OBJECT-ORIENTED SYSTEMS DESIGN
[Exercise]: UML and Patterns

2022/05/25

Department of Computer Science,
Hanyang University

Lecturer: Taeuk Kim
TA: Taejun Yoon, Seong Hoon Lim, Young Hyun Yoo

UML

Chapter 12.1

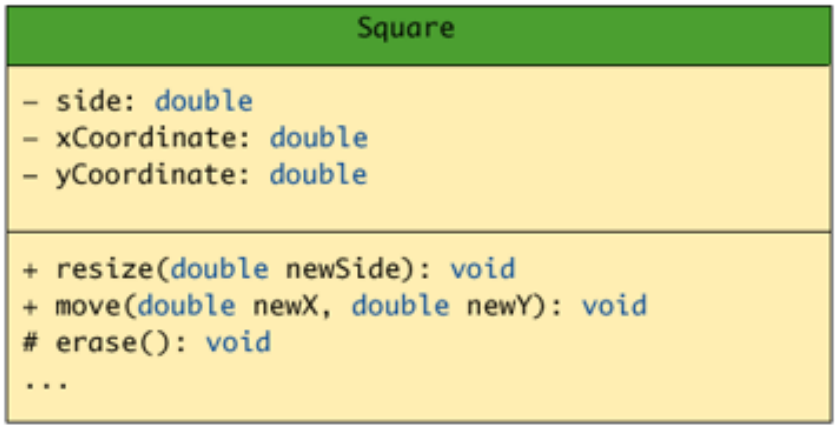
UML

- **Pseudocode is a way of representing a program in a linear and algebraic manner.**
  - It simplifies design by eliminating the details of programming language syntax.
- **Graphical representation systems for program design have also been used.**
  - *Flowcharts* and *structure diagrams* for example.
- ***Unified Modeling Language (UML)* is yet another graphical representation formalism.**
  - UML is designed to reflect and be used with the OOP philosophy.

A UML Class Diagram

```java
package exercise;

public class Square {
    private double side;
    private double xCoordinate;
    private double yCoordinate;

    public void resize(double newSide) {
        this.side = newSide;
    }

    public void move(double newX, double newY) {
        this.xCoordinate = newX;
        this.yCoordinate = newY;
    }

    protected void erase() {
        this.side = 0;
    }
}
```

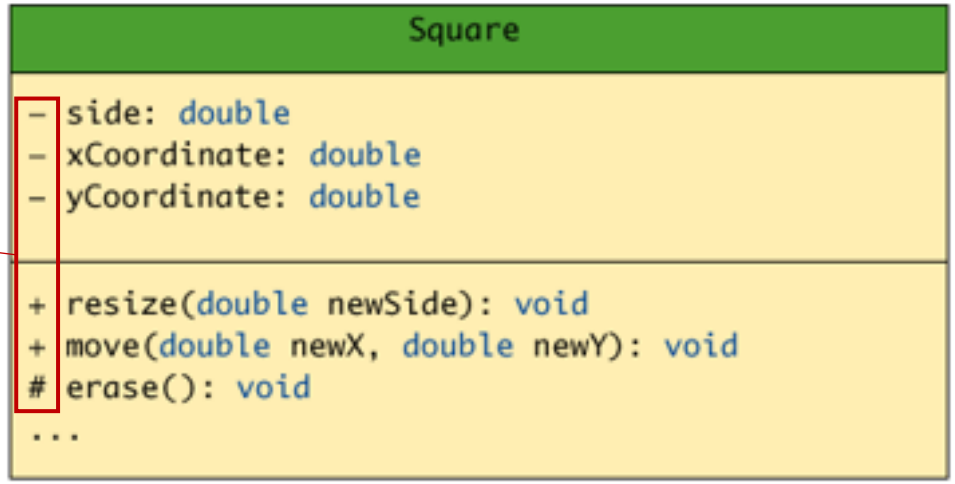
Code representation



UML Class Diagram representation

A UML Class Diagram

Access type :
(-) for private,
(+) for public,
and
(#) for protected



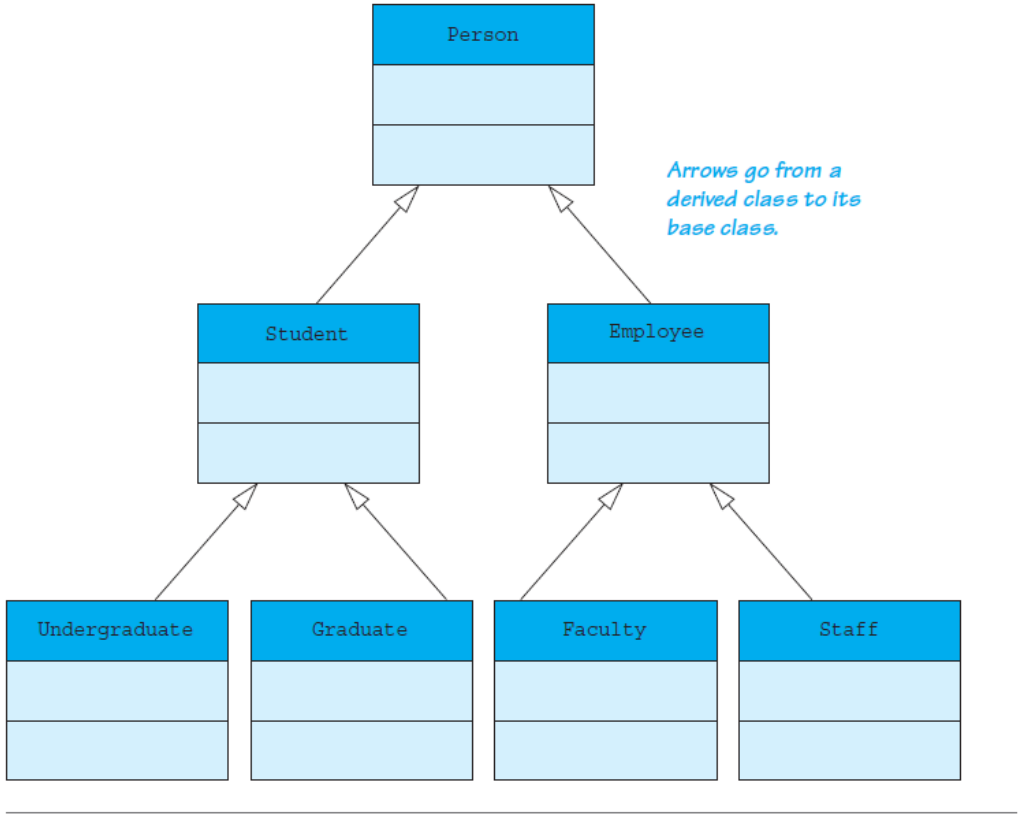
Instance variables

Class methods

Inheritance Diagrams

- **An *inheritance diagram* shows the relationship between a base class and its derived class(es).**

Display 12.2 A Class Hierarchy in UML Notation

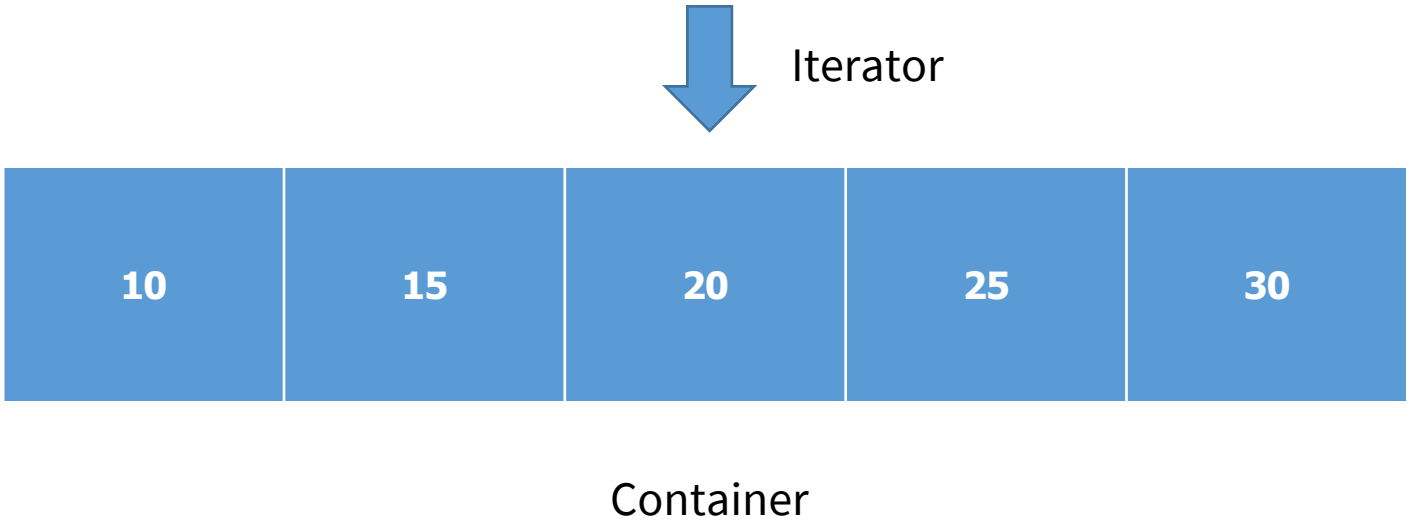


Patterns

Chapter 12.2

Container-Iterator Pattern

- **A *container* is a class or other construct whose objects hold multiple pieces of data.**
  - An array is a kind of container.
- **Any construct that can be used to cycle through all the items in a container is an *iterator*.**
  - An array index is an iterator for an array.



Container-Iterator Pattern examples

```java
public static void main(String[] args) {
    // Array is a container
    int[] array = new int[10];

    // Array index is an iterator for an array
    for(int i = 0; i < array.length; i++) {
        array[i] = i + 1;
    }
}
```

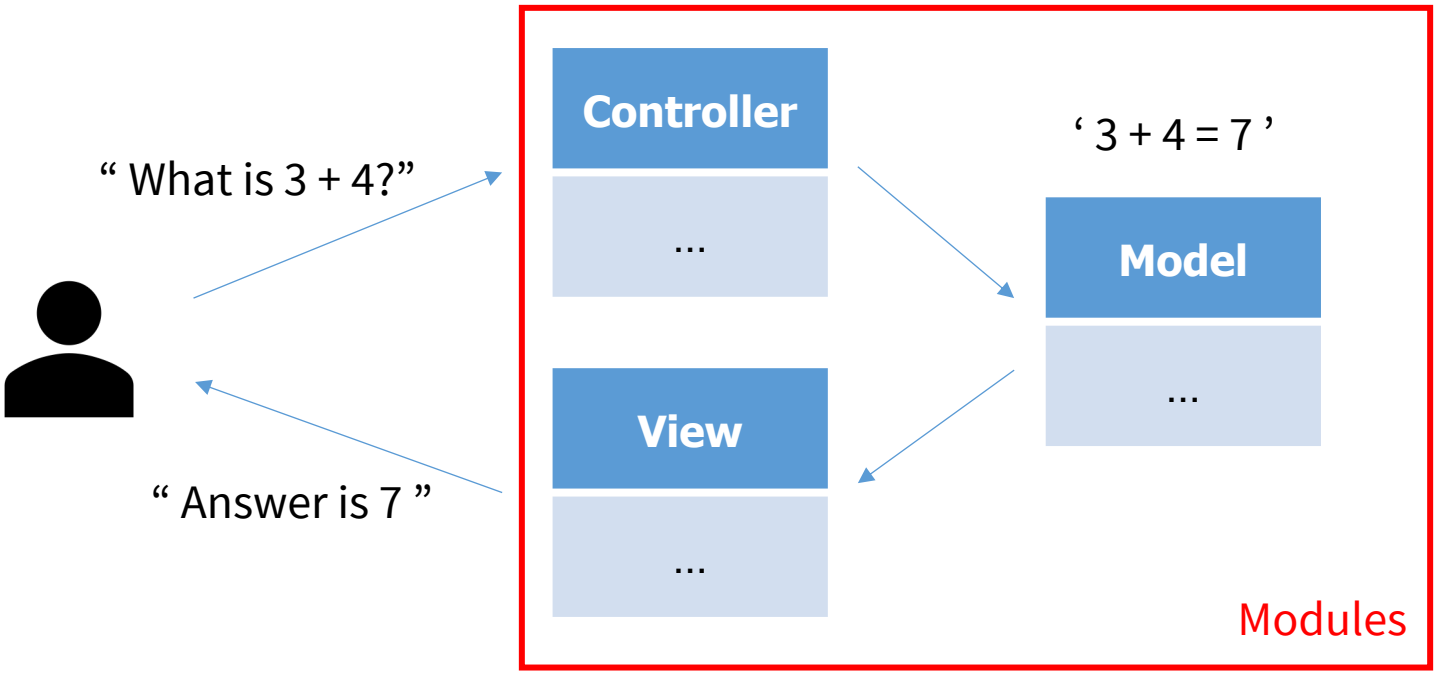
Array-index pattern

```java
public static void main(String[] args) {
    // Vector is a container (We'll cover it Chapter 16)
    Vector<Integer> vec = new Vector<Integer>();

    // Initialization
    for(int i = 1; i <= 5; i++) {
        vec.add(i);
    }

    // It has 'Iterator' type instance objects
    for(Iterator iter = vec.iterator(); iter.hasNext();)
        System.out.println(iter.next());
}
```

Vector-Iterator pattern

The Model-View-Controller Pattern

- **The *Model-View-Controller* pattern is a way of separating the I/O task of an application from the rest of the application.**
  - The **Model** part of the pattern performs the heart of the application.
  - The **View** part displays (outputs) a picture of the Model's state.
  - The **Controller** is the input part: It relays commands from the user to the Model.



OBJECT-ORIENTED SYSTEMS DESIGN
[Exercise]: Interfaces and Inner Classes

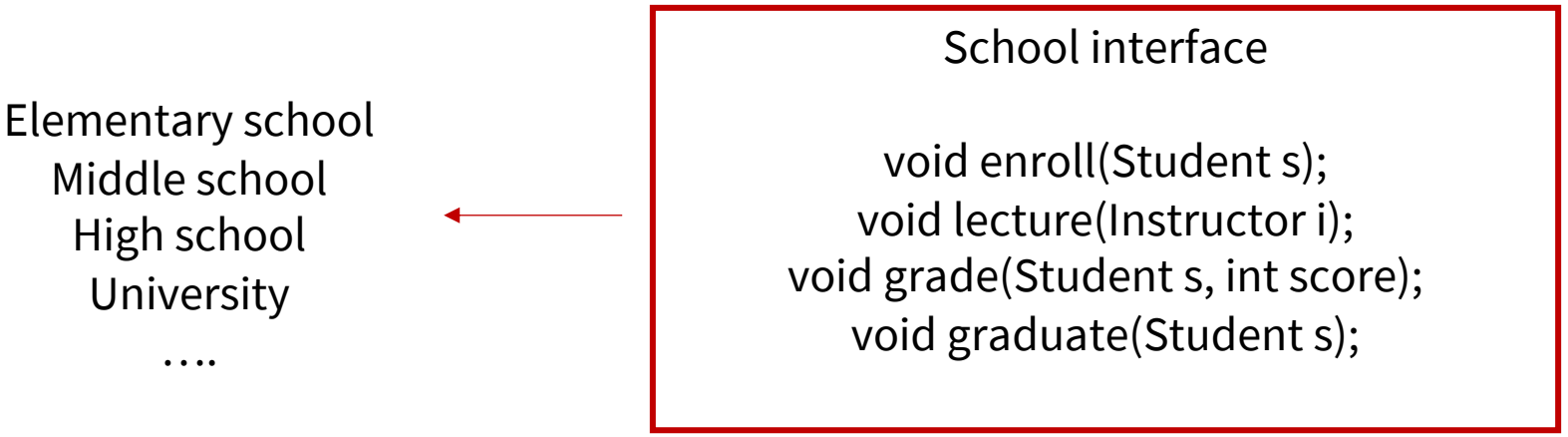
2022/05/25

Department of Computer Science,
Hanyang University

Lecturer: Taeuk Kim
TA: Taejun Yoon, Seong Hoon Lim, Young Hyun Yoo

Interfaces

- **An interface specifies a set of methods that any class that implements the interface must have.**
  - Suppose there is a 'school interface' that every school must follow.
  - Every school must implement 'enroll', 'lecture' ,'grade' and 'graduate' methods.



Interfaces

- **An interface specifies a set of methods that any class that implements the interface must have.**
  - Suppose there is a ‘school interface’ that every school must follow.
  - Every school must implement ‘enroll’, ‘lecture’ ,‘grade’ and ‘graduate’ methods

Elementary school
Middle school
High school
University
….


```java
public interface School {
    // Must implement these methods For all class that implements this interface
    void enroll(Student s);
    void lecture(String subject);
    void grade(Student s, int score);
    void graduate(Student s);
}
```

Interfaces Example

- **School interface example : ElementarySchool class**

```java
public class ElementarySchool implements School{
    private Student[] students;
    private int numStudents = 0;
    private static final int MAX_STUDENTS = 1000;

    @Override
    public void enroll(Student s) {
        if(numStudents < MAX_STUDENTS) {
            students[numStudents] = s;
            numStudents++;
        }
    }

    @Override
    public void lecture(String subject) {
        System.out.println("[" + subject +"] Studying hard...");
    }
```

```java
@Override
public void grade(Student s, int score) {
    if(score > 60) {
        System.out.println("Pass!");
    } else {
        System.out.println("Fail...");
    }
}

@Override
public void graduate(Student s) {
    if(s.getGrade() > 6) {
        System.out.println("Congratulations!");
    }
}
```

Interfaces Example

- **School interface example : MiddleSchool class**

```java
public class MiddleSchool implements School{
    private Student[] students;
    private int numStudents = 0;
    private static final int MAX_STUDENTS = 1000;

    @Override
    public void enroll(Student s) {
        if(numStudents < MAX_STUDENTS) {
            students[numStudents] = s;
            numStudents++;
        }
    }

    @Override
    public void lecture(String subject) {
        System.out.println("[" + subject +"] Studying hard...");
    }
```

```java
@Override
public void grade(Student s, int score) {
    if(score > 60) {
        System.out.println("Pass!");
    } else {
        System.out.println("Fail...");
    }
}

@Override
public void graduate(Student s) {
    if(s.getGrade() > 3) {
        System.out.println("Congratulations!");
    }
}
```

Seems very similar to ElementarySchool class…

Interfaces

- **We can define *default* methods (optional)**
  - Interface can provide a default body for a method.
  - Derived class can use the method directly or override for its purpose.

```java
public interface School {
    // Must implement these methods For all class that implements this interface
    void enroll(Student s);

    default void lecture(String subject) {
        System.out.println("[" + subject +"] Studying hard...");
    }

    default void grade(Student s, int score) {
        if(score > 60) {
            System.out.println("Pass!");
        } else {
            System.out.println("Fail...");
        }
    }
    void graduate(Student s);
}
```

Simple Uses of Inner Classes

Chapter 13.2

Simple Uses of Inner Classes

- **Inner classes are classes defined within other classes.**
  - The class that includes the inner class is called the outer class.
  - Inner and outer classes have access to each other's private members

```
public class Calculator {
    public static class ComplexNumber {
        private int real;
        private int imaginary;

        public ComplexNumber(int real, int imaginary) {
            this.real = real;
            this.imaginary = imaginary;
        }
    }

    public static ComplexNumber add(ComplexNumber com1, ComplexNumber com2) {
        return new ComplexNumber(com1.real + com2.real, com1.imaginary + com2.imaginary);
    }

    public static ComplexNumber sub(ComplexNumber com1, ComplexNumber com2) {
        return new ComplexNumber(com1.real - com2.real, com1.imaginary - com2.imaginary);
    }
}
```

Declarations

- **In the case of a non-static inner class, it must be created using an object of the outer class.**

- **In the case of a static inner class, the procedure is similar to, but simpler than, that for non-static inner classes.**

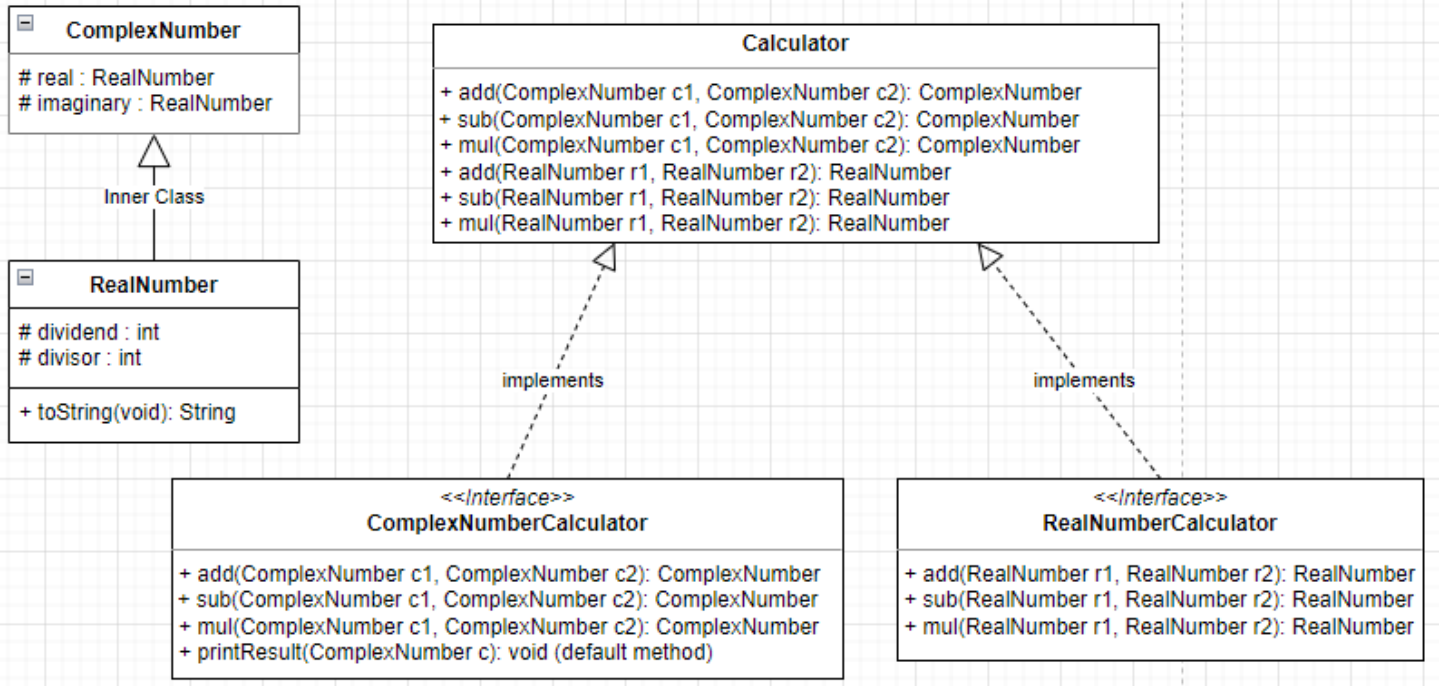
```
public static void main(String[] args) {
    // If inner class is declared as non-static
    Calculator calc = new Calculator();
    Calculator.ComplexNumber comp1 = calc.new ComplexNumber(1, 2);

    // If inner class is declared as static
    Calculator.ComplexNumber comp2 = new Calculator.ComplexNumber(5, 3);

}
```

Practice

Exercise/WeekN/ComplexNumberCalculator.java,
/RealNumberCalculator.java,
/ComplexNumber.java
/Calculator.java

Practice for Today

- **Define a Calculator for complex numbers**
  - Specification is listed as below.
  - All of calculations are based on mathematics.
  - Define the main method that ensures all of requirements are satisfied.



Practice for Today

- **Expected output**
  - You don’t have to **reduce a fraction.** (약분 필요 X)
    - Any form is allowed if the value is true (e.g., 1/10=10/100=100/1000).
    - But you may implement it if you want.

```
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        ComplexNumber c1 = new ComplexNumber(new ComplexNumber.RealNumber(4, 10), new ComplexNumber.RealNumber(3, 2));
        ComplexNumber c2 = new ComplexNumber(new ComplexNumber.RealNumber(3, 10), new ComplexNumber.RealNumber(-4, 2));

        calc.printResult(calc.sub(c1, c2));
    }
```

```
Main ×
C:\Users\LSH\.jdks\openjdk-17.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2021
Real : 10/100, Imaginary : 14/4

Process finished with exit code 0
```

```
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        ComplexNumber c1 = new ComplexNumber(new ComplexNumber.RealNumber(4, 10), new ComplexNumber.RealNumber(3, 2));
        ComplexNumber c2 = new ComplexNumber(new ComplexNumber.RealNumber(3, 10), new ComplexNumber.RealNumber(-4, 2));

        calc.printResult(calc.mul(c1, c2));
    }
```

```
Main ×
C:\Users\LSH\.jdks\openjdk-17.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 202
Real : 12/100, Imaginary : -12/4

Process finished with exit code 0
```

Time for Practice

Get it started, and ask TAs if you are in a trouble.