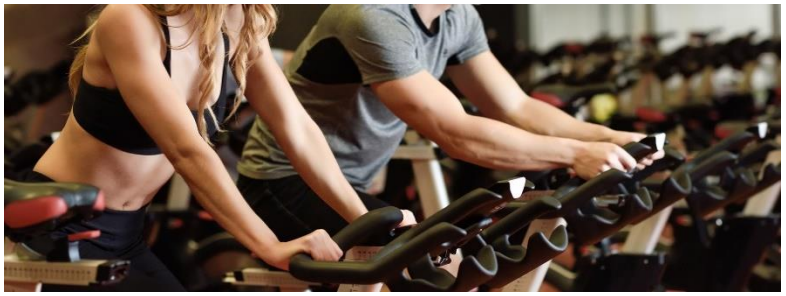
IECS273/274 [1111 3670/3671]
物件導向設計與實習

04#1 OOP Fundamentals
Encapsulation 1/2



01 OOP

Object Oriented Technology

- A “New” Paradigm!
  - What is Object-Oriented?
    - ✓ *paradigm*
  - Why is Object-Oriented?
    - ✓ *software crisis*
  - Advantage of Object-Oriented?
    - ✓ problem domain vs. computer domain
- *Everything is Object!*
  - **Encapsulation**
  - **Inheritance**
  - **Polymorphism**

Object Oriented Programming in Java

- Object-Oriented Programming (OOP) uses **classes** to encapsulate data (attributes) and methods (behavior).
  - Object-oriented programming data and functions are closely related.
  - Information hiding - implementation details are hidden in the class itself.
- Classes are central to Java. Java programming unit: **class**.
  - A **class** is a special kind of programmer-defined type, and variables can be declared of a class type. A class is like a blueprint -- **reusable**.
  - A value of a class type is called an **object** or an **instance** of the class. An object is the result of instantiating (constructing, created) from a class. For example, house is an instance of "blueprint class".
  - In C language, programmers focus on **function development**, while in Java language, programmers focus more on **class design**.

02

Classes in Java

Class

- A class defines the properties and behaviors for objects.
- A class definition specifies the **data items** and **methods** that all of its objects will have.
- These data items and methods are sometimes called members of the object. *Data items are called fields or instance variables.*


```
class 類別名稱 {
      public:
      資料成員 / 成員函式
      .....
      protected:
      資料成員 / 成員函式
      .....
      private:
      資料成員 / 成員函式
      .....
};
```

Data Items (Fields or Instance Variables)

- Instance variables can be defined as in the following two examples
  - Note the public modifier (for now):

```
public double instanceVar1;
public String instanceVar2;
```

- In order to refer to a particular instance variable, preface it with its object name as follows:

```
objectName.instanceVar1
objectName.instanceVar2
```

Methods

- Method definitions are divided into two parts: a heading and a method body:

```
public void Method()                          == Heading
{
        // code to perform some action        == Body
        // and/or compute a value
}
```

- Methods are invoked using the name of the calling object and the method name as follows:

```
classVar.myMethod();
```

- Invoking a method is equivalent to executing the method body
- There are two kinds of methods:
  - *Methods that perform an action : this type of method does not return a value, and is called a **void** method*
  - *Methods that compute and return a value*

Class Encapsulation

- The content of a class contains data and methods.
  - Java programming language writes data and methods together in **class**es.
  - **Encapsulation** means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details.

```
struct Student {
  char name[20];
  char ID[12];
  struct birthdate {
    int year;
    int month;
    int day;
  }
  int score[6]; // assume 6 courses
};
struct Student stu;
```

```
float average(Student S) {
  int sum = 0;
  int i;

  for (i=0; i<6; i++) sum+=S.score[i];
  return sum / 6.0;
}
```

Class Encapsulation

```
class Circle {
        // The radius of this circle
        double r = 1;
        // Construct a circle object
        Circle() {
        }
        // Construct a circle object
        Circle(double newRadius) {
                r = newRadius;
        }
        // Return the area of this circle
        double getArea() {
                return Math.PI * r * r;
        }
        // Return the perimeter of this circle
        double getPerimeter() {
                return 2 * Math.PI * r ;
        }
        // Set a new radius for this circle
        void setRadius(double newRadius) {
                 r = newRadius;
        }
}
```

ClassDemo.java

```
public class Demo {
    public static void main(String[] args) {

        // Create a rectangle1 with (height, width) = (1,1)
        Rectangle rectangle1 = new Rectangle();
        System.out.println("The area of the rectangle (height, width) = ("
                + rectangle1.height + " , " + rectangle1.width + ") is " +
                rectangle1.getArea());

        // Create a rectangle2 with (height, width) = (5,6)
        Rectangle rectangle2 = new Rectangle(5,6);
        System.out.println("The area of the rectangle (height, width) = ("
                + rectangle2.height + " , " + rectangle2.width + ") is " +
                rectangle2.getArea());

        // Modify rectangle1 with (height, width) = (9,20)
        rectangle1.setValues(9,20);
        System.out.println("The area of the rectangle (height, width) = ("
                + rectangle1.height + " , " + rectangle1.width + ") is " +
        rectangle1.getArea());
    }
}
```

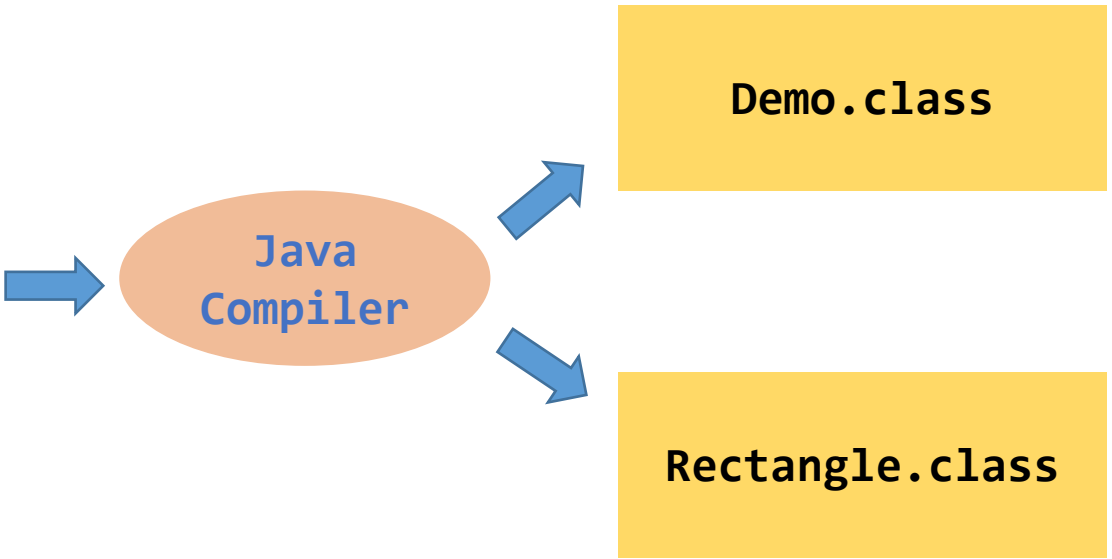
ClassDemo.java

```java
//Define the rectangle class with two constructors
class Rectangle {
        // The height and width of this rectangle
        double height = 1;
        double width = 1;
        Rectangle() {                   // Construct a rectangle object
                System.out.println("Constructor 1");
        }
        Rectangle(double newHeight, double newWidth) {
                System.out.println("Constructor 2");
                height = newHeight;
                width = newWidth;
        }
        double getArea() {              // Return the area of this rectangle
                return height * width;
        }
        double getPerimeter() {         // Return the perimeter of this rectangle
                return 2 * (height + width);
        }
        void setValues(double newHeight, double newWidth) { // Set new height
                System.out.println("Set New Values");
                height = newHeight;
                width = newWidth;
        }
}
```

Demo.class

```
Constructor 1
The area of the rectangle (height, width) = (1.0 , 1.0) is 1.0
Constructor 2
The area of the rectangle (height, width) = (5.0 , 6.0) is 30.0
Set New Values
The area of the rectangle (height, width) = (9.0 , 20.0) is 180.0
```

```
//File: ClassDemo.java

public class Demo {
        ……
}

class Rectangle {
        ……
}
```



Note: Each class in the source code file is compiled into a **.class** file.

Class Encapsulation

- Java programming language uses **class**es to encapsulate **data members** and **member functions (methods)**.
- A .java file can only have one class.
- Java programming language can put multiple classes in a **package**.
- Java language classes have four accessibility modifiers:
  - **public**: accessible by any class or package.
  - **protected**: used in the superclass/subclass relationship, access rights are only within this class and subclasses and this package.
  - ***default***: no modifiers, access permissions are internal to this class and the same package.
  - **private**: access permissions are only within this class.

	this class	**this package**	**subclass**	**external package**
public	√	√	√	√
protected	√	√	√	X
default	√	√	X	X
private	√	X	X	X

Java Constructors

- Every Java class has a **default constructor**. A constructor is a special kind of method that is designed to initialize the instance variables for an object.
  - The default constructor has no parameters.
  - The default constructor of primitive types, such as **byte**, **char**, **short**, **int**, **long**, **float**, and **double**, sets 0 as the default value and **boolean** to **false**. Other **reference types** such as **arrays**, self-defined **classes** sets **null** as the default value.
- Java classes can also define their own constructor (**user-defined constructor**).
  - The name of the user-defined constructor must be the same as the name of the class.
  - User-defined constructors can take parameters.
  - Java supports **overloading**, that is, a class can have *multiple user-defined constructors*, but *the number or types of parameters must be different*.
  - Java supports **garbage collection**, therefore it does have destructor.

Java Constructors

- Example: Student.java defines a class public class Student, which has four constructors：

```
public Student()                        // Default constructor, no parameters
public Student(String str, int n)       // Overloaded constructor, 2 parameters student name and age
public Student(String str)              // Overloaded constructor, 1 parameter student name
public Student(int n)                   // Overloaded constructor, 1 parameter student age
```

  - The four constructors are **overloading**, and the constructor names are the same as the class name, but the number or types of parameters are different.
  - Other methods in a class can also be overloading. Overloading is when two or more methods in the same class have the same method name.
  - To be valid, any two definitions of the method name must have different **signatures**.

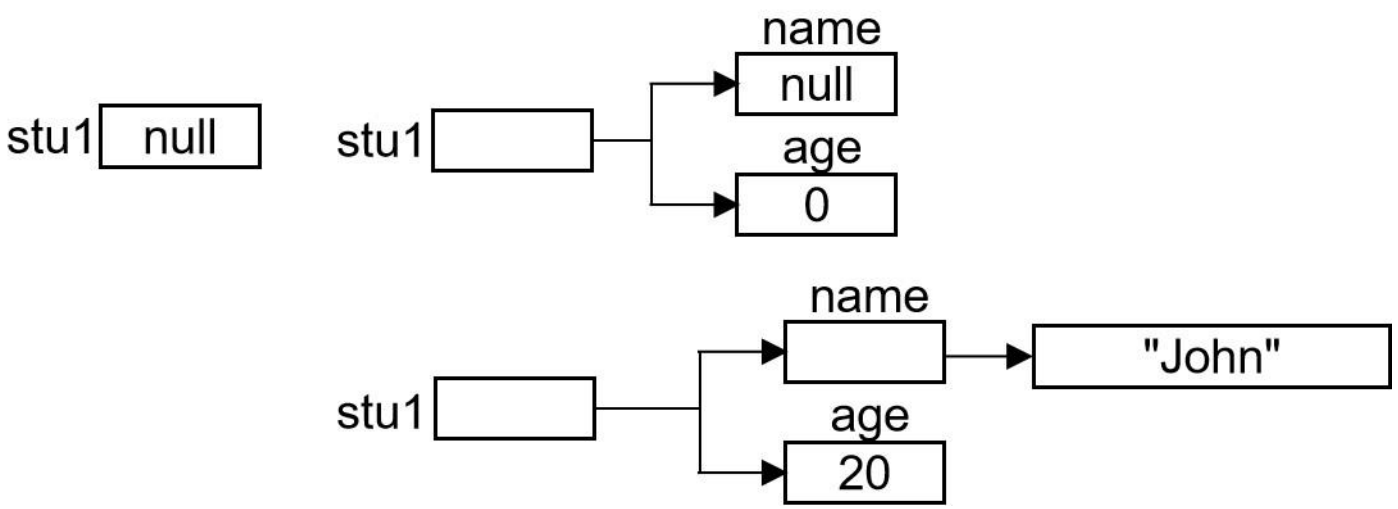
Constructor & Destructor

Circle	類別名稱
-x,y:int	圓心
-radius:int	半徑
+Circle()	建構函式
+setX(int,int):void	設定圓心座標X
+set(int,int):void	設定圓心座標Y
+getX():int	取得圓心座標X
+getY():int	取得圓心座標Y
+setRadius(int):void	設定圓的半徑
+getRadius():int	取得圓的半徑
+getArea():double	取得圓的面積
+getPerimeter():double	取得圓的周長
+~Circle()	解構函式

Object Creation and Use

- Classes in the Java are types, which can be used for **variable declarations**; then, **object creation.**
- Example: Create a Student object

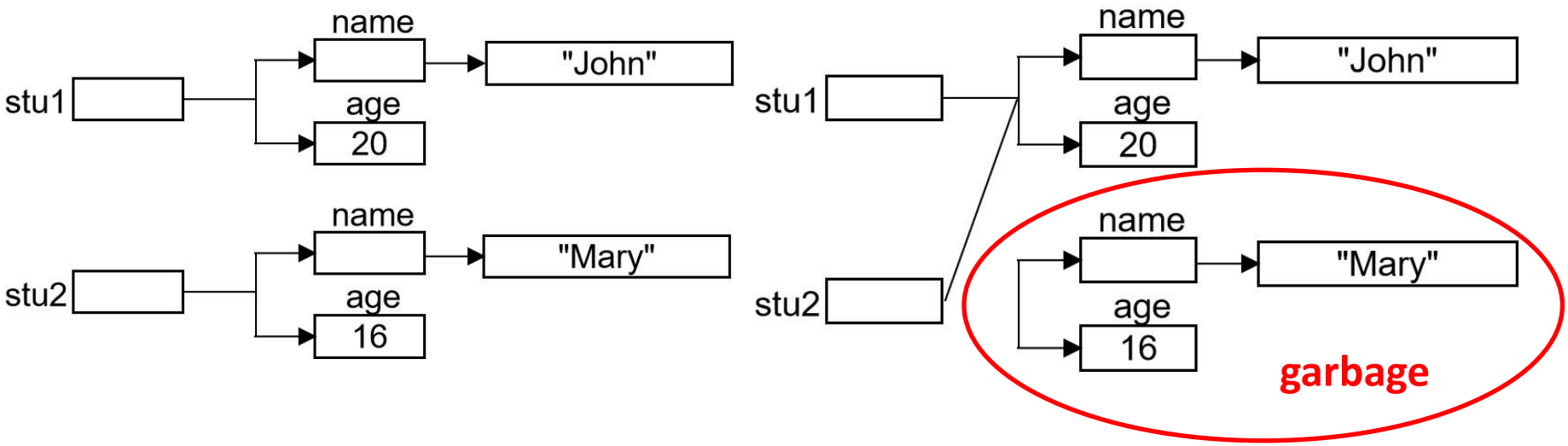
```
Student stu1;                          // Only create memory for object pointer of null value.
stu1 = new Student();                  // Create memory for object members,
                                       // stu1.name is a null pointer, stu1.age is 0.
stu1 = new Student("John", 20);        // Create memory for object members, as a string array
                                       // and an integer value.
```



Object Creation and Use

- Example: Create two Student objects

```
stu1 = new Student("John", 20);      // Create memory for object members, as a string array
                                     // and an integer value.
stu2 = new Student("Mary", 16);      // Create the second object
stu2 = stu1;                         // Two variables pointing to the same object.
```



Methods Definition

- The body of both types of methods contains a list of declarations and statements enclosed in a pair of braces

```
public <void or typeReturned> myMethod()
{
        // declarations
        // statements
}
```

- An invocation of a void method is simply a statement:

```
objectName.methodName();
```

- A method that returns a value can also perform an **action**. An invocation of a method that returns a value can be used as an expression anyplace that a value of the **TypeReturned** can be used:

```
TypeReturned tRVariable;
tRVariable = objectName.methodName();
```

More Methods

- A program in Java is just a class that has a **main** method
  - When you give a command to run a Java program, the run-time system invokes the method main
  - Note that main is a void method, as indicated by its heading:

```
public static void main(String[] args)
```

- The body of a method that returns a value must also contain one or more **return** statements
  - A return statement specifies the value returned and ends the method invocation:

```
return Expression;
```

More Methods

- A **parameter list** provides a description of the data required by a method. Some methods need to receive additional data via a list of parameters in order to perform their work. It indicates the number and types of data pieces needed, the order in which they must be given, and the local name for these pieces as used in the method.

```
public double methodName(int n1, double n2)
```

- When a method is invoked, the appropriate values must be passed to the method in the form of **arguments**.

```
int a=1;

double b=2.3;

double result = methodName(a, b);
```

  - The value of each argument (primitive type) is plugged into the corresponding method parameter.
  - This method of plugging in arguments is known as the **call-by-value mechanism.**

Get and Set Methods

- In order to achieve the purpose of **information hiding,** the access rights of data members are often set to *private* in the class such that they are only accessed by the program code in this class. If there are other classes to modify or use private data members, they must go through the **public set and get methods.**
- Example: Class **public class** Student.
  - It defines **two private data members.**
  - The access rights of private data members are *limited* to the Student class. Outside Student class, you need to modify or use these two private data members through the set and get methods.

```
private String name;                    // student name
private int age;                        // student age

public void setName(String str)         // Set student's name, string parameter
public void setAge(int n)               // Set student's age, integer parameter
public String getName()                 // Get student's name, return string
public int getAge()                     // Get the student's age, return an integer
```

Methods Overloading

- **Overloading** is when two or more methods in the same class have the same method name. To be valid, any two definitions of the method name must have different **signatures**.
  - The signature of a method only includes the method name and its parameter types.
  - Differing signatures must have different numbers and/or types of parameters.
  - The signature does not include the returned type.
- If Java cannot find a method signature that exactly matches a method invocation, it will try to use **automatic type conversion**.
  - Ambiguous method invocations will produce an error in Java.

this – Self Reference

- Sometimes in a class you need to refer to an object of itself, Java uses **this** as a **self-reference object**.
- Example: methods setName and setAge change the values of data members name and age of **class** Student to the parameter values, respectively:

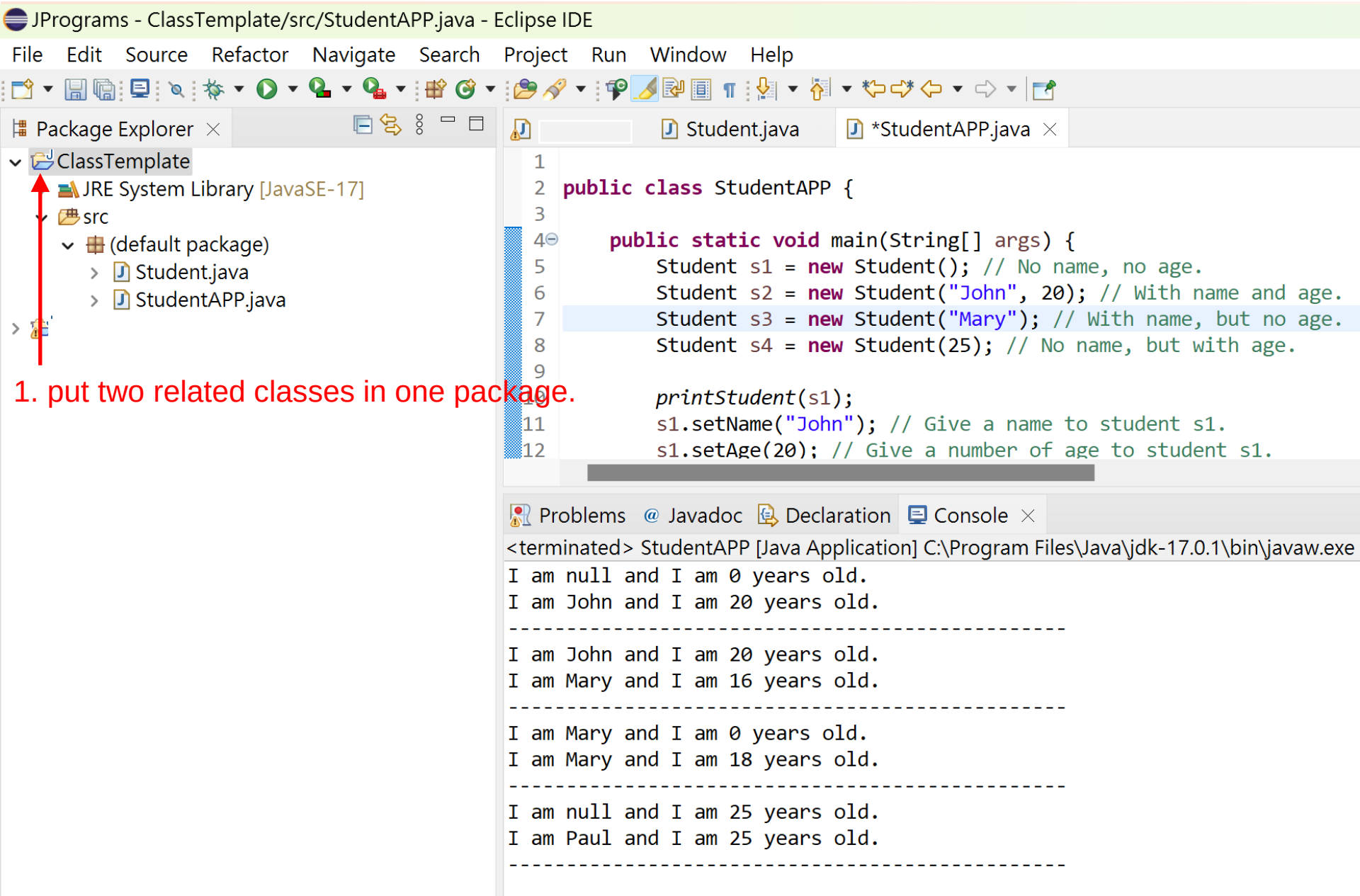
Both are parameter name.

```
// Set student's name.
public void setName(String name) {
  name = new String(name); }

// Set student's age.
public void setAge(int age) {
  age = age; }
```

Error!

```
// Set student's name.
public void setName(String name) {
  this.name = new String(name); }

// Set student's age.
public void setAge(int age) {
  this.age = age; }
```

Use **this** to differentiate class member names and parameter name of methods.


JPrograms - ClassTemplate/src/StudentAPP.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
ClassTemplate
JRE System Library [JavaSE-17]
src
(default package)
Student.java
StudentAPP.java
1. put two related classes in one package.
Student.java
*StudentAPP.java
1
2 public class StudentAPP {
3
4 public static void main(String[] args) {
5 Student s1 = new Student(); // No name, no age.
6 Student s2 = new Student("John", 20); // With name and age.
7 Student s3 = new Student("Mary"); // With name, but no age.
8 Student s4 = new Student(25); // No name, but with age.
9
printStudent(s1);
11 s1.setName("John"); // Give a name to student s1.
12 s1.setAge(20); // Give a number of age to student s1.
Problems Javadoc Declaration Console
<terminated> StudentAPP [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\javaw.exe
I am null and I am 0 years old.
I am John and I am 20 years old.
I am John and I am 20 years old.
I am Mary and I am 16 years old.
I am Mary and I am 0 years old.
I am Mary and I am 18 years old.
I am null and I am 25 years old.
I am Paul and I am 25 years old.

Student.java

2.Use **package** to put two related classes in one package.

```
package ClassTemplate;
public class Student {
  private String name;
  private int age;
  // There is a default constructor with no parameters.
  public Student() { }
  // An overloaded constructor with name and age.
  public Student(String str, int n) {
    name = new String(str);
    age = n; }
  // An overloaded constructor with name only.
  public Student (String str) {
    name = new String(str); }
  //An overloaded constructor with age only.
  public Student(int n) {
    age = n; }
  // Set student's name.
  public void setName(String str) {
    name = new String(str); }
  // Set student's age.
  public void setAge(int n) {
    age = n; }
  // Get student's name.
  public String getName() {
    return name; }
  // get student's age.
  public int getAge() {
    return age; }
}
```

StudentAPP.java

2.Use **package** to put two related classes in one package.

```
package ClassTemplate;
public class StudentAPP {
    public static void main(String[] args) {
        Student s1 = new Student(); // No name, no age.
        Student s2 = new Student("John", 20); // With name and age.
        Student s3 = new Student("Mary"); // With name, but no age.
        Student s4 = new Student(25); // No name, but with age.
        printStudent(s1);
        s1.setName("John"); // Give a name to student s1.
        s1.setAge(20); // Give a number of age to student s1.
        printStudent(s1);
        System.out.println("-----------------------------------------");
        printStudent(s2);
        s2.setName("Mary"); // Change the student name of S2.
        s2.setAge(16); // Change the student age to 16.
        printStudent(s2);
        System.out.println("-----------------------------------------");
        printStudent(s3);
        s3.setAge(s1.getAge()-2); // Set age as two years younger than student s1.
        printStudent(s3);
        System.out.println("-----------------------------------------");
        printStudent(s4);
        s4.setName("Paul"); // Set name for student s4.
        printStudent(s4);
        System.out.println("-----------------------------------------");
    }
    // Print the student's name and age.
    public static void printStudent(Student stu) {
        System.out.println("I am " + stu.getName() + " and I am " + stu.getAge() +
                           " years old.");
    }
}
```

StudentAPP.class

```
I am null and I am 0 years old.
I am John and I am 20 years old.
-----------------------------------------------
I am John and I am 20 years old.
I am Mary and I am 16 years old.
-----------------------------------------------
I am Mary and I am 0 years old.
I am Mary and I am 18 years old.
-----------------------------------------------
I am null and I am 25 years old.
I am Paul and I am 25 years old.
-----------------------------------------------
```

03 Variable & Reference

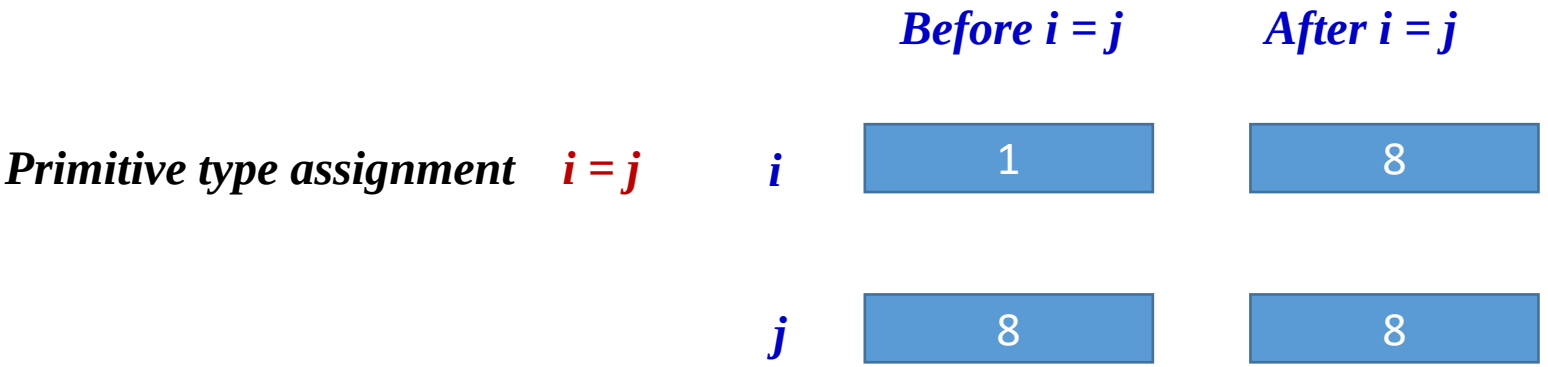
Primitive and Class Type Variables

- Every **variable** is implemented as *a location in computer memory*.
  - When the variable is a **primitive type**, the value of the variable is stored in the memory location assigned to the variable.
    - ✓ Each primitive type always require the same amount of memory to store its values.
  - When the variable is a **class type**, only the memory address (or reference) where its object is located is stored in the memory location assigned to the variable.
    - ✓ The object named by the variable is stored in some other location in memory.
    - ✓ The object, whose address is stored in the variable, can be of any size.
    - ✓ Like primitives, the value of a class variable is a fixed size
    - ✓ Unlike primitives, the value of a class variable is a memory address or reference.

Primitive Type Variables

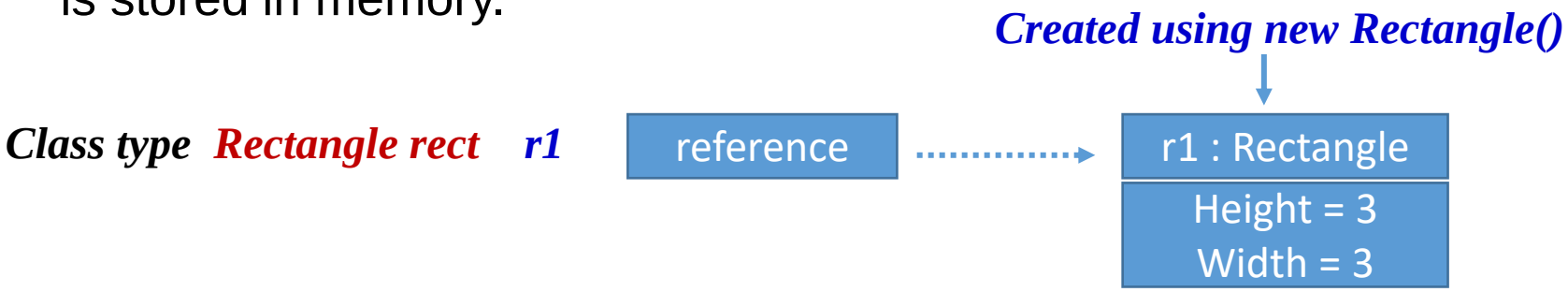
- A variable of a primitive type holds a value of the primitive type.

- Primitive variable j is copied to variable i.



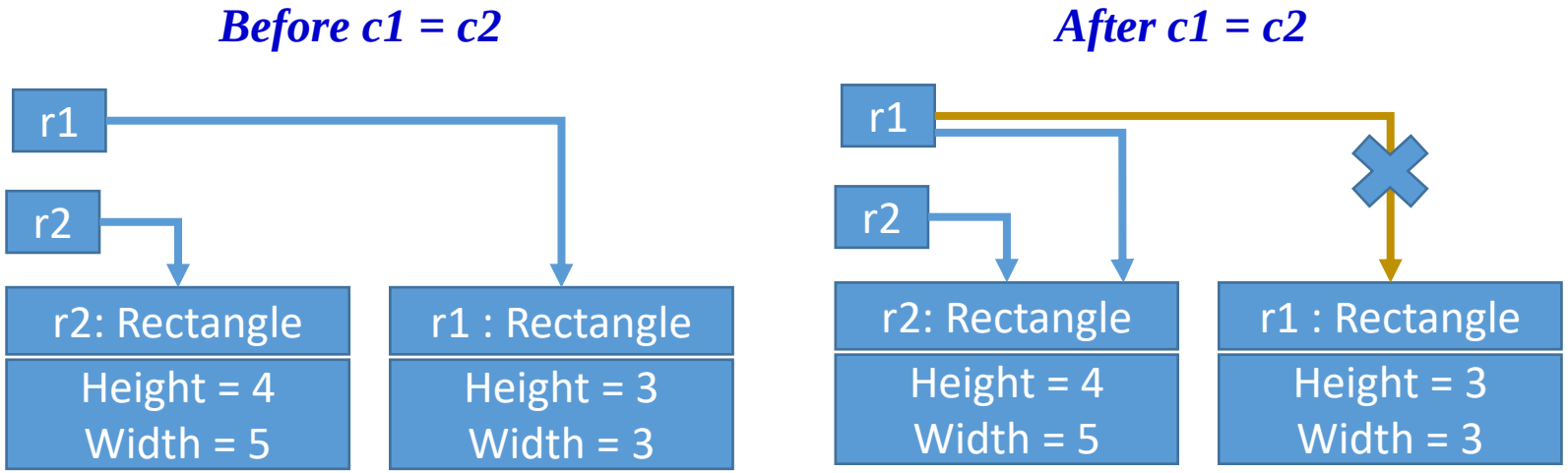
Class Type Variables

- A variable of a reference type holds a reference to where an object is stored in memory.



- Class type variable r2 is copied to variable r1.

Class type assignment r1 = r2



Class Type Variables

- Two class type variables (object type/reference variables) can contain the same reference, and therefore name the same object.
  - The assignment operator sets the reference (memory address) of one class type variable equal to that of another.
  - Any change to the object named by one of theses variables will produce a change to the object named by the other variable, since they are the same object.

```
public class whaleClass {
        public String  stuName;
        public int stuID;
}

whaleCass varStudent1 = new whaleClass (“Catherlin”, 38);

whaleClass varStudent2;
varStudent2 = varStudent1;
```

Default Variable Initializations

- **Default Variable Initializations**: Instance variables are automatically initialized in Java
  - *boolean* types are initialized to **false**
  - Other primitives are initialized to the **zero** of their type
  - Class types are initialized to **null**
- The data fields can be of reference types. The following Student class contains a data field name of the String type. String is a predefined Java class.

```
class Student {
        String name;                // name has the default value null
        int age;                    // age has the default value 0
        boolean isScienceMajor;     // isScienceMajor has default value false
        char gender;                // gender has default value '\u0000'
}
```

Class Parameters

- **Primitive type** parameters in Java are ***call-by-value*** parameters.
  - A parameter is a local variable that is set equal to the value of its argument. Therefore, any change to the value of the parameter cannot change the value of its argument.
  - A method **cannot** change the value of a variable of a *primitive type* that is an *argument* to the method.
- **Class type** parameters appear to behave differently from primitive type parameters. They appear to behave in a way similar to parameters in languages that have the ***call-by-reference*** parameter passing mechanism.
  - Passing an object to a method is to pass the **reference** of the object.
  - A method **can** change the values of the instance variables of *a class type* that is an *argument* to the method.

CallByValueTest.java

```
public class demo {
    public static void main(String[] args) {
        int value = 123;
        System.out.println("Before  : argument value:" + value);
        byValue(value);
        System.out.println("After   : argument value:" + value);
    }
    public static void byValue(int value) {
        value = 678;
        System.out.println("InMethod: argument value:" + value);
    }
}
```

CallByReferenceTest.java

```
public class demo {
    public static void main(String[] args){
        whaleClass aStudent = new whaleClass("PeterWang", 18);
        System.out.println("Before  : argument value - " + aStudent.toString());
        byReference(aStudent);
        System.out.println("After   : argument value - " + aStudent.toString());
    }
    public static void byReference(whaleClass aStudent) {
        aStudent.set("Jacky Lin",42);
        System.out.println("InMethod: argument value - " + aStudent.toString());
    }
}
public class whaleClass {
    private String name;
    private int ID;
    public whaleClass(String initialName, int initialID) {
        name = initialName;
        ID = initialID;
    }
    public String toString( ){
        return (name + " " + ID);
    }
    public void set(String newName, int newID) {
        name = newName;
        ID = newID;
    }
}
```

CallByValueTest.class

```
Before  : argument value:123
InMethod: argument value:678
After   : argument value:123
```

CallByReferenceTest.class

```
Before  : argument value - PeterWang 18
InMethod: argument value - Jacky Lin 42
After   : argument value - Jacky Lin 42
```

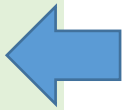
04 Immutable Objects and Classes

Immutable Objects and Classes

- We create an object whose *contents cannot be changed once the object has been created*. We call such an object as **immutable object** and its class as **immutable class**.
  - You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.
  - If a class is immutable, then all its data fields must be **private** and it *cannot* contain public setter methods for any data fields.
  - A class with all private data fields and no mutators is *not necessarily* immutable.
- For a class to be immutable, it must meet the following requirements:
  - *All data fields must be private.*
  - *There can't be any mutator methods for data fields.*
  - *No accessor methods can return a reference to a data field that is mutable.*

Immutable Objects and Classes

```
public class Student {
  private int id;
  private String name;
  private java.util.Date dateCreated;
  public Student(int ssn, String newName) {
                id = ssn;
                name = newName;
                dateCreated = new java.util.Date();
  }
  public int getId() {
                return id;
  }
 public String getName() {
                return name;
  }
  public java.util.Date getDateCreated() {
                return dateCreated;
  }
}
```



```
public class Test {
  public static void main(String[] args) {
    Student student =
                new Student(111223333, "John");
    java.util.Date dateCreated =
                        student.getDateCreated();
    dateCreated.setTime(200000);
    // Now dateCreated field is changed!
  }
}
```

Because, *through getDateCreated()* method,
the content for *dateCreated* can be changed! It is not an immutable class.

05

Programming Practice

Practice 1：Quadratic Equation

① Write a Java program to input double type numbers a, b, c, r1, and r2, where a, b, and c are the coefficients of quadratic equation$aX^2+bX+c=0$. Test whether r1 and r2 are the roots of the quadratic equation. If both r1 and r2 are the roots, print a message to confirm the roots.

② Write a Java program to input **double** type numbers a, b, and c, where a, b, and c are the coefficients of quadratic equation $aX^2+bX+c=0$.

The solution of quadratic equation is:

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

Solve and output the quadratic equation and produce the real roots or complex roots. Also, judge if the solution is a pair of real roots, a pair of complex roots, or multiple real roots.

Practice 1：Quadratic Equation

```
Enter coefficient a: 2
Enter coefficient b: 4
Enter coefficient c: 6
Enter the first root r1: 1
Enter the second root r2: 1
1.0000 and 1.0000 are not the roots of equation 2.0000 x**2 + 4.0000 x +
6.0000 = 0.
```

```
Enter coefficient a: 1
Enter coefficient b: 10
Enter coefficient c: 25

The multiple real root of equation X**2+10.0000X+25.0000=0.0000 is -5.0000.


Enter coefficient a: 4
Enter coefficient b: 2
Enter coefficient c: 6

The complex roots of equation 4.0000X**2+2.0000X+6.0000=0.0000 are -
0.2500+1.1990i and -0.2500-1.1990i.
```

Practice 2：Wish Tea

- Create a class named **WishTea** that stores information about a single fruit tea. It should contain the following:
  1. Private instance variables to store the variety of fruit tea name, the size of the fruit tea (either small, medium, or large), the ice level (either no ice, a little, or regular), and with topping (such as bubble) or not.
  2. Constructor(s) that set all of the instance variables.
  3. Public methods to get and set the instance variables.
  4. A public method named calcCost( ) that returns a double that is the cost of the fruit tea. For example:
     - ✓ The signature fruit tea cost is determined by:
       - Small: $60, Medium: $80, Large: $90, + $5 per topping
     - ✓ The bubble milk tea cost is determined by:
       - Small: $40, Medium: $50, Large: $60, + $10 per topping
  5. A public method named getDescription( ) that returns a String containing the fruit tea size, with topping or not, and the fruit tea cost as calculated by calcCost( ).
- Write a program to let users order several fruit teas and output their descriptions and total cost (include 5% tax).

Practice 2： Wish Tea

```
Please input your order:
         ……


Order Description
=================================================================
1. Signatured Fruit Tea  Small    No Topping       2        $130
2. Bubble Tea                     Large    No Topping       4        $240
-----------------------------------------------------------------
   Total Cost                                                6        $370
=================================================================
```

Practice 3：Rational Number

- A rational number p/q consists of two relatively prime integers p and q, where p is the numerator, q is the denominator, and q cannot be zero. If q is 0, the rational number p/0 is set to the value 0/1 a rational number of value 0. Write a Java program that **defines and implements a rational number class.** The class of rational numbers has three constructors:
  - Default constructor, constructs the rational number 0/1,
  - Constructor of an integer parameter p, constructing the rational number p/1, and
  - Constructor for two integer parameters p and q, constructs a rational number p/q.

  Arithmetic operations of rational numbers:
  - Addition: a/b＋c/d = (ad＋bc)/bd,
  - Subtraction: a/b－c/d = (ad－bc)/bd,
  - Multiplication: a/b×c/d = ac/bd,
  - Division: a/b÷c/d = ad/bc,
  - Absolute value: |a/b| = |a|/|b|.

  Supporting methods:
  - Get numerator: **int** getNume();
  - Get denominator: int getDeno();
  - Set numerator: void setNume(int);
  - Set denominator: void setDeno(int);
  - Print rational number: void printRational();

Practice 3：Rational Number

If rational number p/q that p and q are not co-prime, simplify p/q by dividing p and q by their **greatest common divisor (GCD)** If p/q is a negative rational number, simplify the rational number to p<0 and q>0.

In the application class, get the five rational numbers a, b, c, d, and e, and print them in the main program. Calculate and output the following arithmetic expressions:

① a + b,
② c - d,
③ a × b,
④ c ÷ d,
⑤ |e|,
⑥ (a × |d - b|) - (b + (c ÷ a)) × |(b × e) - (c ÷ d)|.

Practice 3： Rational Number

```
a = -1/2
b = 1/2
c = -4
d = 1/4
e = 1/2

-----------------------------------------------

a+b = 0
c-d = -17/4
a*b = -1/4
c/d = -16
|e| = 1/2
a*|d-b|-(b+c/a)*|b*e-c/d| = -553/4
```