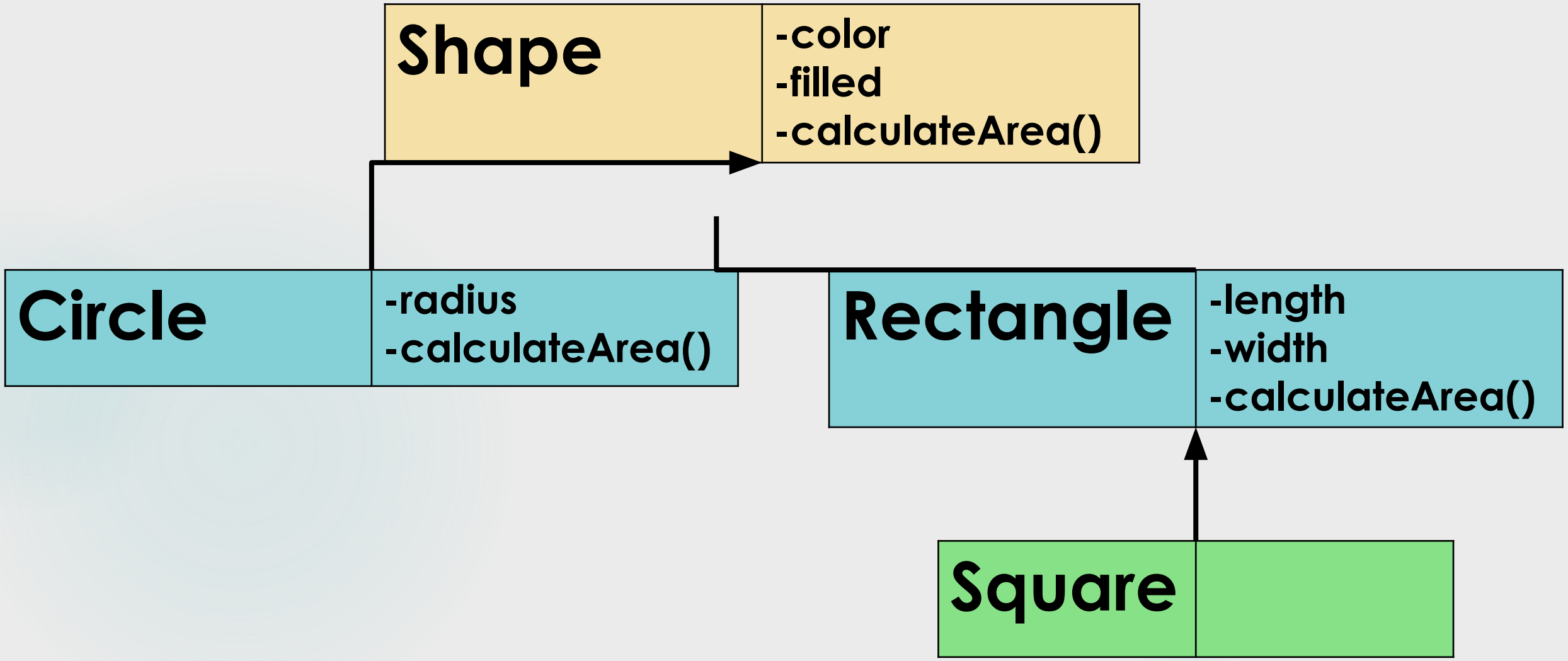
Object Oriented Programming Lab

CSE 1206

LAB 6

Remember the inheritance example?



Previous Codes:

Shape Class's Code:
https://paste.ubuntu.com/p/jMpNRPFcQg/

Circle Class's Code:
https://paste.ubuntu.com/p/KSrj7n3YXY/

Rectangle Class's Code:
https://paste.ubuntu.com/p/QzybQfddNn/

Square Class's Code:
https://paste.ubuntu.com/p/ZXVtqqrwzd/

A superclass reference variable can refer to a subclass object.

```
Shape obj1 = new Circle(5.5);
Shape obj2 = new Rectangle(5.5,6.7);

Rectangle obj3 = new Square(5.5);
Shape obj4 = new Square(6.5);
```

This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

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


    Shape obj1 = new Circle(5.5);
    Shape obj2 = new Rectangle(5.5,6.7);

    Rectangle obj3 = new Square(5.5);
    Shape obj4 = new Square(6.5);


    System.out.println(obj1.calculateArea());
    System.out.println(obj2.calculateArea());
    System.out.println(obj3.calculateArea());
    System.out.println(obj4.calculateArea());
```

Dynamic Method Dispatch

- This upcasting is used to call overridden methods.
- The super class variable will only recognize the overridden methods.
- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time.

Example

```
public static void callArea(Shape shpObj, String type)
{
    if(type.equals("circle"))
    {
        shpObj= new Circle(3.5);
        System.out.println(shpObj.calculateArea());
    }
    else if(type.equals("rectangle")){

        shpObj= new Rectangle(5,6);
        System.out.println(shpObj.calculateArea());
    }
    else if(type.equals("square")){

        shpObj= new Square(6.7);
        System.out.println(shpObj.calculateArea());
    }
```

```
Shape callObj = null;
callArea(callObj, "rectangle");
```

Company Sales Software for different types of Orders

Store Order

Factory Order

- Process the order
- Validate the order
- Cancel the order

You will need an employee to process the order. So there will be an employee class

Employee

- empID;
- name;
- designation;

An order will be created against a product. So there will be a product class:

Product

- productID;
- productType;
- productName;

Create a project and create the Product, Employee and Order class.

Product Class

https://paste.ubuntu.com/p/SpSdvcTB5b/

Employee Class

https://paste.ubuntu.com/p/SYRn6G2cC8/

Order Class

https://paste.ubuntu.com/p/TNk8wXsHv6/

What if more order types are created?

Store Order

Factory Order

Online Order

Facebook Order

Instagram Order

Solution:
Create separate classes for each order type but each class must be **forced to implement** the methods process() and validate()

This can be done using **Abstract**

Abstract Class

A class which contains the abstract keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain abstract methods, i.e., **methods without body** (**public void get();**)
- But, if a class has at least one abstract method, then the class must be declared abstract.
- If a class is declared abstract, **it cannot be instantiated**.
- To use an abstract class, you have to inherit it in another class and provide implementations to all the abstract methods in it.

Rules:

- If you declare variables in abstract class then they must be initialized using a parameterized constructor.
- The parameterized constructor must be called in the sub class using super();
- An abstract class cannot be initialized with its own constructor. You will need to use the sub class constructor to do so.

An Example of how to make and use abstract class

```
public abstract class Employee {

    private String name;
    private String address;
    private int number;

    Employee(String name, String address, int number) {

    }
    //ABSTRACT METHOD
    abstract double computePay();

    //NON-ABSTRACT METHOD
    void mailCheck() {
        System.out.println("Mailing a check to " + this.name
                           + " " + this.address);
    }
```

Abstract Method has no body

```
public class Salary extends Employee {

    private double salary;   // Annual salary

    //Must implement the Abstract Method
    double computePay() {
        System.out.println("Computing monthly salary pay for " + getName());
        return salary / 12;
    }

    //Overriding the Non Abstract Method as well
    void mailCheck() {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName()
                + " with salary " + salary
                + " and check no: " + getNumber());
    }
```

```java
public class AbstractDemo {

    public static void main(String[] args) {

        Salary s = new Salary("Zakia", "Mirpur", 3, 360000.00);
        Employee e = new Salary("Tania", "Gulshan", 2, 2400000.00);
        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();
        System.out.println(s.computePay());
        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
        System.out.println(e.computePay());
    }
}
```

Now Declare the two methods abstract in Order class

```
public abstract void validate();

public abstract void process() ;
```

Create two separate class for Store and Factory Orders and design the code accordingly.