NUS National University of Singapore

School of Computing

CS2040 Data Structures and Algorithms Lecture Note #1

Introduction to Java Part 2

4.6 Essential OOP Concepts for CS2040

What makes Java object-oriented?

OOP in java→ Classes/objects (1)

- OOP or object oriented programming solves computational problem by modeling entities of the problems as classes/objects
- Class == blueprint/mold
- Object == actual entity instantiated from a class

Encapsulation in Java Class

- Encapsulation
  - Bundling data and associated functionalities
  - Hide internal details and restricting access

Data in a java Class (1)

- Data in a java Class is represented by attributes/variables that you declare outside of the methods but within the class
  - Instance Attribute → Each created instance/object of a class has it's own copy of the instance attribute
  - Class Attribute → Associated with the class, and is shared by all objects created from the class
    - Usually used to denote constants or in some rare cases global variables (e.g keeping a count of the number of objects created for a class)

Data in a java Class (2)

- Declaring Attributes in Java
  - Instance Attribute → Similar to declaring any variable but it should be outside of any method in the class, e.g

```
class Circle {
  public int radius;

  …
}
```

  - Class Attribute → Add the keyword static in front of the variable, e.g

```
class Circle {
  public static int NUM_CIRCLE=0;

  …
}
```

  - **for the purpose of this course prefix all attribute declaration/method declaration with public*

Functionality in java Class (1)

- Functionality in a java Class is represented by methods
  - Instance methods → methods that can be called via an object of the associated class e.g nextInt() method of scanner
  - Class methods → methods that can be called via the class itself (there is no need to create an object to call it)

Note: *Variables that you declare in methods are not attributes but local variables*

Functionality in java Class (2)

- Declaring methods in Java
  - Instance methods → similar to how functions are declared in other languages, except here all return values, and method parameters must have the type declared (if no return value it must be declared as void) e.g

```
class Circle {
   …

   public void updateRadius(int rad){
      radius = rad;
   }
}
```

  - Class methods → like class attributes, prefix the method declaration with keyword **`static`** like the gcd method in the FractionV4 class
    - Note that you cannot use instance attributes in class methods!

Simple design guidelines (1)

- Instance attribute → Something that is associated with a specific object e.g in a Circle class, radius should be an instance attribute

- Class attribute → Usually represent a constant value that is shared by all objects of the class e.g in a Circle class, PI should be a class attribute, or some attribute shared by all objects, e.g NUM_CIRCLE which represent total circle objects created

Simple design guidelines (2)

- Instance method → Method which needs to operate on the instance attributes in an object

- Class method → Method which does/should not operate on any instance attributes in an object (eg gcd method in our Fraction class)

- For this course we are not so concerned about design as long as the code works ….
  (don't tell your CS2030 lecturer…)

Fraction class → OOP design (1)

- Make the Fraction class actually represent a fraction (1)
  - Make the numerator, denominator as instance attributes
  - Make instance methods to access/modify the numerator/denominator (also know as accessor/mutator methods)
  - Make **gcd** a class method (not really associated with a specific fraction)

Fraction class → OOP design (2)

- Make the Fraction class actually represent a fraction (2)
  - The addition operation can be made either a class or instance method
    - Class method if you do not modify the attributes of the object but merely return a new Fraction that is the result of the addition (pass both fractions into add method)
    - Instance method if you modify the attributes of an object after perform addition with another Fraction object ← we will implement this one

FractionOOPV1 class (1)

```
class FractionOOPV1 {
  public int num, denom;

  public int getNum() { return num; }
  public int getDenom() { return denom; }
  public void setNum(int iNum) {num = iNum;}
  public void setDenom(int iDenom) {denom = iDenom;}
  public static int gcd(int e, int f) {
    int rem;

    while (f > 0) {
      rem = e%f;
      e = f;
      f = rem;
    }
    return e;
  }
  // instance method add -> takes in another fraction add to this fraction
  //and modify it
  public void add(FractionOOPV1 iFrac) {
    num = num*iFrac.getDenom()+denom*iFrac.getNum();
    denom = denom*iFrac.getDenom();
    int divisor = gcd(num,denom);
    num /= divisor;
    denom /= divisor;
  }
}
```

Instance attributes

Instance methods

FractionOOPV1.java

FractionOOPV1 class (2)

- Note there is no more public in front of the class, as FractionOOPV1 no longer contains the main method
- FractionOOPV1 is now known as a service class

Overloading methods

- Methods in a class can be overloaded
  - Having multiple methods with the same name but different parameters
  - The correct method will be called based on what arguments are supplied for the parameters

- You see method overloading a lot in the Java API as shown earlier

- e.g in our FractionOOPV1 class we can have an overloaded add method which simply takes in 2 arguments, the numerator and denominator

FractionOOPV1 class (3)

FractionOOPV1.java

```
class FractionOOPV1 {
  …

  //instance method add -> takes in another fraction add to this fraction
  //and modify it
  public void add(FractionOOPV1 iFrac) {
    num = num*iFrac.getDenom()+denom*iFrac.getNum();
    denom = denom*iFrac.getDenom();
    int divisor = gcd(num,denom);
    num /= divisor;
    denom /= divisor;
  }

  //overloaded add method -> takes in a numerator and denominator instead of
  //a fraction object
  public void add(int iNum, int iDenom) {
    num = num*iDenom+denom*iNum;
    denom = denom*iDenom;
    int divisor = gcd(num,denom);
    num /= divisor;
    denom /= divisor;
  }
}
```

However we still have no way to create a Fraction object !

Constructor

- To instantiate/create an object of a class, the class must provide a constructor (it is basically a special method)
- Each class has one or more constructors
  - Default constructor has no parameter and is automatically generated by compiler if class designer does not provide any constructor.
  - Non-default constructors are added by class designer
  - Constructors can be overloaded
- Main use of providing your own custom constructor is to initialize the attributes of the object properly

FractionOOPV1 Constructor

```
class FractionOOPV1 {
  public int num, denom;

  public FractionOOPV1(int iNum, int iDenom) {
    num = iNum;
    denom = iDenom;
  }

  …

}
```

FractionOOPV1.java

- A Java constructor has no return type and it must be the same as the class

Fraction class → OOP design (3)

- FractionOOPV1 is now called a service class (like the classes in the Java API)
- It can be used by anyone who has access to your Fraction class whenever they need to represent fractions
- Now in order to test out or make use of our FractionOOPV1 class we have to create a client class
  - This is the public class with the main method

TestFractionOOPV1 client class V1

- A client class we can create is as follows

```
public class TestFractionOOPV1 {

  public static void main(String args[]) {
    FractionOOPV1 f1 = new FractionOOPV1(1,2);
    FractionOOPV1 f2 = new FractionOOPV1(3,4);
    f1.add(f2);
    System.out.println(f1.getNum()+"/"+f1.getDenom());
    f1.add(4,5);
    System.out.println(f1.getNum()+"/"+f1.getDenom());
  }
}
```

TestFractionOOPV1.java

- Actually the client class should not worry about formatting of the fraction to be printed,as it should be taken care of by the FractionOOPV1 class itself

Overriding methods

- All classes in Java “inherit” from the Object class
- One method inherited is the toString method which returns a string representation of the object for output/printing purposes
- However in order to be useful you will have to override the method (provide your own implementation to format your object as a string nicely)

Overriding the toString method

FractionOOPV1.java

```
class FractionOOPV1 {

  …

  public String toString() {
    return num+"/"+denom;
  }
}
```

- Note that the method header must be the same as the method header in Object class (check the Java API)

TestFractionOOPV1 client class V2

- The client class can now be updated as follows

```
public class TestFractionOOPV1 {

  public static void main(String args[]) {
    FractionOOPV1 f1 = new FractionOOPV1(1,2);
    FractionOOPV1 f2 = new FractionOOPV1(3,4);
    f1.add(f2);
    System.out.println(f1);
    f1.add(4,5);
    System.out.println(f1);
  }
}
```

TestFractionOOPV1.java

Finishing touches

- Usually we don't just perform 1 addition operation
- We can be given a list of addition operations
- The input would be then be
  - First the number of addition operations N
  - Followed by N addition operations (given N number of fraction pairs to add)

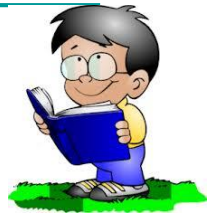
Finished TestFractionOOPV1

```
public class TestFractionOOPV1 {

  public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    int numAdd = sc.nextInt();

    for (int i=0; i < numAdd; i++) {
      FractionOOPV1 f1 = new FractionOOPV1(sc.nextInt(),sc.nextInt());
      FractionOOPV1 f2 = new FractionOOPV1(sc.nextInt(),sc.nextInt());
      f1.add(f2);
      System.out.println(f1);
    }
  }
}
```

TestFractionOOPV1.java

- Note that FractionOOPV1 and TestFractionOOPV1 need not be in a separate file each, in fact we want you to write all your service and client classes in one file (file name will be the client class name)

Extra: "this" reference (1)



- What if the parameter of a method (or a local variable in a method) has the same name as an instance attribute?

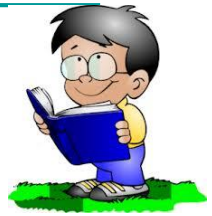
```
public void setNum(int num) {
   num = num;
}

public void setDenom(int denom) {
   denom = denom;
}
```

These methods will not work, because **num** and **denom** here refer to the parameters, not the instance attributes (overshadowing).

The original code:

```
public void setNum(int iNum) {
   num = iNum;
}
public void setRadius(int iDenom) {
   denom = iDenom;
}
```

Extra: "this" reference (2)



- The "**this**" reference is used to solve the problem in the preceding example where parameter name is identical to attribute name

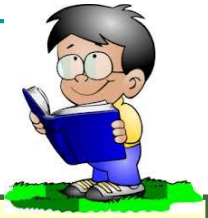
```
public void setNum(int num) {
   num = num;
}

public void setDenom(int denom) {
   denom = denom;
}
```

```
public void setNum(int num) {
   this.num = num;
}

public void setDenom(int denom) {
   this.denom = denom;
}
```

attributes

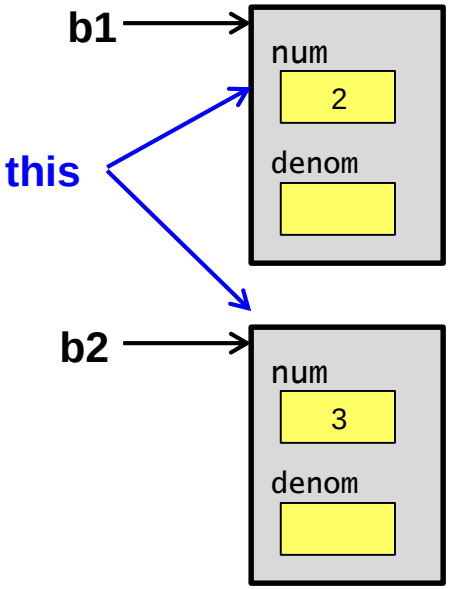
parameters

"this" reference (3)



```
// b1 & b2 are FractionOOPV1 objects
b1.setNum(2);
b2.setNum(3);
```

- A common confusion:
  - How does the method "know" which "object" it is communicating with? (There could be many objects created from that class.)
- Whenever a method is called,
  - a reference to the calling object is set automatically
  - Named **"this"** in Java, meaning "*this particular object*"
- All attributes/methods are then accessed <u>implicitly</u> through this reference (only need to explicitly use this for ambiguous cases)



End of file