY

COMP2046

Problem Solving Using Object Oriented Approach

Abstract Class

This course is colocated with *COMP2026* Problem Solving Using Object Oriented Programming

Overview

- Abstract Class
- Interface
- Anonymous Class
- Lambda Expression

Class Hierarchy

- Superclass is designed for the purpose of abstraction
- Sometimes the superclass class does not have a reasonable implementation on a method.

```
class Shape {
  private String color;
  void setColor(String c) { color = c; }
  double getArea() {
    return ??;
  }
  void print() {
    System.out.print("This is a " + color);
  }
}
```

```
class Rectangle extends Shape {
  private int width, height;
  @Override
  double getArea() { return width * height;  }
  @Override
  void print() {
    super.print();
    System.out.println(" rectangle!");
  }
}
class Circle extends Shape {
  private int radius;
  @Override
  double getArea() { return PI * radius * radius; }
  @Override
  void print() {
    super.print();
    System.out.println(" circle!");
  }
}
```

The superclass


```
class Shape {
  private int color;
  void setColor(int c) { color = c; }
  double getArea() {
    return ??;
  }
  void print() {
    System.out.print("This is a " + color);
  }

}
```

- What should we put in the ?? 0?
- It is not a really big problem, you would not instantiate a `Shape` object anyway
  - A shape object does not make sense. A shape must be either a Rectangle, a circle, a square...
  - Oh.. but what if someone make a Shape object?

Abstract Class

- An **abstract class** holds the generalized logic of classes.
- An abstract class should never been instantiated. And it is **NOT allowed** to instantiated.
- The reason for having abstract class is that some method of the abstract is not defined in that level (e.g. `getArea()`)
- Java defines an abstract as follows.

```
public abstract class Shape {
  private int color;
  void setColor(int c) { color = c; }
  abstract double getArea();
  void print() {
    System.out.print("This is a " + color);
  }
}
```

Abstract Class


```
public abstract class Shape {
  private int color;
  void setColor(int c) { color = c; }
  public abstract double getArea();
  void print() {
    System.out.print("This is a " + color);
  }
}
```

- The keyword `abstract` appears twice.
- `public abstract class` says this class is abstract.
- `public abstract double getArea();` say the method `getArea()` is abstract.

About Abstract Class

- An abstract class is like other normal class. Follow the same set of rules of constructor, method overloading, method overriding, polymorphism, etc...
- An abstract class can never been instantiated.

```
Shape s = new Shape(); //error!
```

- Unless it is an abstract class, a class cannot have abstract method.
- An abstract class can be inherited as an abstract class or an ordinary class

```
public abstract class RoundShape extends Shape {}
```

- An abstract class cannot be final.

About Abstract Method

- An **abstract method** is a method that has no implementation.
- All non abstract subclass must has an implementation of the abstract method.
  - A subclass that does not have an implementation of the abstract method can only be an abstract class!

```
class Square extends Shape {
  private int width;
  public double getArea() {   return width*width;   }
}
```

More about Abstract class


```
abstract class A {
  abstract void aMethod();
}
abstract class B extends A {
  void bMethod() { }
}
class C extends B {
  void aMethod() { }
}
class D extends C {}
```

- Class `B` does not implement the abstract method, thus, it is abstract
- Class `C` implements the abstract method, thus it is allowed to be declared as non-abstract class.
- Class `D` does not implement on its own. But it has an implementation (from C), thus, it is can be declared as non-abstract. ▶

Multiple inheritance is not allowed

- What is multiple inheritance?
  - e.g. Bat-man, want to be a bat and a man at the same time. Inherit from both Bat class and Man class.
- Why not allowed?
  - Suppose both superclass have the same method `eat`. Do you eat like a bat or eat like a man?
  - Even they don't share the same method, which superclass's constructor to call? Who first?
- But then how can I create something that is a vehicle that can also fly like a bird?

```
class Bird {
  void fly() {}
}
```

```
class Vehicle {
  ...
}
```

```
class Passenger {
  void getOn(Vehicle v)
  {..}
}
```

Interface

- Interface works almost like an abstract class
- It **does not have** any implementation, no constructor, and no field except for `public static final` variables.

```
public interface Flyable {
  public void fly();
}
```

- Syntax of an interface starts with the keyword `interface`.
- **No implementation** should be provided inside an interface^ - Just a `;` after the method name
- All methods are by default `public`.
- All variables are by default `public static final`.

^Another lie. For those who want to uncover it, search: ***interface default java***

Implements an Interface

- We don't `extends` an interface, instead, we `implements` it.

```
class Bird implements Flyable {
  @Override
  public void fly() {...}
}
```

Or interface and inherit at the same time

```
class Aeroplane extends Vehicle implements Flyable {
  @Override
  public void fly() {...}
  @Override
  public void move() {...}
}
```

Interface


```
class Passenger {
  void getOn(Vehicle v)
  {..}
}
```

```
class Radar {
  void detect(Flyable f)
  {..}
}
```

- A passenger is expecting a vehicle to get on. Apparently an `Aeroplane` is a vehicle
- Radar can detect every `Flyable` object, so a Aeroplane of course!

More about Interface

- Interface is a **description** about objects “Trust me, the object can do this!”
- Interface is a **contract** about objects “I promise you that the object can do this!”
- Interface groups (completely different) objects by **what they can do** (even though they could be doing it in a completely different way)
- Suppose we write a method that expects an object that is writable as its parameter
- Interface can make this very flexible

```
public void writingTo (Writable obj) {...}
```

- With Interface, code becomes flexible, general, and we don’t even need to know what class the object belongs to, but only what the object is capable of doing

More about Interface

- Much like two objects are related if they inherit from the same superclass, objects can be related if they implement the same interface (that is, both of them are capable of doing something)

Superclass

- Mainly used to *factor out* **responsibilities** among similar classes.
- Superclass models an “is-a” hierarchy
- For example…
  - Every cat and dog **is-an** animal
  - Every pencil and crayon **is-a** pen

Interface

- ainly used to *factor out* **capability** among very different classes.
- Interface models **can-be** relationships
- For example…
  - All Birds and Aeroplanes **can-be** flyable
  - All Apples and Pizzas **can-be** eatable

Interface

- Interface only declares capabilities that the objects must have – there is no definition of code
- thus no code to re-use! (only declarations)
- purely “contractual” mechanism (in this sense, similar to abstract methods)
- allows the compiler to do error checking!
- Interface is even more abstract than abstract class, because it does not contribute code at all;

An interface, should have no single line of implementation, at least it is true in our course. (see `default` method for exception - *Optional* content)

Implementing Interface

- Classes can extend only one other class
- Classes can implement any number of interfaces
- Classes can extend a superclass and implement interfaces
- For example, Vehicle could implement interfaces which categorize objects that are able to move, hold passengers, be driven, be repaired, etc.

```
class Vehicle implements Movable, Drivable, Repairable {

}
```

- Classes implement an interface must implements all methods of an interface.

Implementing Interface

- What if two interfaces have the same method signature?
- No problem! It refers to the same implementation!

```
interface MusicallyPlayable { void play(); }
interface GamePlayable { void play(); }
class MusicGame implements MusicallyPlayable, GamePlayable {
  void play() { System.out.println("Play a music game!");}
}
...
MusicallyPlayable music = new MusicGame();
GamePlayable game = new MusicGame();
music.play();
game.play();
```

- Can I respond differently? No.

Paying the Bill


```
public interface Payable {
  boolean pay(String payer,
      String payee, int amount);
}
class Bill {
  String shop;
  String customer;
  int amount;
  Bill(String shop, String customer, int amount) {
    this.shop = shop;
    this.customer = customer;
    this.amount = amount;
  }
  void settle(Payable p) {
    System.out.println(shop + ":" + customer + ":" + amount);
    if (p.pay(customer, shop, amount) == true)
      System.out.println("The bill is settled!");
    else
      System.out.println("Not yet settled");
  }
}
```

Paying the Bill


```
public class Cash implements Payable {
  private int total;
  public Cash(int total) {this.total = total;}
  public boolean pay(String payer, String payee, int amount) {
    if (total > amount) {
      total -= amount;
      System.out.printf("%s gives %s $%d in cash.\n", payer, payee, amount);
      System.out.println("There are $" + total + " left.");
      return true;
    }
    return false;
  }

}
```

Paying the Bill


```
Bill bill = new Bill("Electricity", "Kevin Wang", 100);
Cash wallet = new Cash(500);
bill.settle(wallet);
```

```
Electricity:Kevin Wang:100
Kevin Wang gives Electricity $100 in cash.
There are $400 left.
The bill is settled!
```

- A `Bill` needs a `Payable` to settle.
- `Cash` is commonly use and deserves a dedicated class.
- But some `Payable` does not deserve a separate class.

Anonymous class

- An ad-hoc class that fits a niche situation.
- You would probably not use this class anymore.
- Use an **anonymous** class to implement it.
- An anoymous class always implement an **interface** or extends an **abstract class**. Syntax:

```
InterfaceX i = new InterfaceX() {
                    @Override
                    public void abstractMethod() {...}
                };
```

or

```
AbstractClassY a = new AbstractClassY() {
                    @Override
                    public void abstractMethod() {...}
                };
}
```

Anonymous class


```
Bill bill = new Bill("Parking Ticket", "Tattoo Man", 100);
Payable p = new Payable() {
              public boolean pay(String payer, String payee, int amount) {
                System.out.println("Don't bother me!");
                if (amount < 30)
                  return true;
                return false;
              }
            };
bill.settle(p);
```

```
Parking Ticket:Tattoo Man:100
Don't bother me!
Not yet settled
```

Anonymous class


```
Payable p = new Payable() {
              public boolean pay(String payer, String payee, int amount) {
                System.out.println("Don't bother me!");
                if (amount < 30)
                  return true;
                return false;
              }
            };
```

- In Java an anonymous class can be created inline.
- An anonymous class does not have a name.
- `p` is a `Payable` object - constructed by `new Payable();`
- The class definition of this object is after `{}`.

Anonymous class

- An anonymous class is handy
- It is usually used when the object is used only one-off.
- Anonymous class also allows you to **capture** local variable.

```
double discount = 0.8;
Payable p = new Payable() {
                public boolean pay(String payer, String payee, int amount) {
                  System.out.println("Amount after discount: " + amount * discount);
                  return true;
                }
              };
bill.settle(p);
```

- `discount` is used directly inside the anonymous class without parameter passing!

A more realistic example


```
JButton myButton = new JButton("Click me!");
myButton.addActionListener(
  new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      System.out.println("The button is clicked");
    }
  }
)
```

- JButton is a Java GUI component.
- You register an ActionListener to the button. When there is any action done by the user, it **calls back** the ActionListener.
- Each button has different behavior, the ActionListener is unlikely to be repeated.
- Writing an anonymous class seems more economic.

More on anonymous class

- Anonymous class does not have a constructor
- Anonymous class does not have static non-final variable
- Anonymous class cannot be further inherited
- Therefore Anonymous class cannot be abstract
- For more details, please read Oracle Java Tutorial.

Lambda Expression

- Anonymous class is saving the code length.
- Java says we can do better!
- **Lambda expression** can be used to create an anonymous class for **an interface with only one method**.
- For reducing the length of anonymous class, keeping the only necessary.

Lambda Expression

- Which parts are redundant?

```
Payable p = new Payable() {
              public boolean pay(String payer, String payee, int amount) {
                System.out.println("Amount after discount: " + amount * discount);
                return true;
              }
            };
```

- `new Payable()` - you are creating an Payable object, can be inferred from left hand side
- `public boolean pay` - there is only one method, of course you are coding for it

Lambda Expression


```
Payable p = (String payer, String payee, int amount) -> {
                System.out.println("Amount after discount: " + amount * discount);
                return true;
              };
```

Or even shorter, omitting the type of the variables.

```
Payable p = (payer, payee, amount) -> {
                System.out.println("Amount after discount: " + amount * discount);
                return true;
              };
```

Lambda Expression Syntax

A lambda expression consists of the following:

1. A comma-separated list of formal parameters enclosed in parentheses.

```
(String payer, String payee, int amount)
```

- You can omit the data type of the parameters in a lambda expression.
- In addition, you can omit the parentheses if there is only one parameter.

2. The arrow token, `->`
3. A body, which consists of **a single expression** or **a statement block**.

```
wallet > amount && !payee.equals("Gangster")
//or
{
  System.out.println("Amount after discount: " + amount * discount);
  return true;
}
```

Lambda Expression Syntax

- A return statement is not an expression;
- In a lambda expression, you must enclose statements in braces (`{}`).
- However, you do not have to enclose a `void` method invocation in braces. For example, the following is a valid lambda expression:

```
email -> System.out.println(email)
```

- Note that a lambda expression looks a lot like a method declaration; you can consider lambda expressions as anonymous methods—methods without a name

Lambda Expression - Payable


```
Bill bill = new Bill("ABC Telecomm", "Kevin Wang", 100);
bill.settle( (a, b, c) -> {
  System.out.println(b + " pays " + a + " $" + c + " by QR code");
  return true;
});
```

Explanation

- `(a, b, c)` - three parameters for the method `pay` in the anonymous `Payable`
- `->` - syntax for lambda
- `{...}` - the block of statement executed in the method `pay`
- In `bill.settle()` it will invoke `p.pay(customer, shop, amount)`, the lambda written here decide **what to do** when this method is invoked.

Example - Lab 8

- We rewrite the entire lab 8 using ArrayList.
- The main problem is how to perform sorting
- List provides a method Sort, but you need to tell Sort how to do comparison!
- How would the *sorter* know contact A should be placed in front of contact B

```
void sort(Comparator<T> c)
```

e.g.

```
contactlist.sort(contactListCompartorObject);
```

Comparator

- Comparator requires classes to implement **one abstract method** in this interface.

int	**compare**(T o1, T o2) Compares its two arguments for order.

- You can write anonymous class

```
Comparator<Contact> c = new Comparator<>() {
                          int compare(Contact o1, Contact o2) {
                            return o1.getName().compareTo(o2.getName());
                          }
                        }
contactlist.sort(c);
```

Comparator


```
int                    compare(T o1, T o2)
                       Compares its two arguments for order.
```

- Or you can write a lambda

```
contactlist.sort( (o1, o2) -> o1.getName().compareTo(o2.getName()));
```

- `(o1, o2)` - parameters for the method `compare`
- no need return because there is only one single expression
- no need {} because there is only one single expression

Tips

- The syntax is weird, the concepts is tricky
- Suggest doing the following:
  - Read carefully the examples that I have created for you, all of them
  - Test the code on your own and try to rewrite them, even just type it again
  - Create some simple abstract class like `Animal`, interface like `Eatable` and see how they should be put together

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

End of All Examable

Errata

Page 15, the words capability and responsibility are swapped.