

## JOB SHEET 7 - OVERLOADING DAN OVERRIDING

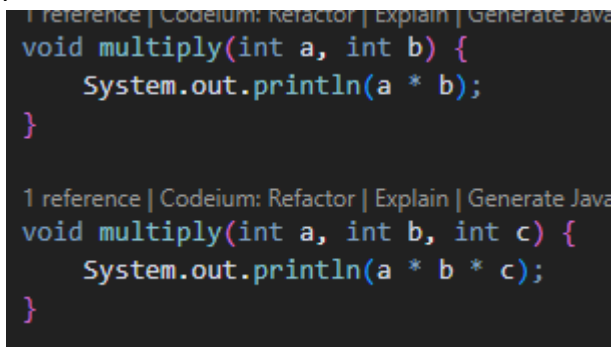
Kevin Bramasta Arvyto Wardhana

2341720233 / 13

### Excercise

1. From the above source code, where is the overloading?

The class has two methods with the same name, multiply, but with different parameter lists:



```
1 reference | Codeium: Refactor | Explain | Generate Java
void multiply(int a, int b) {
    System.out.println(a * b);
}

1 reference | Codeium: Refactor | Explain | Generate Java
void multiply(int a, int b, int c) {
    System.out.println(a * b * c);
}
```

The overloading occurs because both methods share the same name, multiply, but differ in the number of parameters they accept.

2. If there any overloading, how many parameters are different?

In the overloaded methods of the MyMultiplication class, the difference is in the number of parameters:

- The first multiply method has 2 parameters: (int a, int b).
- The second multiply method has 3 parameters: (int a, int b, int c).

The difference is 1 parameter between the two methods. This variation in the parameter count is what enables method overloading.

```

package week7;

public class MyMultiplication {

    void multiply(int a, int b) {
        System.out.println(a * b);
    }

    void multiply(double a, double b) {
        System.out.println(a * b);
    }

    public static void main(String args[]) {
        MyMultiplication obj = new MyMultiplication();

        obj.multiply(25, 43);
        obj.multiply(34.56, 23.7);
    }
}

```

3. From the above source code, where is the overloading?

The multiply method is overloaded because there are two versions with the same name but different parameter types:

void multiply(int a, int b) takes two int parameters.

void multiply(double a, double b) takes two double parameters.

The Java compiler can differentiate between the two methods by providing different parameter types based on the arguments passed when calling the multiply method in the main method.

4. If there any overloading, how many parameters are different?

The first multiply method takes two int parameters: int a and int b.

The second multiply method takes two double parameters: double a and double b.

```

4 references | Codeium: Refactor | Explain | Qodo Gen: Options | Test this class
public class Fish {
2 references | Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Options |
    public void swim() {
        System.out.println(x:"Fish can swim");
    }
}

1 reference | Codeium: Refactor | Explain | Qodo Gen: Options | Test this class
class Piranha extends Fish {

2 references | Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Options |
    public void swim() {
        System.out.println(x:"Piranha can eat meat");
    }
}

0 references | Codeium: Refactor | Explain | Qodo Gen: Options | Test this class
public class FishMain {
Run | Debug | 0 references | Codeium: Refactor | Explain | Generate J
    public static void main(String[] args) {
        Fish a = new Fish();
        Fish b = new Piranha();
        a.swim();
        b.swim();
    }
}

```

5. From the above source code, where is the overriding?

the overriding is happening in the Piranha class where the swim method is being overridden.

In Java, method overriding occurs when a subclass provides a specific implementation of a method that is already provided by its superclass. In this case, the swim method in the Piranha class overrides the swim method from the Fish class.

By providing a new implementation for the swim method in the Piranha class, we are changing the behavior of the method for instances of the Piranha class without modifying the superclass Fish implementation

But you should put an `@Override` annotation in the Piranha class when overriding a method from a superclass. This annotation helps to ensure that you are actually overriding a method from the superclass. If you mistakenly misspell the method name or change the method signature in the superclass, the compiler will generate an error, alerting you to the issue.

```

4 references | Codeium: Refactor | Explain | Qodo Gen: Options | Test this class
public class Fish {
2 references | Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Opti
    public void swim() {
        System.out.println(x:"Fish can swim");
    }
}

1 reference | Codeium: Refactor | Explain | Qodo Gen: Options | Test this class
class Piranha extends Fish {
    Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Options | Test t
    @Override
    2 references
    public void swim() {
        System.out.println(x:"Piranha can eat meat");
    }
}

```

6. If there any overriding, how many parameters are different?

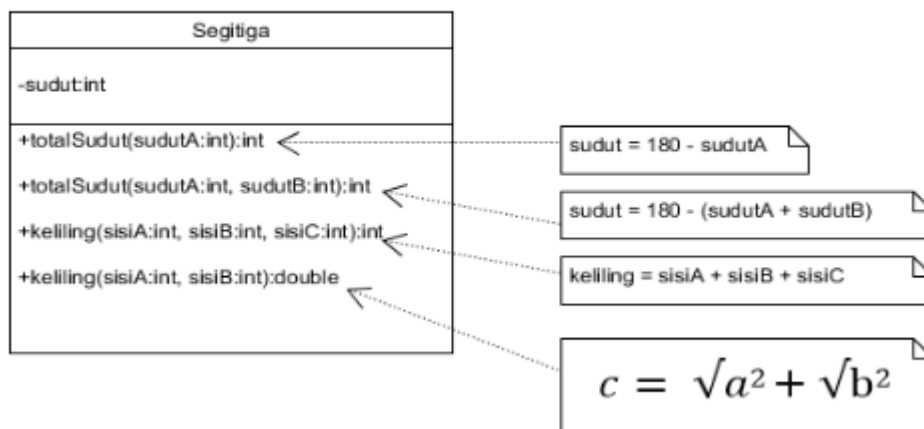
In the given code snippet, the swim method in the Piranha class is overriding the swim method in the Fish class. When overriding a method in Java, the method signature, including the number and types of parameters, must be the same in both the superclass and subclass.

In this case, the swim method in both the Fish and Piranha classes does not take any parameters. Therefore, there are no differences in the number of parameters between the overridden method in the superclass and the overriding method in the subclass.

## Task

### Overloading

Implement overloading concept into this class diagram:



```

package week7;

0 references | Codeium: Refactor | Explain | Qodo Gen: Options | Test this class
public class Triangle {
    private int angle;

    0 references | Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Options | Test this method
    int totalAngle(int angleA) {
        return angle = 180 - angleA;
    }

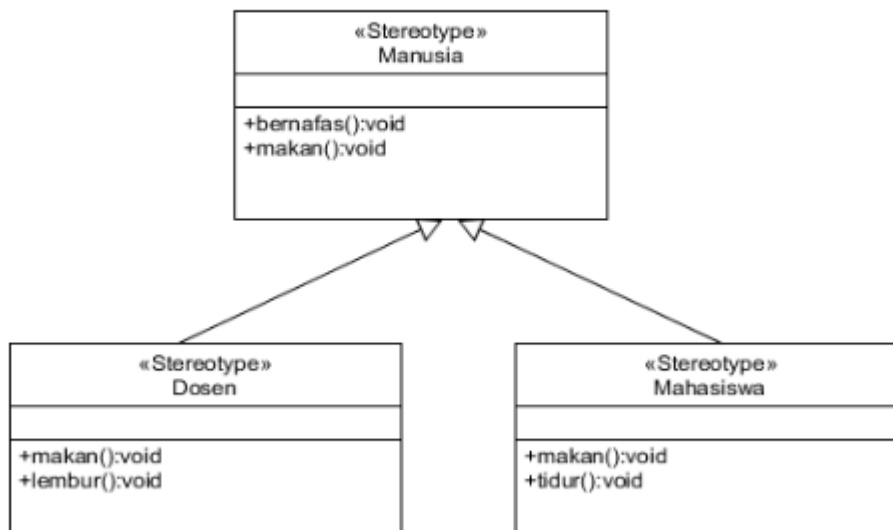
    0 references | Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Options | Test this method
    int totalAngle(int angleA, int angleB) {
        return angle = 180 - (angleA + angleB);
    }

    0 references | Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Options | Test this method
    int area(int angleA, int angleB, int angleC) {
        return angleA + angleB + angleC;
    }

    0 references | Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Options | Test this method
    double area(int angleA, int angleB) {
        return Math.sqrt(Math.pow(angleA, b:2)) + Math.sqrt(Math.pow(angleA, b:2));
    }
}
  
```

## Overriding

Implement overriding for these class using dynamic method dispatch :



```
public class Human {
    2 references | Codeium: Refactor | Explain | Generate Javadoc | X | C
    void breathe() {
        System.out.println(x:"Human breathe");
    }

    2 references | Codeium: Refactor | Explain | Generate Javadoc | X | C
    void eat() {
        System.out.println(x:"Human eat");
    }
}
```

```
public class Lecturer extends Human{
    Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Options | T
    @Override
    2 references
    public void eat() {
        System.out.println(x:"Lecturer eat");
    }

    1 reference | Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen
    void overtime() {
        System.out.println(x:"Lecturer overtime");
    }
}
```

```

public class Student extends Human {
    Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Opt
    @Override
    2 references
    public void eat() {
        System.out.println(x:"Student eat");
    }

    1 reference | Codeium: Refactor | Explain | Generate Javadoc | X | Q
    void sleep() {
        System.out.println(x:"Student sleep");
    }
}

```

```

public class HumanMain {
    Run | Debug | 0 references | Codeium: Refactor | Explain | Generate Javadoc | X | Qodo Gen: Options | Test this
    public static void main(String[] args) {
        Human human1 = new Lecturer();
        Human human2 = new Student();

        // Dynamic method dispatch in action
        human1.breathe();
        human1.eat(); // Will call Lecture's eat()

        human2.breathe();
        human2.eat(); // Will call Student's eat()

        // To call subclass-specific methods, we need to cast the objects
        if (human1 instanceof Lecturer) {
            ((Lecturer) human1).overtime();
        }

        if (human2 instanceof Student) {
            ((Student) human2).sleep();
        }
    }
}

```

Output:

```

Human breathe
Lecturer eat
Human breathe
Student eat
Lecturer overtime
Student sleep

```

Method Overriding: The makan() method is overridden in both Dosen and Mahasiswa classes.

Dynamic Method Dispatch: The overridden makan() method is invoked based on the actual object (either Dosen or Mahasiswa) at runtime.

Type Casting: Casting is used to access subclass-specific methods like lembur() and tidur().

<https://github.com/justKevv/Uni-stuff/tree/main/programming-courses/object-oriented-programming/week7>