JAVA

OOP

STUDY GUIDE

BY BATOUL HAMIEH

TABLE OF CONTENT

CHAPTER 01

OOP BASICS

~~Chapter 01 OOP Basics~~

- **What is OOP ?**

Object-Oriented Programming is a programming paradigm based on the concept of "objects", which are instances of classes that encapsulate data and behavior. It emphasizes four main principles: encapsulation, inheritance, polymorphism, and abstraction, enabling modular, reusable, and maintainable code.

- **What is an object ?**

An object represents an entity in the real world. It has a unique state (set of data fields/attributes) and behaviors (set of methods).

- **What is a class ?**

A class is a template that defines objects of the same type, and what each objects' data fields and methods will be. Creating an instance (object) of a class is referred to as instantiation.

- **What is a constructor ?**

A class provides a special method, called constructor, that is invoked to construct objects from the class. The constructor method has the same name as the class. It has no return type, not even void. Three types of constructors :
- No argument constructor : doesn't take arguments.
- Constructor with arguments : takes one or more argument.
- Default constructor : A no argument constructor with empty body. It is provided automatically in the class if no constructors are declared explicitly.

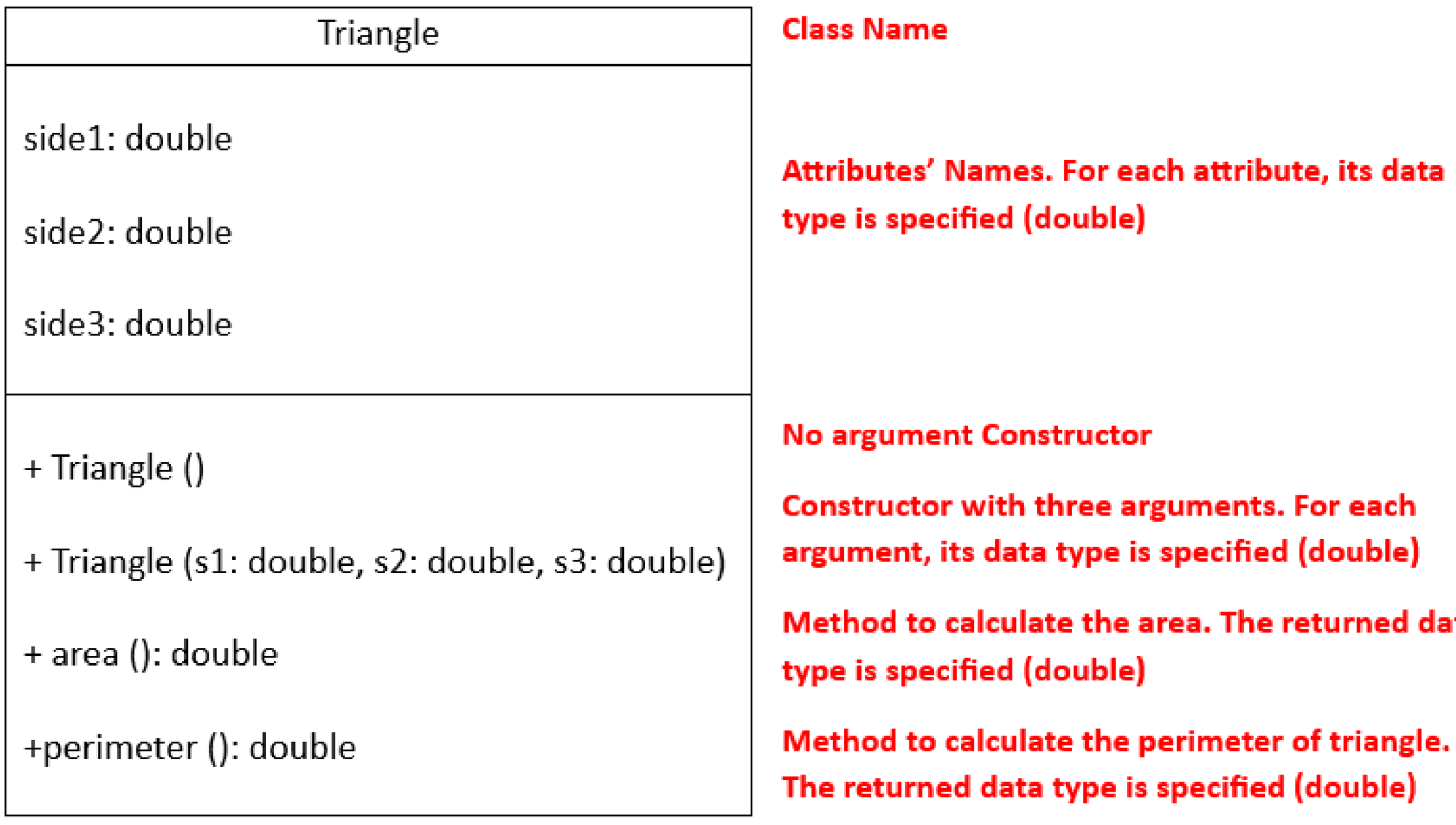
- **What is a UML Class Diagram ?**

Unified Modeling Language Class Diagram is a visual representation of the structure of the classes, showing their attributes, methods, and the relationships between them.

- **Question :**

Design a class named Triangle to represent a triangle.
The class contains :
- Three double data fields named side1, side2, and side3 that specify the three sides of the triangle.
- A no arg constructor that creates a default triangle. Use the value 1 for each of the three sides.
- A constructor that creates a triangle with specified values for side1, side2, and side3.
- A method named perimeter() that returns the perimeter of this triangle.
- A method named area() that returns the area of this triangle.
The area is calculated using the below formula :
area = $\sqrt{[s*(s-side1)*(s-side2)*(s-side3)]}$
s = (side1 + side2 + side3) / 2

- **Solution :**
  - UML diagram of class Triangle :



- **Solution :**
  - Class Circle :

```
public class Circle {

double radius;
static int nbOfObjects = 0;

public Circle (double r) {
radius = r;
nbOfObjects ++;
}

public double getArea () {
return radius*radius*Math.PI;
}

public static int getNbOfObjects () {
return nbOfObjects;
}

}
```

- **Solution :**
  - Test Code :

```
public class Application {

public static void main (String [] args) {
// main() function in class Application

Triangle triangle1 = new Triangle();
 /* To create an object of class Triangle, note that in this
command:
Triangle: reference type
triangle1: object reference type
Triangle(): no-arg constructor that initializes an object */

Triangle triangle2 = new Triangle(4,5,6);
 /* another object of class Triangle is created using
constructor with arguments
where side1=4cm, side2=5cm, and side3=6cm */

System.out.println("Triangle 1:");
System.out.println("The area is" +triangle1.area());
//triangle1.area(): to invoke object's method
System.out.println("The perimeter is
"+triangle1.perimeter());
System.out.println("Triangle 2:");
System.out.println("The area is "+triangle2.area());
System.out.println("The perimeter is
"+triangle2.perimeter());
}
}
```

STATICS :

- **Instance Vs Static Data Field :**

An instance variable is tied to a specific object of the class.
This variable is only accessed via the object name.
A static variable is shared by all objects of the class. It stores its value in a memory location common to all objects of the class.
This variable is accessed either via class name or via object name.

- **Instance Vs Static Method :**

An instance method is a method that returns instance data field.
A static method is a method that returns a static data field.

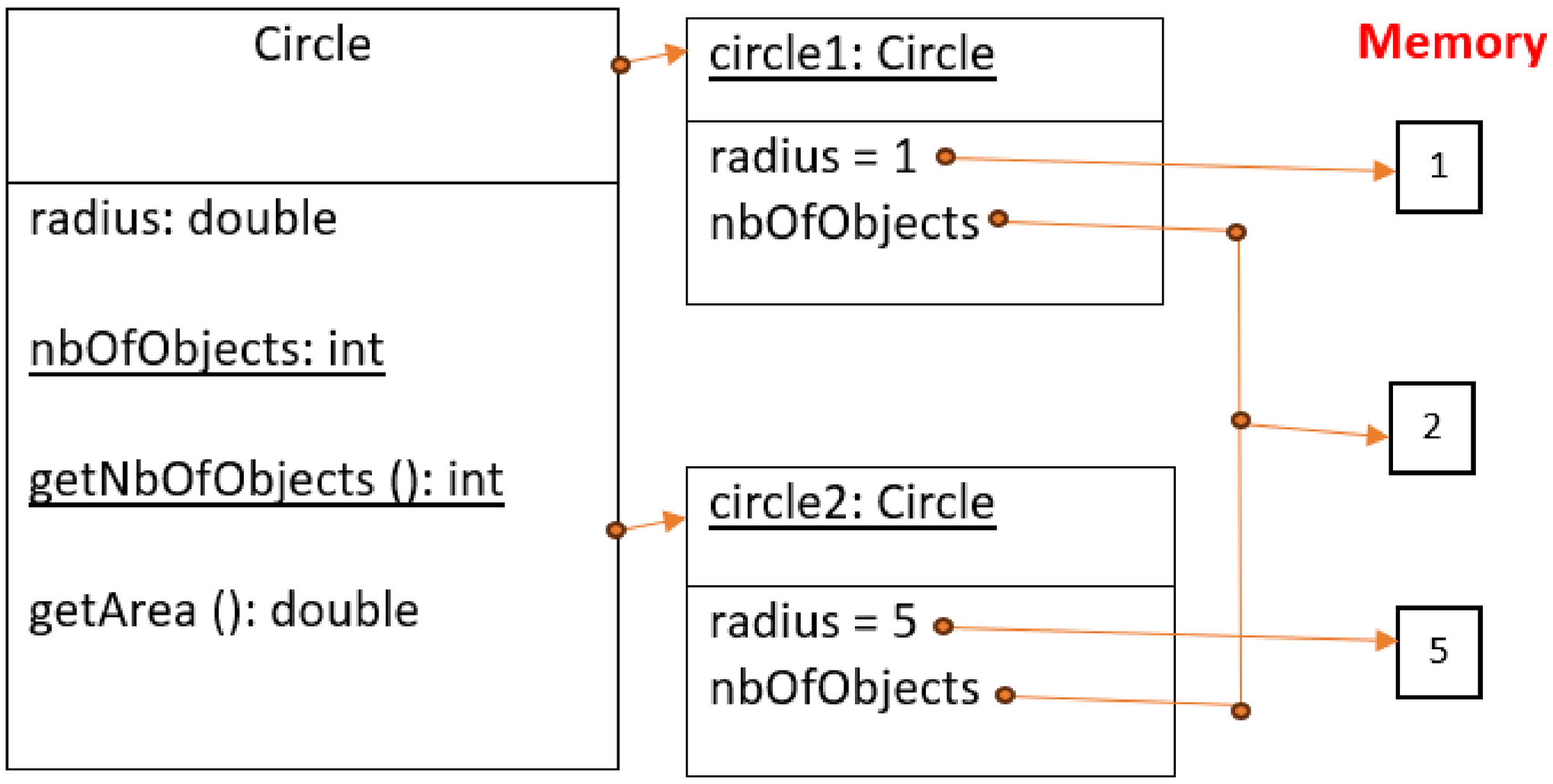
- **Static Constant :**

A static constant is a final variable shared by all instance of the class.
Ex : PI in Math class

Notes :

- Static data fields and methods are underlined in the UML diagram.
- Through coding, add the 'static' keyword before the datatype of a data field or the returned type of a method, to indicate that it is a static one.

- **Example :**
  - UML Diagram :



  - **Class Circle :**

```
public class Circle {

double radius;
static int nbOfObjects = 0;
public Circle (double r) {
radius = r;
nbOfObjects ++;
}
public double getArea () {
return radius*radius*Math.PI;
}
public static int getNbOfObjects () {
return nbOfObjects;
}
}
```

- **Solution :**
  - Test Code :

```
public class Test {

 public static void main (String [] args) {

 System.out.println("Before creating objects:");
 System.out.println("The nb of Circle objects is
"+Circle.nbOfObjects);//access the static variable via
class name

 Circle circle1 = new Circle(1);
 System.out.println("After creating circle1:");
 System.out.println("The nb of Circle objects is
"+circle1.nbOfObjects);//access nbOfObjects via
circle1 object

 Circle circle2 = new Circle(5);
 System.out.println("After creating circle2:");
 System.out.println("The nb of Circle objects is
"+circle2.nbOfObjects);//output: The nb of Circle
objects is 2

 System.out.println("The nb of Circle objects is
"+Circle.nbOfObjects);//output: The nb of Circle
objects is 2
}


}
```

VISIBILITY MODIFIERS :

- **public :**

The data field or method is visible to any class in any package.
' + ' sign in UML means public.

- **private** :

The data field or method can be accessed only by the declaring class.
' - ' sign in UML means private.

- **default** :

By default, the data field or method can be accessed by any class in the same package.
(It is default when it is not public nor private)

DATA FIELD ENCAPSULATION :

Declaring data field as private, to protect data from being changed in an invalid way.

- To make a private data field accessible, provide a get to return its value :

public returntype getPropertyName ()

- To enable a private data field to be updated, provide a set method to set a new value :

public void setPropertyName (datatype propertyName)

METHODS TO OUTPUT DATA FIELDS :

- print method :

public void print()

- toString method :

public String toString()

PASSING OBJECTS TO METHODS :

- byte, short, int, long, float, double, char, and boolean are called primitive data types.
- classes such as String, Date, Circle, .. are called reference type.
- reference type variables are passed by reference (the reference to the object is passed to the parameter).

RK :

Arrays are objects in Java, they are passed by reference.

Example :

Assume that Class Circle is already defined :

```
public class Application {
 public static void main (String [] args) {
  Circle c1 = new Circle(5);
System.out.println("The radius is "+c1.getRadius());
  updateCircle(c1);
System.out.println("The radius is now "+c1.getRadius());
 }
 public static void updateCircle(Circle c) {
c.setRadius(c.getRadius()+1);
 }}
```

RK :

If two different functions are implemented in the same class as in the above example (main and updateCircle), functions must be both static.

ARRAY OF OBJECTS :

- An array of objects is an array of reference variables.
- When an array of objects is created, its elements are assigned the value of null.

Example :

Assume the class Circle is defined, create and initialize an array of type Circle.

```
import java.util.Random;

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

 Random r = new Random();
 Circle[] arrayCircle = new Circle[10];
 for(int i=0; i<arrayCircle.length; i++) {
  arrayCircle[i] = new Circle(r.nextDouble()*10);
System.out.println("Circle "+i+" of radius =
"+arrayCircle[i].getRadius());
System.out.println("Its area is "+arrayCircle[i].getArea());
  }

  }
}
```

THE THIS REFERENCE :

- “this” keyword is the name of a reference that refers to an object itself.
- Usage of “this”:
  - to reference a class's hidden data fields.

```
public class Ex {
private int i = 5;
public void setI (int i) {
this.i = i;  //this.i is the data field and i is the parameter
}
}
```

  - to enable a constructor to invoke another constructor of the same class.

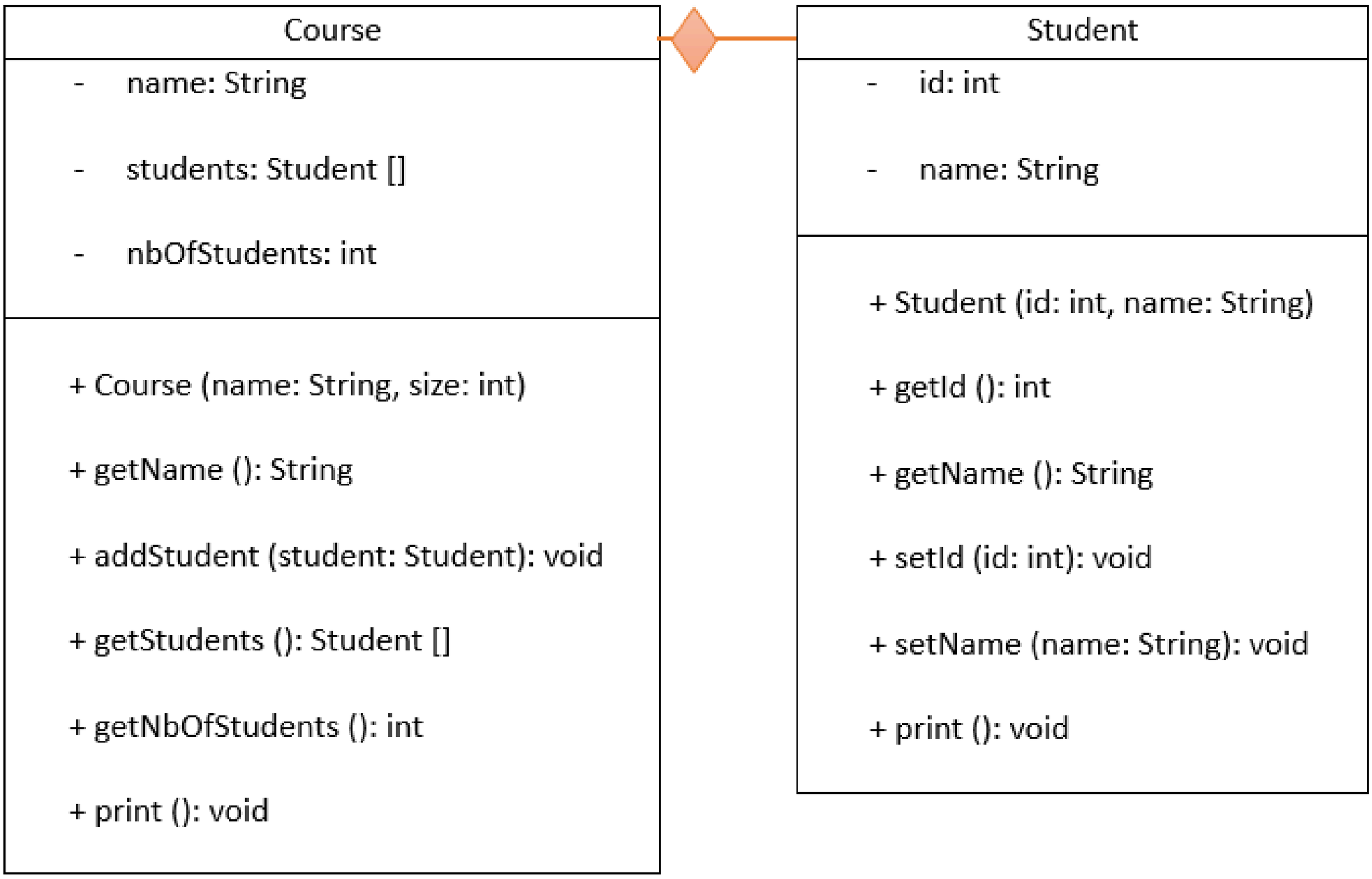
```
public class Circle {
private double radius;
public Circle (double radius) {
this.radius = radius;}
public Circle () {
this (1.0); //used to call the previous constructor
}
public double getArea () {
return radius*radius*Math.PI;
 }}
```

OBJECT COMPOSITION :

- An object can contain another object.
- The relationship btw the two is called composition, or 'has a' relationship.

Example:

- UML Diagram containing composition :



- **Solution :**

  - Student Class :

```
public class Student {

private int id;
private String name;

public Student (int id, String name) {
this.id = id;
this.name = name;
}
public int getId () {
return id;
}
public String getName () {
return name;
}
public void setId (int id) {
this.id = id;
}
public void setName (String name) {
this.name = name;
}
public void print () {
System.out.println("id = "+id+", name is "+name);
}
}
```

- Course Class :

```
public class Course {
private String name;
private Student [] students; // this statement creates the
composition relationship
private int nbOfStudents;
public Course (String name, int size) {
this.name = name;
nbOfStudents = 0;
students = new Student[size];
}
public void addStudent(Student student) {
if(nbOfStudents == students.length) System.out.println("Cannot
add more students, this section is full");
else{ students[nbOfStudents]=student;
nbOfStudents++;
 }
}
public Student [] getStudents() {
return students;
}
public int getNbOfStudents() {
return nbOfStudents;
}
public String getName() {
return name;
}
public void print() {
System.out.println("The course name is "+name);
System.out.println("The number of students is "+nbOfStudents);
System.out.println("The students registered in this course are :");
for(int i=0;i<nbOfStudents; i++)
students[i].print(); //calling print() of Student class
}
}
```

- Test Code :

```
public class Test {
public static void main (String [] args) {
Course course1 = new Course("Object-Oriented
Programming", 5);
Student s1 = new Student(1, "A");
Student s2 = new Student(2, "B");
Student s3 = new Student(3, "C");
Student s4 = new Student(4, "D");
course1.addStudent(s1);
course1.addStudent(s2);
course1.addStudent(s3);
course1.addStudent(s4);
Student s5 = new Student(5, "E");
Student s6 = new Student(6, "F");
course1.addStudent(s5);
course1.addStudent(s6); /* this student will not be
added since the array is full (specified size is 5) */
course1.print();
}
}
```

CHAPTER 02

JAVA CLASSES

THE OBJECT CLASS :

- Every class in Java is a subclass of the class Object found in the java library in the lang package.
- If no inheritance is specified when a class is defined, the superclass of the class is Object by default.
- The toString method in the Object class :
  public String toString()
- If no toString() is defined in a class, then a call to toString() will call the toString() of its superclass Object.

THE OBJECT EQUALS() METHOD :

- A method defined in the Object class.
- It compares the contents of two objects.
- Its default implementation in the Object class :

```
public boolean equals (Object obj) {
return (this == obj);
}
```

- When overriding it :

```
@Override
public boolean equals(Object o) {
if(o instanceof Circle)
  return radius == ((Circle)o).radius;
return false;
}
```

- **Classes from the Java Library :**

Java API (Application Programming Interface) contains set of classes for developing Java programs.

- Check the Date Class :

java.util.Date	
+ Date ()	- Constructs a Date object for the current time
+ Date (elapseTime: long)	- Constructs a Date object for a given time, in ms
+ toString (): String	- Returns a string representing the date and time
+ getTime (): long	- Returns the number of milliseconds
+ setTime (elapseTime: long): void	- Set a new elapse time in the object

- Check the Random Class :

java.util.Random	
+ Random ()	- Constructs a Random object with the current time as its seed
+ Random (seed: long)	- Constructs a Random object with a specified seed
+ nextInt (): int	- Returns a random int value
+ nextInt (n: int): int	- Returns a random int value between 0 and n (exclusive)
+ nextLong (): long	- Returns a random long value
+ nextDouble (): double	- Returns a random double value btw 0.0 and 1.0 (exclusive)
+ nextFloat (): float	- Returns a random float value btw 0.0F and 1.0F (exclusive)
+ nextBoolean (): boolean	- Returns a random boolean value

- **Date and Random Classes Example :**
  - Employee Class :

```
import java.util.Date;
public class Employee {
int id;
String name;
Date hiringDate;
public Employee (int i, String n) {
id = i;
name = n;
hiringDate = new Date();
}}
```

  - Test Code :

```
import java.util.Date;
import java.util.Random;
import java.util.Scanner;// for user input
public class Test {
 public static void main (String [] args) {
  Scanner input = new Scanner(System.in);
//creating an object of class Scanner to allow user input
  Random r = new Random();
  String employeeName; // entered by the user
  int employeeId; // generated randomly
  Employee e1;
System.out.println("Enter the name of the employee:");
  employeeName = input.nextLine();
//nextLine() is used for String inputs
  employeeId = r.nextInt(10)+1;
// r.nextInt(10) generates random number btw 0-9, +1 shifts the
range to 1-10
  e1 = new Employee(employeeId, employeeName);
System.out.println("ID: "+e1.id+" ,name: "+e1.name+" ,hiring
date: "+e1.hiringDate.toString());
  }}
```

STRING CLASS :

Two ways to construct a String :

- ***String s = new String("Hello Students");***
- ***String s = "Hello Students";***

A String object is immutable, its content cannot be changed.

- **String Comparison :**

Ex :

```
String s1 = new String ("Welcome");
String s2 = "Welcome";
String s3 = s1;
if(s1 == s2)
// false because they refer to different objects
if(s1 == s3)
// true because both references point to the same object
```

- **Obtaining Substrings :**

java.lang.String
+ length (): int **Returns number of characters in this string** + charAt (index: int): char **Returns the character at the specified index** + concat (s1: String): String **Returns a new string that concatenate this string with string s1** + substring (beginIndex: int): String **Returns this string's substring that begins at the specified beginIndex and extends to the end of the string** + substring (beginIndex: int, endIndex: int): String **Returns this string's substring that begins at the specified beginIndex and extends to the character at index: endIndex-1**

- **Converting, replacing, and splitting Strings :**

java.lang.String
+ toLowerCase (): String **Returns a new string with all characters converted to lowercase**
+ toUpperCase (): String **Returns a new String with all characters converted to upper case**
+ trim (): String **Returns a new string with blank characters trimmed on both sides**
+ replaceAll (oldString: String, newSting: String): String **Returns a new string that replaces all matching substrings in this string with new substrings**
+ replaceFirst (oldString: String, newSting: String): String **Returns a new string that replaces the first matching substring in this string with a new substring**
+ split (delimiter: String): String [] **Returns an array of strings consisting of the substrings split by the delimiter**

- **String Indexes :**

java.lang.String
+ indexOf (ch: char): int **Returns the index of the first occurrence of ch in the string. Returns -1 if not matched**
+ indexOf (ch: char, fromIndex: int): int **Returns the index of the first occurrence of ch from the specified index in the string. Returns -1 if not matched**
+ indexOf (s: String): int
+ indexOf (s: String, fromIndex: int): int
+ lastIndexOf (ch: char): int **Returns the index of the last occurrence of ch in the string. Returns -1 if not matched**
+ lastIndexOf (ch: char, fromIndex: int): int
+ lastIndexOf (s: String): int
+ lastIndexOf (s: String, fromIndex: int): int
+ toCharArray (): char [] **Converts this string to a new character array**

- **Converting Characters, Array of Characters, and Numeric Values to Strings :**

java.lang.String
+ valueOf (c: char): String **Returns a string consisting of character c** + valueOf (data: char []): String **Returns a string consisting of the characters in the array** + valueOf (d: double): String **Returns a string consisting the double value** + valueOf (f: float): String + valueOf (i: int): String + valueOf (l: long): String + valueOf (b: boolean): String

- **Converting String to Integer and Double :**

Integer.parseInt(String s)
Double.parseDouble(String s)

- **Question :**

Write a method that counts the number of letters in a string using the following header :

public static int countLetters(String s)

Write a test program that prompts the user to enter a string and displays the number of letters in the string.

- **Solution :**

```
import java.util.Scanner;
import java.lang.String;

public class Test {

public static void main (String [] args) {
Scanner input = new Scanner(System.in);
System.out.println("Enter a string: ");
String s = input.nextLine();
System.out.println("Number of letters in "+s+" is
"+countLetters(s));
}

public static int countLetters(String s) {
int count =0;for(int i=0; i<s.length(); i++)
if((s.charAt(i)>='A' && s.charAt(i)<='Z') || (s.charAt(i)>='a'
&& s.charAt(i)<='z'))
count++;return count;
}

}
```

- **Extra Exercise :**

Implement a class named "Triangle" to represent a triangle.
The class contains :

- Three private int data fields named side1, side2, and side3 that represents the three sides of a triangle with default values 3, 4, and 5.
- A no-arg constructor that creates a triangle with default values.
- A constructor that creates a triangle with the specified sides.
- A static method isValid() that takes as parameters three integer values and returns true if these values form a valid triangle and false otherwise. Note that the sum of every two sides of a triangle must be greater than the third one.
- A boolean method isIsosceles() that checks whether the triangle is isosceles or not (two sides are equal).
- A boolean method isRight() that checks whether the triangle is a right triangle or not. A triangle is right if it meets the Pythagorean Theorem.
- A boolean method isEquilateral() that checks whether the triangle is equilateral or not. (all sides are equal)
- A method getPerimeter() that returns the perimeter of a triangle.
- A method toString() that returns a String representation of the triangle. For example, if the triangle has side lengths of 3, 4, and 5, then this method will return the String "(3, 4, 5)"

Write a Client class to test the Triangle class as follows :

- Create an array of triangles which size is set by the user.
- The sides of each triangle in the array are generated randomly such that the length of each side is between 1 and 10 inclusive. Only valid triangles are accepted so the sides of each triangle is repeatedly generated until a valid one is found.
- Your program should display each of the generated triangles, its perimeter and its type: Isosceles, Equilateral, or Scalene (none of the sides are equal to each other) and whether it is right or not.

- **Solution :**
  - **Triangle Class :**

```
public class Triangle {
private int side1, side2, side3;
public Triangle () {
side1 = 3;
side2 = 4;
side3 = 5;}
public Triangle (int s1, int s2, int s3) {
side1 = s1;
side2 = s2;
side3 = s3;}
public static boolean isValid (int a, int b, int c) {
// isValid is static because it is invoked by the class itself
if((a + b > c) && (a + c > b) && (b + c > a))
   return true;
return false;}
public boolean isRight() {
if((Math.pow(side1, 2) + Math.pow(side2, 2) == Math.pow(side3, 2)) ||
(Math.pow(side1, 2) + Math.pow(side3, 2) == Math.pow(side2, 2)) ||
(Math.pow(side2, 2) + Math.pow(side3, 2) == Math.pow(side1, 2)))
  return true;
return false;}
public boolean isIsosceles() {
if((side1 == side2) || (side1 == side3) || (side2 == side3))
  return true;
return false;}
public boolean isEquilateral() {
if(side1 == side2 && side2 == side3)
  return true;
return false;}
public int getPerimeter() {
return side1 + side2 + side3;}
public String toString() {
return "("+side1+","+side2+","+side3+")"; }}
```

- Client Class :

```
import java.util.Scanner;
import java.util.Random;

public class Client {
public static void main (String [] args) {
Scanner input = new Scanner(System.in);
System.out.println("How many triangles do you want to generate ?");
int N = input.nextInt();
Triangle [] triangles = new Triangle[N];
int k = 0;
Random r = new Random();
while(k < N){
int a = r.nextInt(10)+1;
int b = r.nextInt(10)+1;
int c = r.nextInt(10)+1;
if(Triangle.isValid(a, b, c)){
triangles[k] = new Triangle(a, b, c);
k++;
}}
System.out.println("Sides \t Perimeter \t Type");
for(int i=0; i<N; i++) {
System.out.print(triangles[i].toString()+ "\t"+
triangles[i].getPerimeter()+ "\t");
if(triangles[i].isIsosceles())
 System.out.print("Isosceles");
else
 if(triangles[i].isEquilateral())
   System.out.print("Equilateral");
else
  System.out.print("Scalene");
  System.out.print("\t");
if(triangles[i].isRight())
System.out.print("Right");
System.out.print("\n");
}
}}
```

ARRAYLIST CLASS :

- An arraylist can be used to store a list of objects with unlimited number of objects.
- Non-generic array list : stores any type of objects in it :
  ArrayList list = new ArrayList();
- Generic array list : specify a concrete type when creating an array list :

Ex :

ArrayList<String> list = new ArrayList<String>();

// **or**

ArrayList<String> list = new ArrayList<>();

// **or**

ArrayList<String> list = new ArrayList();

- Array Vs. ArrayList :

Operation	Array	ArrayList
Creating	String [] a = new String [10]	ArrayList<String> list = new ArrayList <> ()
Accessing an element	a [index]	list.get (index)
Updating an element	a [index] = “London”	list.set (index, “London”)
Returning size	a.length	list.size()
Adding a new element		list.add (“London”)
Inserting a new element		list.add (index, “London”)
Removing an element		list.remove (index)
Removing an element		list.remove (Object)
Removing all elements		list.clear ()

- **ArrayList methods :**

java.util.ArrayList
+ ArrayList () **Creates an empty list** + add (o: E): void **Appends a new element o at the end of the list** + add (index: int, o: E): void **Appends a new element o at the specified index in the list** + clear (): void **Removes all elements from this list** + contains (o: Object): boolean **Returns true if the list contains the element o** + get (index: int): E **Returns the element at the specified index** + indexOf (o: Object): int **Returns the index of the first matching element in the list** + isEmpty (): boolean **Returns true if this list contains no elements** + lastIndexOf (o: Object): int **Returns the index of the last matching element in this list** + remove (o: Object): boolean **Removes the element o from this list** + remove (index: int): Object **Removes the element at the specified index** + size (): int **Returns the number of elements in the list** + set (index: int, o: E): E **Sets the element at the specified index**

CHAPTER 03

INHERITANCE

~~Chapter 03 Inheritance~~

INHERITANCE:

- Inheritance enables defining a general class (super/ parent/ base class) and later extend it to more specialized classes (subclasses/ child/ extended/ derived class).
- super and subclasses : c1 is a class extended from another class c2, then c1 is a subclass of superclass c2.
- A subclass inherits accessible data fields and methods from its superclass. It may add new data fields, add new methods, or override the superclass's methods.

RK:

Subclass declaration syntax is as follows :

public class Subclass extends Superclass

Example :

- **Superclass**: GeometricObject
- **Subclasses**: Circle, Rectangle

- GeometricObject Class :

```
public class GeometricObject {
private String color;
private boolean filled;
public GeometricObject () {
color = "white";
filled = false;
}
public GeometricObject (String c, boolean f) {
color = c;
filled = f;
}
public String getColor () {
return color;
}
public void setColor (String c) {
color = c;
}
public boolean isFilled () {
return filled;
}
public void setFilled (boolean f) {
filled = f;
}
@Override
public String toString () {
String s = "The color is " + color;
if(filled)
 s+= "\n The geometric object is filled";
else
 s+= "\n The geometric object is not filled";
return s;
}}
```

- Circle Class :

```
public class Circle extends GeometricObject {
private double radius;
public Circle (String c, boolean f, double r) {
setColor(c);   /* Using setter methods in the constructor
to initialize inherited properties */
setFilled(f);
radius = r;
}
public double getRadius () {
return radius;
}
public void setRadius (double radius) {
this.radius = radius;
}
public double area () {
return radius*radius*Math.PI;
}
@Override
public String toString () {
String s = "The color is " + getColor();
if(isFilled())
 s+= "\n The circle is filled";
else
 s+= "\n The circle is not filled";
s+= "\n The radius is " + radius;
return s;
}}
```

- Rectangle Class :

```
public class Rectangle extends GeometricObject {
private double width;
private double height;
public Rectangle (String c, boolean f, double width, double height){
setColor(c);
setFilled(f);
this.width = width;
this.height = height;}
public double getWidth () {
return width;}
public void setWidth (double width) {
this.width = width;}
public double getHeight () {
return height;}
public void setHeight (double height) {
this.height = height;}
public double area () {
return width*height;}
public double perimeter () {
return 2*(width + height);}
@Override
public String toString () {
String s = "The color is " + getColor();
if(isFilled())
 s+= "\n The rectangle is filled";
else
 s+= "\n The rectangle is not filled";
s+= "\n The height is " + height;
s+= "\n The width is " + width;
return s;
}}
```

SUPER KEYWORD :

- “super” refers to the superclass.
- It can be used in the subclass to call a superclass constructor or to call a superclass method.
- Superclass constructors are not inherited by the subclass. Can only be invoked from the subclass constructors using the keyword “super”.
- If “super” is not used, the superclass no-arg constructor is automatically invoked in the subclass constructor.

Syntax:

- ***super()*** **:** invoking no-arg constructor.
- ***super(parameter)*** **:** invoking arg constructor.

Example :

Regarding Circle subclass of superclass GeometricObject, use super keyword :

```
// constructor
public Circle (String c, boolean f, double r) {
super(c, f);
radius = r;
}
//method
public String toString () {
return super.toString() + "\n The radius is " + radius;
}
```

CONSTRUCTOR CHAINING :

Constructor chaining is constructing an instance of a class that invokes all the super classes' constructors along the inheritance.

Example :

```
class Faculty extends Employee {
public static void main (String [] args) {
new Faculty();}
public Faculty () {
System.out.println("Faculty's no-arg constructor is invoked");}
}
class Employee extends Person {
public Employee () {
this("Invoke Employee's overloaded constructor");
System.out.println("Employee's no-arg constructor is invoked");}
public Employee (String s) {
System.out.println(s);}
}
class Person {
public Person () {
System.out.println("Person's no-arg constructor is invoked");}
}
```

Steps of Execution of the code :

- Start from the main method.
- Invoke Faculty() constructor.
- Invoke Employee() no-arg constructor.
- Invoke Employee(String s) constructor.
- Invoke Person() constructor.
- Execute the print statement inside the Person() constructor.
- Execute the print statement inside the Employee(String s) constructor.
- Execute the print statement inside the Employee() no-arg constructor.
- Execute the print statement inside the Faculty() constructor.

OVERRIDING METHODS :

- Overriding is when the subclass modify the implementation of a method defined in the superclass.
- An instance method can be overridden only if it is accessible (private method cannot be overridden because it is not accessible outside its own class).
- A static method can be inherited but cannot be overridden.

Example :

```
public class Circle extends GeometricObject {
//
@Override
public String toString () {
return super.toString() + "\n The radius is " + radius;
}
}
```

OVERLOADING METHODS :

- Overloading is when two methods have the same name but different signatures (parameter lists) within one class.
- When modifiers or returned types are different, methods are not overloaded. Overloaded methods must have different parameter lists.

- Parameter lists are different when :
    - The type of parameters is different

```
public static int max (int num1, int num2)
public static double max (double num1, double num2)
```

    - The number of parameters is different

```
public static int sum (int num1, int num2)
public static int sum (int num1, int num2, int num3)
```

    - The order of parameters is different

```
public static void method (int num, char ch)
public static void method (char ch, int num)
```

Note :

Overridden methods are in different classes related by inheritance. While, overloaded methods can be either in the same class or in different classes related by inheritance.

~~Chapter 03 Inheritance~~

Question 1 :

An isogram is a word that contains no repeated letters. For example, the word "computer" is an isogram because each letter in the word appears exactly once, but the word "banana' is not because 'a' and 'n' are repeated.
Write a boolean isIsogram method that accepts as input a String containing a single word, then returns true if the word is an isogram and false otherwise. Assume the word consists of only letters (no digits and no other characters such as spaces and punctuation), but it might contain both upper-case and lower-case letters. Write also a main method to test the method isIsogram.

Solution:

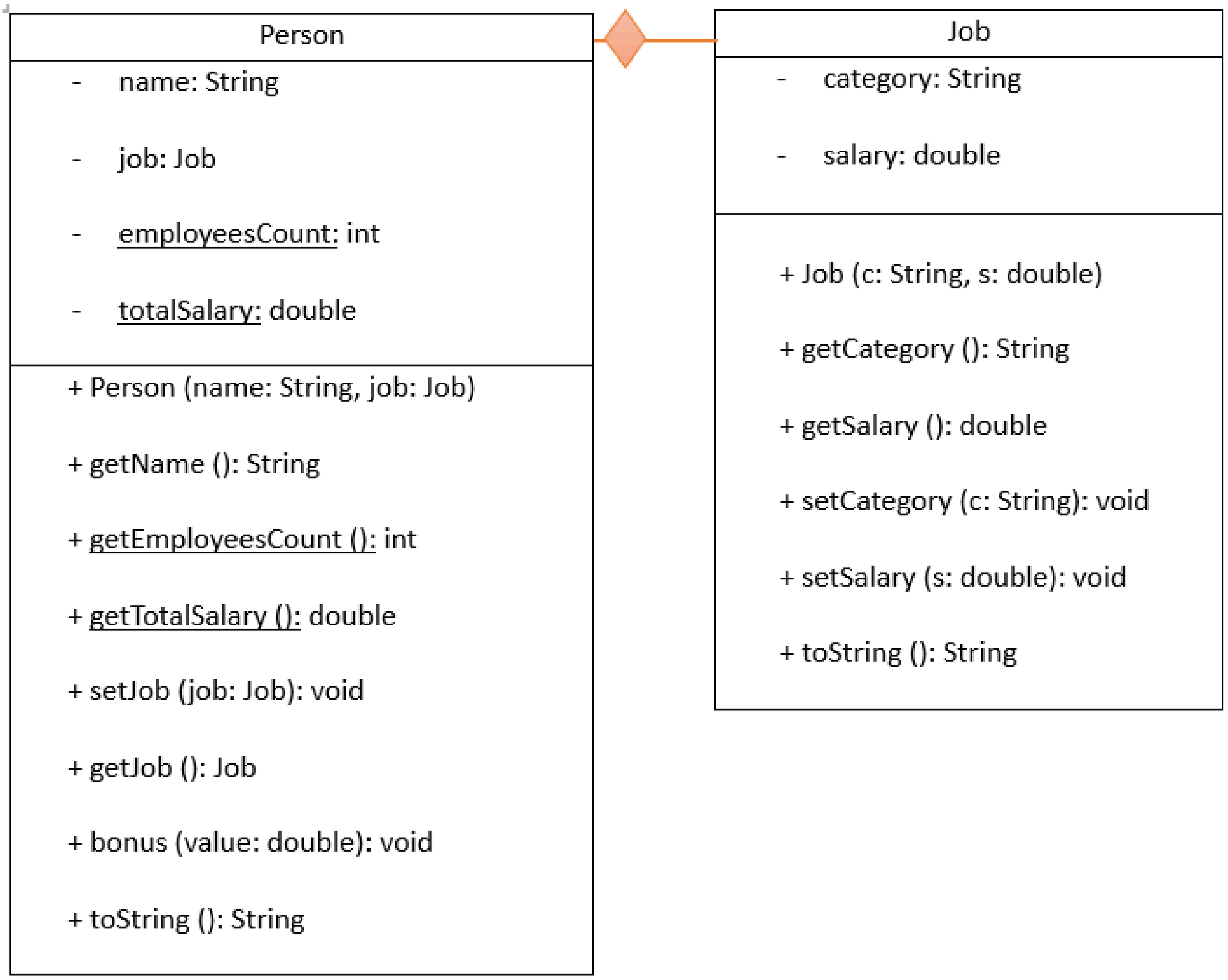
```
import java.util.Scanner;
import java.lang.String;


public class Isogram {
public static boolean isIsogram (String s) {
String c = s.toLowerCase();
for(int i=0; i<c.length(); i++){
 for(int j=i+1; j<c.length(); j++){
 if(c.charAt(i)==c.charAt(j))
  return false;}}
return true;}
public static void main (String [] args) {
Scanner input = new Scanner(System.in);
System.out.println("Enter a single word:");
String s = input.next();
if(isIsogram(s))
 System.out.println(s + " is an isogram");
else
 System.out.println(s + " is not an isogram");
}
}
```

Question 2:

- Implement the Job class with all the data fields and methods stated in the UML. The toString() method returns a String description of the job.
- Implement the Person class with all the data fields and methods stated in the UML.
  - Initialize the static data fields employeesCount and totalSalary to 0.
  - The constructor must increment by 1 the employeesCount and add the salary of the job to the totalSalary.
  - The setJob method subtracts the old salary from totalSalary, updates the job, and adds the new salary to totalSalary.
  - The method bonus add the value to the salary of the person and add this value to the totalSalary.
  - The toString() method returns a full description of the Person with the related job.
- Write an application with a main Method :
  - Create an array of 4 persons with data fields' values given by the user.
  - Change the job of the the first person in the array to be "Eng. 1500" using setJob method.
  - Change the job category of the first person in the array to be "Engineer".
  - Display the actual salary of every person.
  - Display the average of all salaries of all persons (=totalSalary/employeesCount)
  - Display the name of the person with the maximum actual salary.
  - Add 100$ to the actual salary of all the person's work "Engineer".
  - Display a full description of all the persons.

Question 2:

Given the below UML diagram, implement the classes and the application.



- **Solution :**
  - Job Class :

```
public class Job {
private String category;
private double salary;
public Job (String c, double s) {
category = c;salary = s;
}
public String getCategory () {
return category;
}
public double getSalary () {
return salary;
}
public void setCategory (String c) {
category = c;
}
public void setSalary (double s) {
if (s >= 0)
 salary = s;
else
 System.out.println("Salary must be non-negative.");
}
@Override
public String toString () {
return "The job category is " + category + "\n The job salary
is " + salary;
}
}
```

- Person Class :

```
public class Person {
private String name;
private Job job;
private static int employeesCount = 0;
private static double totalSalary = 0;
public Person (String name, Job job) {
this.name = name;
this.job = job;
employeesCount++;
totalSalary+=job.getSalary();}
public String getName () {
return name;}
public static int getEmployeesCount () {
return employeesCount;}
public static double getTotalSalary () {
return totalSalary;}
public void setJob(Job job) {
totalSalary -= this.job.getSalary();
this.job = job;
totalSalary += job.getSalary();}
public Job getJob () {
return job;}
public void bonus (double value) {
job.setSalary(job.getSalary() + value);
totalSalary+=value;}
@Override
public String toString () {
return "Name:" + name + ",category:" + job.getCategory() +
",actual salary:" + job.getSalary() + ",total salaries:" +
totalSalary;}
}
```

- Application Code :

```
import java.util.Scanner;
public class Application {
public static void main (String [] args) {
Scanner input = new Scanner(System.in);
Job [] jobs = new Job[4];
Person [] persons = new Person[4];
for(int i=0; i<persons.length; i++){
System.out.println("Enter person information:");
String n = input.nextLine();
String c = input.next();
double s = input.nextDouble();
input.nextLine(); // consume leftover newline
jobs[i] = new Job(c, s);
persons[i] = new Person(n, jobs[i]); }
persons[0].setJob(new Job("Eng.", 1500));
persons[0].getJob().setCategory("Engineer");
System.out.println("The actual salary of every person:");
for(int i=0; i<persons.length; i++)
  System.out.print(persons[i].getJob().getSalary() +"\t");
System.out.println("\n The average of all salaries of all persons:" +
Person.getTotalSalary()/Person.getEmployeesCount());
double max = 0;
for(int i=0; i<persons.length; i++) {
if(persons[i].getJob().getSalary()>max)
 max = persons[i].getJob().getSalary();}
for(int i=0; i<persons.length; i++) {
if(persons[i].getJob().getSalary()==max)
 System.out.println("The person with the maximum actual salary is"
+ persons[i].getName());}
System.out.println("The description of all persons:");
for(int i=0; i<persons.length; i++){
if(persons[i].getJob().getCategory().equals("Engineer"))
 persons[i].bonus(100); System.out.println(persons[i].toString());}
}}
```

CHAPTER 04

POLYMORPHISM

POLYMORPHISM :

- Polymorphism means a reference from a superclass can refer to objects from different types provided, that these types are from subclasses of the superclass.

- **Example:**

```
GeometricObject g1;
g1 = new GeometricObject("Red", true);
// g1 refers to a GeometricObject object
// or
g1 = new Circle("Red", true, 1);
// g1 refers to a Circle object
// or
g1 = new Rectangle("Red", true, 5, 6);
// g1 refers to a Rectangle object
// g1 can refer to different types, it has many forms
```

- **Polymorphic array and polymorphic reference :**

GeometricObject [] g = new GeometricObject[120];

- g is called polymorphic array because g[i] can refer to an object of any of the three types : GeometricObject, Circle, or Rectangle.
- g[i] hence is called polymorphic reference.

DYNAMIC BINDING:

Binding is linking a method call to a specific method. It is dynamic because it happens during runtime.

Ex: ***g[i].toString();***

- g[i].toString() has many forms. It is polymorphic call.
- It is not the reference type that determines which toString() will be called, it is rather the object type.

CASTING OBJECTS AND THE INSTANCEOF OPERATOR :

- Casting object is when one object reference is type casted (converted) into another object reference.
- Ensure that the object is an instance of another object before attempting a casting, using the “instanceof” operator.

- **Example:**

```
Object O = new Student();
Student b = O; // false
Student b = (Student) O; // true


/* This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Students
Note that here the code is available, but indeed you don't have insights that Object O is of Student type */
```

- **Use the instanceOf operator to test whether an object is an instance of a class.**
    - **Example:**

```
public class CastingDemo {
public static void main (String [] args) {
GeometricObject g1, g2;
g1 = new Circle("Red", true, 1);
g2 = new Rectangle("Red", true, 5, 6);
displayObject(g1);
 displayObject(g2);
}
public static void displayObject(GeometricObject g) {
if(g instanceof Circle)
  System.out.println("Circle area is " +
((Circle)g).area());
else
 if(g instanceof Rectangle)
   System.out.println("Rectangle perimeter is " +
((Rectangle)g).perimeter());
}
}
```

THE PROTECTED MODIFIER :

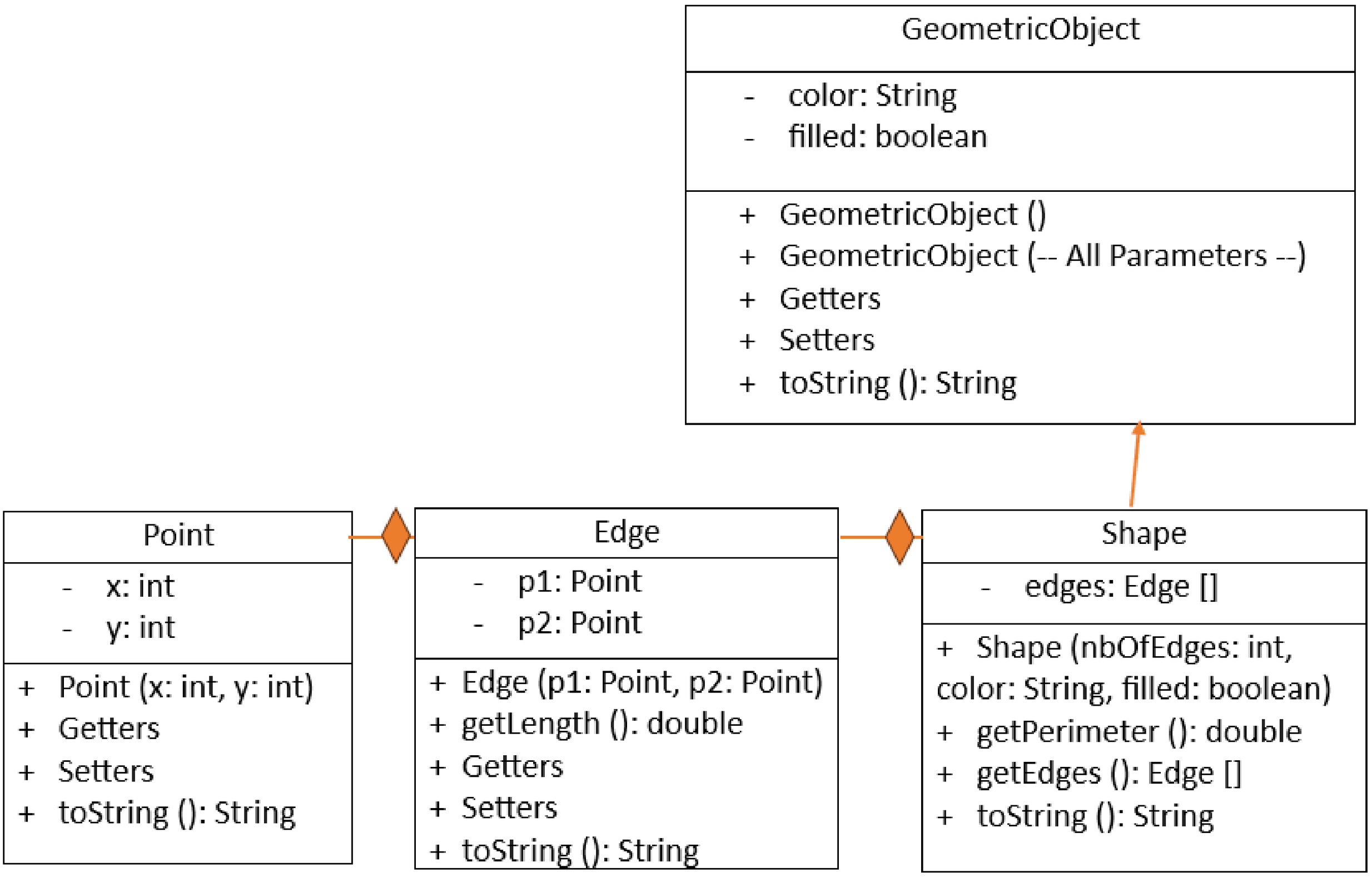
- A protected data field or method in a public class can be accessed by any class in the same package and by its subclasses, even if the subclasses are in different package.
- protected keyword is used to allow subclasses to access data fields and methods defined in the super class, but not to allow non-subclasses to access these data fields and methods.
- ' # ' sign in UML means protected.

Example:

```
public class GeometricObject {
protected String color;
protected boolean filled;
// constructors
public String getColor () {
// for classes not able to access the color data field
return color;}
 // other methods
}
public class Circle extends GeometricObject {
private double radius;
public Circle (String c, boolean f, double r) {
color = c; /* can be accessible in subclass since it is
protected in the superclass */
filled = f;
radius = r;}
// other methods }
public class App {
public static void main (String [] args) {
// remains the same}
}
```

Question :

Given the below UML diagram, implement the classes and the application.



- GeometricObject class : toString() describes the object such as : Color: Red Not Filled.
- Point class : toString() returns the point as follows: (3,6)
- Edge class:
  - The getLength method returns the distance between p1 and p2 : √ [(x - x) + (y - y)]
  - toString() returns the edge as follows: Edge {(5,6),(1,7)}

~~Chapter 04 Polymorphism~~

Question :

- Shape class :
  - The constructor creates the edges array given the nbOfEdges (size of the array) as well as the color and filled attributes. The constructor creates the shape from random points which coordinates are generated randomly between 0 and 20.
  - getPerimeter() returns the perimeter of the shape which is the sum of the lengths of all edges.
  - toString() returns a complete description of the shape as per the following example:
    - Shape with 2 Edges
    - Color: Red Not Filled
    - Edges:
      Edge { (3,15),(13,4) }
      Edge { (5,5),(15,11) }

- Application :
  - Create a shape object with nb of edges read from the user.
  - Display the description of the shape.
  - Write a method that takes as parameter a shape and return its longest edge length.
  - Write a method that takes as parameter a GeometricObject array and returns an array of shapes that includes all and only the shapes in the array. The size of the returned array must be exactly equal to the nb of shapes in the array. Invoke it in the main by passing to it the appropriate argument.

- **Solution :**
  - GeometricObject Class :

```
public class GeometricObject {
private String color;
private boolean filled;
public GeometricObject () {
this.color = "white";
this.filled = false;
}
public GeometricObject (String c, boolean f) {
color = c;filled = f;}
public String getColor () {
return color;
}
public void setColor (String c) {
color = c;
}
public boolean isFilled () {
return filled;}
public void setFilled (boolean f) {
filled = f;
}
@Override
public String toString () {
/* Use the ternary operator to return "filled" if the variable is
true, otherwise "not filled" */
return "Color:" + color + " " + (filled ? "filled" : "not filled");
}
}
```

- Point Class :

```
public class Point {
private int x;
private int y;
public Point() {
 this.x = 0;
 this.y = 0;
 }
public Point (int x, int y) {
this.x = x;
this.y = y;
}
public int getX () {
return x;
}
public int getY () {
return y;
}
public void setX (int x) {
this.x = x;
}
public void setY (int y) {
this.y = y;
}
@Override
public String toString () {
return "(" + this.x + "," + this.y + ")";
}
}
```

- Edge Class :

```
public class Edge {
private Point p1;
private Point p2;
public Edge (Point p1, Point p2) {
this.p1 = p1;
this.p2 = p2;}
public double getLength() {
int dx = p1.getX() - p2.getX();
int dy = p1.getY() - p2.getY();
return Math.sqrt(dx*dx + dy*dy);}
public Point getP1 () {
return p1;}
public void setP1 (Point p) {
this.p1 = p;}
public Point getP2 () {
return p2;}
public void setP2 (Point p) {
this.p2 = p;}
@Override
public String toString () {
return "Edge {" + p1.toString() + "," + p2.toString() +
"}\n";}
}
```

- Shape Class :

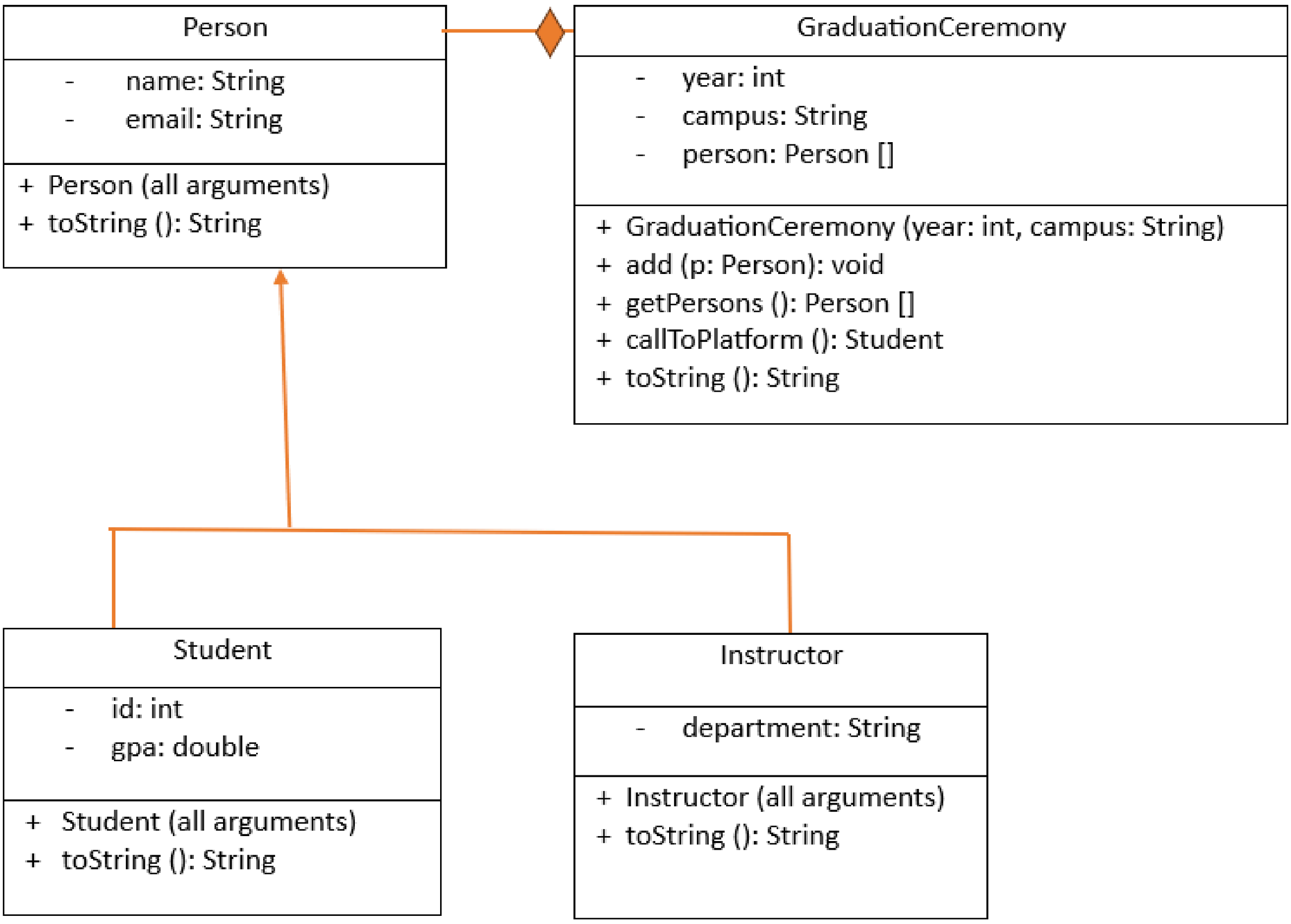
```
public class Shape extends GeometricObject {
private Edge [] edges;
public Shape (int nbOfEdges, String color, boolean filled) {
super(color, filled);
edges = new Edge[nbOfEdges];
for(int i=0; i<edges.length; i++) {
Point p1 = new Point((int)(Math.random()*21), (int)
(Math.random()*21));
Point p2 = new Point((int)(Math.random()*21), (int)
(Math.random()*21));
edges[i] = new Edge(p1, p2);}}
public double getPerimeter () {
double p = 0;
for(int i=0; i<edges.length; i++)
  p+= edges[i].getLength();
return p;}
public Edge [] getEdges () {
return edges;}
public String toString () {
String s = "Shape with " + edges.length + " edges\n" +
super.toString() + "\n Edges:\n";
for(int i=0; i<edges.length; i++)
  s+= edges[i].toString();
return s;}
}
```

- Application code :

```
import java.util.Scanner;
public class Application {
public static void main (String [] args) {
Scanner input = new Scanner(System.in);
System.out.println("Enter nb of edges:");
int nbOfEdges = input.nextInt();
Shape shape = new Shape(nbOfEdges, "Blue", true);
System.out.println(shape);
System.out.println("The longest edge in the previous shape:" +
getLongestEdge(shape));
GeometricObject [] g = new GeometricObject [10];
for(int i=0; i<g.length; i+=2) {
g[i] = new GeometricObject();
g[i+1] = new Shape(3+i, "Red", false);}
Shape [] s;s = getShape(g);
for(int i=0; i<s.length; i++)
 System.out.println(s[i]);}
public static double getLongestEdge(Shape shape) {
double max = 0;
for (Edge e : shape.getEdges()) {
   max = Math.max(max, e.getLength()); }
return max; }
public static Shape [] getShape (GeometricObject [] g) {
int nbOfShapeObjects = 0;
for(int i=0; i<g.length; i++) {
 if(g[i] instanceof Shape)
   nbOfShapeObjects++;}
Shape [] shapes = new Shape[nbOfShapeObjects];
for(int i=0, j=0; i<g.length; i++){
 if(g[i] instanceof Shape) {
    shapes[j] = (Shape)g[i];
    j++; }}
return shapes;}}
```

Question :

Given the below UML. For each class add the getters and setters



- Implement the class Person:
  - Implement the toString() method that returns a String including the name and email of the person.
- Implement the class Student:
  - Override the toString() method that returns a String including the id, name, and email of the student followed by the String "Under Probation" if the gpa <2 or the String "Good Standing" if the gpa >2
- Implement the class Instructor:
  - Override the toString() method that returns a String including the name, email, and the department.

~~Chapter 04 Polymorphism~~

Question :

- Implement the class GraduationCeremony:
  - Implement a constructor to initialize the data year and campus, and create an array of persons of length 100. Don't use an Array List.
  - Implement the method add(p: Person) to add a person p to the ceremony in the first empty cell (containing null) in the array persons. The method will search for the first empty cell in the array.
  - Implement the method callToPlatform(): Student which returns the student s with the higher gpa in the persons array. The method assigns null to the array cell that contains s. If there are no students in the array, the method returns null.
  - Implement the toString() method that returns a String including the year of the ceremony, the total number of invited students, the detailed information for each student, the total number of invited instructors, and the detailed information for each instructor. Use only one loop.
- Application :
  - Create an object named gc of type GraduationCeremony. The year is 2018. Store in gc some students and some instructors. The user will decide whether he/she will store a student or an instructor in each cell. The data of these objects are to be read from the user. Keep reading values from the user until 'n' is entered. The user will decide how many persons will be added to gc.
  - Display the information of gc, the number of students in the gc having gpa >3.5, the name and department only of all instructors in the object gc.
- Call the students to the platform, one by one, until no more students are available.

- **Solution :**
  - Person class :

```
public class Person {
private String name;
private String email;
public Person (String name, String email) {
this.name = name;
this.email = email;
}
public String getName () {
return name;
}
public void setName (String name) {
this.name = name;
}
public String getEmail () {
return email;
}
public void setEmail (String email) {
this.email = email;
}
@Override
public String toString() {
return "Name: " + name + ", Email: " + email;
}
}
```

- Student class :

```
public class Student extends Person {
private int id;
private double gpa;
public Student (String n, String e, int id, double gpa) {
super(n, e);
this.id = id;
this.gpa = gpa;}
public int getId () {
return id;}
public void setId (int id) {
this.id = id;}
public double getGpa () {
return gpa;}
public void setGpa (double gpa) {
this.gpa = gpa;}
public String toString () {
String status = (gpa <2) ? "Under Probation" : "Good Standing";
return super.toString() + "," + id + "," + status;}}
```

- Instructor class :

```
public class Instructor extends Person {
private String department;
public Instructor (String name, String email, String department) {
super(name, email);
this.department = department;}
public void setDepartment (String department) {
this.department = department;}
public String getDepartment () {
return department;}
public String toString () {
return super.toString() + "," + department;}
}
```

- GraduationCeremony class :

```
public class GraduationCeremony {
private int year;
private String campus;
private Person [] persons;
private int numStudents;
private int numInstructors;
public GraduationCeremony (int year, String campus) {
this.year = year;
this.campus = campus;
this.persons = new Person[100];
this.numStudents = 0;
this.numInstructors = 0;}
public int getYear () {
return year;}
public void setYear (int year) {
this.year = year;}
public String getCampus () {
return campus;}
public void setCampus (String campus) {
this.campus = campus;}
public int getNumInstructors () {
return numInstructors;}
public int getNumStudents () {
return numStudents;}
public Person [] getPersons () {
return persons;}
public void add(Person p) {
int index = numStudents + numInstructors;
if(index >= persons.length) {
 System.out.println("Cannot add more persons: array is full.");
 return; }
persons[index] = p;
if(p instanceof Student) {
 numStudents++; }
else if (p instanceof Instructor) {
 numInstructors++; } }
// The code is continued on the next page
```

- GraduationCeremony class :

```
// continuing the previous code

public Student callToPlatform () {
double max = 0;
int maxIndex = -1; // since array indexing starts at 0
for(int i=0; i<persons.length; i++) {
 if(persons[i] instanceof Student) {
  if(((Student)persons[i]).getGpa() > max) {
  max = ((Student)persons[i]).getGpa();
  maxIndex = i; }}}
if(maxIndex != -1) {
Student s = (Student)persons[maxIndex];
persons[maxIndex] = null;
numStudents--;
return s;}
else
return null;}
public String toString () {
String s = "Graduation Ceremony of " + year;
s+= "\n Total number of invited students is: " + numStudents;
s+= "\n total number of invited instructors is: "+ numInstructors;
for(int i=0; i<persons.length; i++) {
 if(persons[i] instanceof Student)
  s+= ((Student)persons[i]).toString() + "\n"; }
for(int i=0; i<persons.length; i++) {
 if(persons[i] instanceof Instructor)
  s+= ((Instructor)persons[i]).toString() + "\n"; }
return s;}
}
```

- Application code :

```
import java.util.Scanner;
public class Application {
public static void main (String [] args) {
Scanner input = new Scanner(System.in);
System.out.print("Enter the campus:");
String campus = input.nextLine();
GraduationCeremony gc = new GraduationCeremony (2018,
campus);
char c;
do{
System.out.print("Add a person (s for student, i for
instructor, n to stop):");
c = input.nextLine().charAt(0);
if(c=='s') {
System.out.print("Enter student name:");
String name = input.nextLine();
System.out.print("Enter student email:");
String email = input.nextLine();
System.out.print("Enter ID:");
int id = input.nextInt();
System.out.print("Enter GPA:");
double gpa = input.nextDouble();
input.nextLine();  /*Consume the leftover newline character
after nextDouble() to avoid input issues in the next input*/
gc.add(new Student(name, email, id, gpa));}

// The code continued on the next page
```

- Application code :

```
// continuing the previous code

else
 if(c=='i') {
System.out.print("Enter instructor name:");
 String name = input.nextLine();
 System.out.print("Enter email:");
 String email = input.nextLine();
 System.out.print("Enter department:");
 String department = input.nextLine();
 gc.add(new Instructor(name, email, department)); }
}while(c != 'n');
System.out.print(gc);
int count=0;
for(int i=0; i<gc.getPersons().length; i++) {
 if(gc.getPersons()[i] != null && gc.getPersons()[i] instanceof
Student)
  if(((Student)gc.getPersons()[i]).getGpa() >3.5)
   count++;}
System.out.println("nb of students with GPA > 3.5:" + count);
System.out.print("Instructors:");
for(int i=0;i<gc.getPersons().length; i++)
 if(gc.getPersons()[i] instanceof Instructor)
  System.out.println(((Instructor)gc.getPersons()
[i]).getName() + "," + ((Instructor)gc.getPersons()
[i]).getDepartment());
System.out.println("Calling students to platform:");
Student student;
while((student = gc.callToPlatform()) != null)
 System.out.println(student.getName() + " called to the
platform");}
}
```

CHAPTER 05

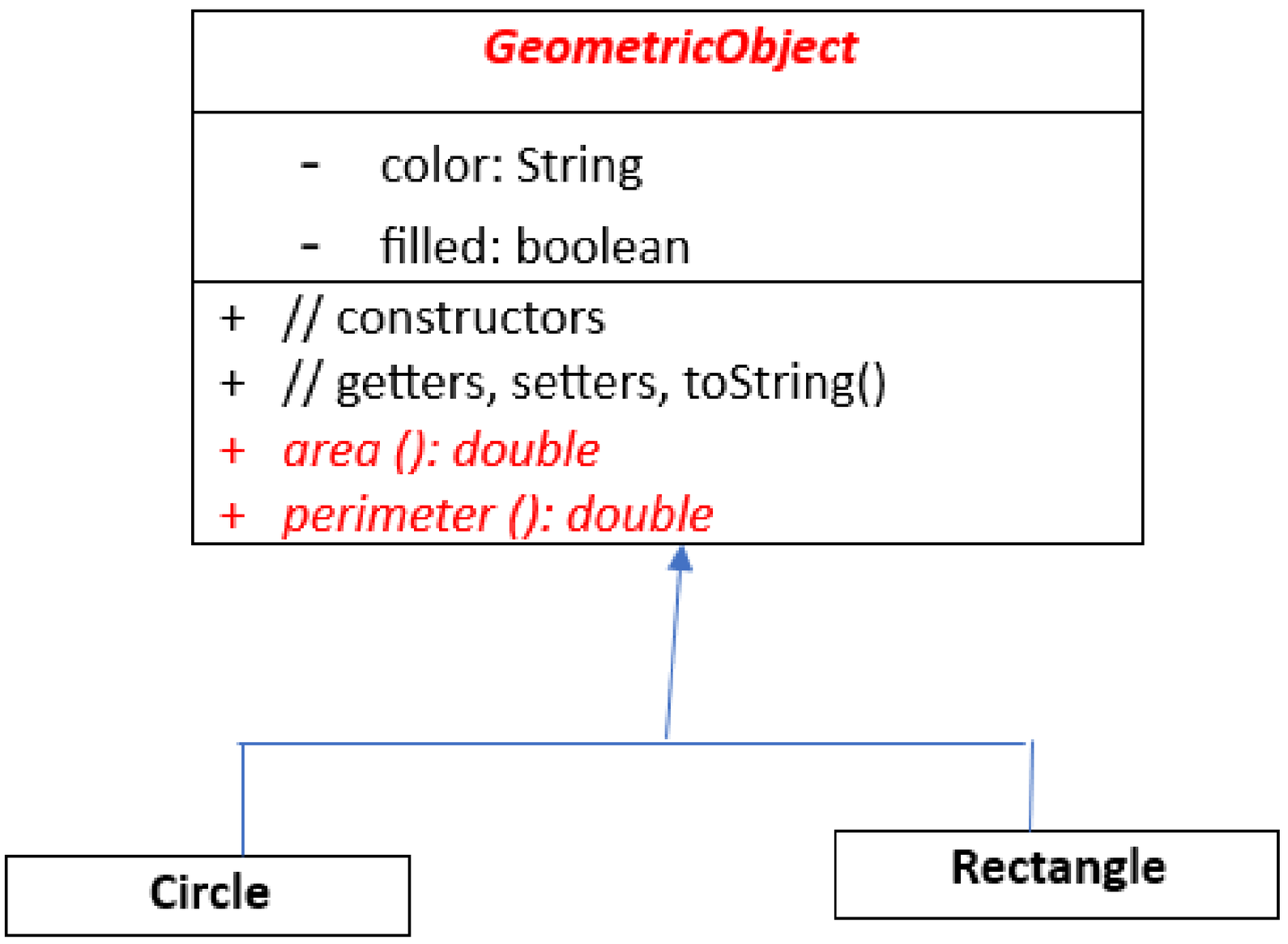
ABSTRACT VS INTERFACE

~~Chapter 05 Abstract Vs Interface~~

ABSTRACT CLASS

- An abstract class cannot be used to create objects.
- Abstract classes are denoted using the abstract modifier in the class header.
- An abstract class can contain abstract methods which are implemented in concrete (non-abstract) subclasses.
- In a non-abstract subclass extended from an abstract class, all the abstract methods must be implemented. If a subclass of an abstract superclass doesn't implement all the abstract methods, the subclass must be defined abstract.
- A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods.
- An abstract class can be used as a datatype.
- In UML, the names of abstract classes and their abstract methods are italicized.

EX:



In the above UML, methods *area()* and *perimeter()* are overridden in Circle and Rectangle classes.

- Geometric abstract class :

```
public abstract class GeometricObject {
// attributes
// constructors
// other methods
public abstract double area();
public abstract double perimeter();
}
```

- Circle class :

```
public class Circle extends GeometricObject {
 private double radius;
// Constructors, Getters, Setters
 // Implement abstract methods
public double area() {
return Math.PI*radius*radius;}
public double perimeter() {
return 2*Math.PI*radius;}
}
```

- Rectangle class :

```
public class Rectangle extends GeometricObject {
private double width;
private double height;
// Constructors, Getters, Setters
 // Implement abstract methods
public double area() {
return width*height;}
public double perimeter() {
return 2*(width + height);}
}
```

The advantage of having the abstract methods *area()* and *perimeter()* in the superclass *GeometricObject i*s to benefit from the idea of polymorphic call and dynamic binding to make the code simpler.

INTERFACES :

- An interface is a class that contains only constants and abstract methods.
- In an interface, all data fields are public final static and all methods are public abstract. For this reason, these modifiers can be omitted.
- A constant defined in an interface can be accessed using syntax : InterfaceName.CONSTANT_NAME, since it is static.
- An instance cannot be created from an interface.
- An interface is similar to abstract class, but the intent of an interface is to specify a behavior for objects.

Syntax :

public interface InterfaceName {
constant declarations;
method signatures; }

- An interface can be used as a datatype for a variable. A variable of an interface type can reference any instance of the class that implements the interface.
- An interface can extend other interfaces not classes.
- A class can extend one superclass and implement multiple interfaces.
- If a class extends an interface, this interface plays the same role as a superclass.
- In UML, the name of interface is italicized and is preceded by <<interface>>. A dashed line represent the " implements " relationship between the class and the interface. This is interface inheritance.

- **Abstract Vs Interface :**

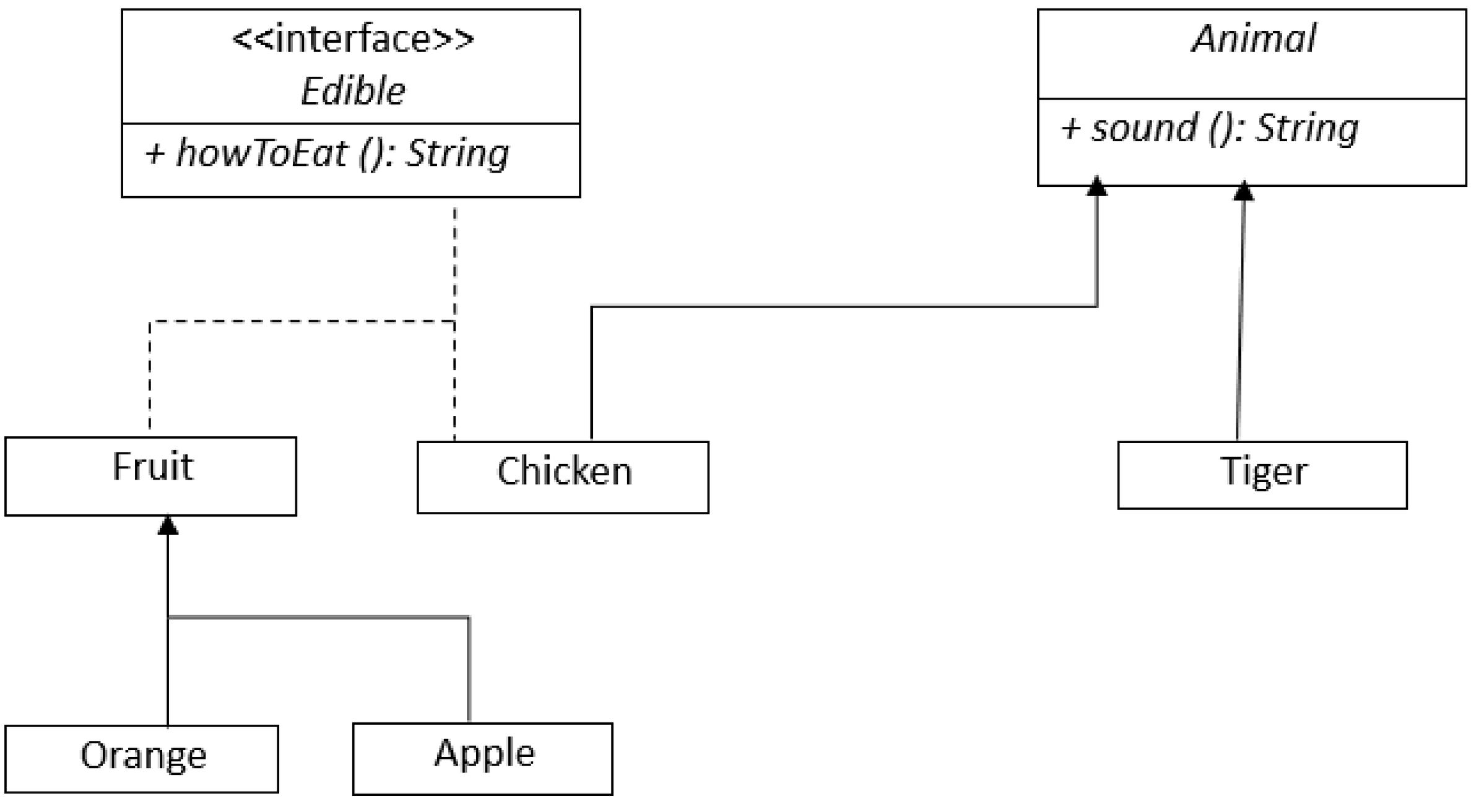
	Variables	Constructors	Methods
Abstract Class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated.	No restrictions
Interface	All variables must be public final static	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Note:

A strong " is a " relationship should be modeled using class inheritance. While, a weak " is a " relationship indicates that an object possesses a certain property, it can be modeled using interfaces.

Example:

Create the Edible interface and use it to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the **implements** keyword.



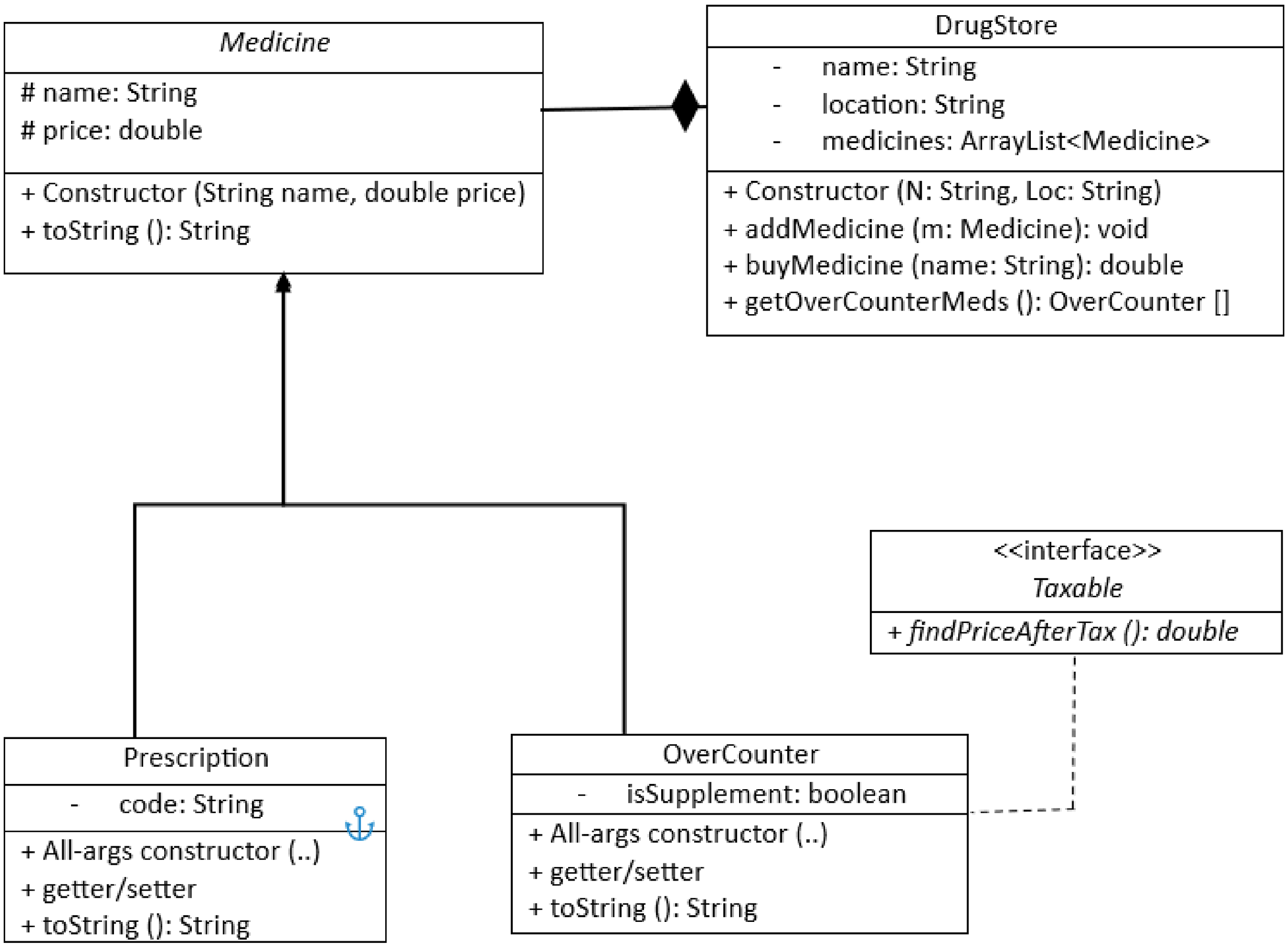
- **C**ode :

```
public interface Edible {
public abstract String howToEat();  }

abstract class Animal {
public abstract String sound();
/* should be overridden in Chicken and Tiger classes, but will
ignore it*/  }

class Chicken extends Animal implements Edible {
public String howToEat() {
return "Chicken: Fry it"; }  }

class Tiger extends Animal {
public String sound() {
return "Roar"; } }

abstract class Fruit implements Edible {}
/* Fruit is declared as an abstract class because it doesn't provide
an implementation for the howToEat() method*/

class Apple extends Fruit {
public String howToEat() {
return "Apple: Make apple cider"; }  }

class Orange extends Fruit {
public String howToEat() {
return "Orange: Make orange juice"; }  }

public class Test {
public static void main (String [] args) {
Edible [] edibles = {new Orange(), new Chicken(), new Apple()};
for(int i=0; i<edibles.length; i++)
 if(edibles[i] instanceof Edible)
   System.out.println(edibles[i].howToEat()); }  }
```

~~Chapter 05 Abstract Vs Interface~~

- **Question :**

Implement the below classes and interface as presented in the following UML diagram. Don't implement getter and setter methods, assume they exist.



~~Chapter 05 Abstract Vs Interface~~

Question :

- Create the abstract class Medicine:
  - Write the AllArgs-constructor that takes as parameters all data fields.
  - The toString() method displays the name and price.
- Create the interface Taxable.
- Don't implement the class Prescription. Assume it exists.
- Implement the class OverCounter:
  - Write the AllArgs-constructor that takes as parameters all data fields.
  - Write the AllArgs-constructor that takes as parameters all data fields.
  - Only over the counter medicines are Taxable with a 10% addition to the price. Implement the findPriceAfterTax method accordingly.
  - The toString() method displays the attributes.
- Implement the class DrugStore, note that :
  - The buyMedicine method searches for a medicine with the given name in the arraylist and returns its price. If it turned out to be an over the counter medicine, then the price after tax is returned.
  - The method getOverCounterMeds returns an array of type OverCounter that contains all of the Over the Counter medicines found in the arraylist.
- Implement the Driver class as follows :
  - Create a DrugStore Object
  - Add three medicines to it of your choice (Not from the user).
  - Read the name of any medicine from the user, buy it and display its price.
  - Display the number of Over the Counter medicines in the drug store.

- **Solution :**

  - Medicine class :

```
public abstract class Medicine {
protected String name;
protected double price;
public Medicine (String name, double price) {
this.name = name;
this.price = price;}
public String getName() {
 return name; }
 public double getPrice() {
 return price; }
public String toString () {
return "Medicine information: \n name: " + name + "\n
price: " + price;}
}
```

  - Taxable Interface :

```
public interface Taxable {
public abstract double findPriceAfterTax ();
}
```

- OverCounter class :

```
public class OverCounter extends Medicine implements
Taxable {
private boolean isSupplement;
public OverCounter (String name, double price,
boolean isSupplement) {
super(name, price);
this.isSupplement = isSupplement;}
@Override
public double findPriceAfterTax() {
double p = this.price*1.1;
return p;}
@Override
public String toString () {
String s = super.toString();
if(isSupplement)
 s+= "\n It is supplement";
else
 s+= "\n It is not supplement";
return s;}
}
```

- DrugStore class :

```
import java.util.ArrayList;
public class DrugStore {
private String name;
private String location;
private ArrayList<Medicine> medicines;
public DrugStore (String N, String loc) {
name = N;
location = loc;
medicines = new ArrayList<Medicine>();}
public void addMedicine (Medicine m) {
medicines.add(m);}
public double buyMedicine (String name) {
for(int i=0; i<medicines.size(); i++) {
if(medicines.get(i).getName().equals(name)) {
if(medicines.get(i) instanceof OverCounter)
return ((OverCounter)medicines.get(i)).findPriceAfterTax();
else
 return ((Prescription)medicines.get(i)).getPrice(); }}
return 0;}
public OverCounter [] getOverCounterMeds () {
int num=0;
for(int i=0; i<medicines.size(); i++)
 if(medicines.get(i) instanceof OverCounter)
num++;OverCounter [] oc = new OverCounter[num];
for(int i=0, j=0; i<medicines.size(); i++)
 if(medicines.get(i) instanceof OverCounter){
  oc[j] = (OverCounter)medicines.get(i);
  j++; }
return oc;}
}
```

- Driver class :

```
import java.util.Scanner;
public class Driver {
public static void main (String [] args) {
Scanner input = new Scanner(System.in);
DrugStore ds = new DrugStore("Hayat", "Zahle");
ds.addMedicine(new OverCounter("Panadol", 150, true));
ds.addMedicine(new OverCounter("Paracetamol", 120,
false));
ds.addMedicine(new Prescription("Flagel", 350));
System.out.println("Enter a medicine's name:");
String n = input.nextLine();
double s = ds.buyMedicine(n);
System.out.println("The price of the medicine bought is: "
+ s);
System.out.println("The number of Over the Counter
medicines is: " + ds.getOverCounterMeds().length);}
}
```

CHAPTER 06

EXCEPTION HANDLING

~~Chapter 06 Exception Handling~~

EXCEPTION HANDLING :

- Exception handling is a mechanism to handle runtime errors, ensuring the normal flow of the application .
- Java uses :
  - Write the AllArgs-constructor that takes as parameters all data fields.
  - Write the AllArgs-constructor that takes as parameters all data fields.
  - try : Code block to monitor for exceptions.
  - catch : Handles the exception.
  - finally : Executes regardless of exception (executed whether an exception occurs or not).
  - throw : Used to explicitly throw an exception.
  - throws : Declares exceptions that a method might throw.

Ex : ArithmeticException : thrown whenever a wrong arithmetic operation appears in the code during run time. It appears whenever the denominator of a fraction is 0.

```
import java.util.Scanner;
public class Application {
public static int quotient(int n1, int n2) {
if (n2 == 0)
 throw new ArithmeticException("Divisor cannot be zero.");
return n1 / n2; }
public static void main(String[] args) {
Scanner input = new Scanner(System.in);
int number1, number2, q;
System.out.print("Enter two integers: ");
number1 = input.nextInt();
number2 = input.nextInt();
try { q = quotient(number1,number2);
      System.out.println("The quotient is " + q); }
catch (ArithmeticException e) {
  System.out.println("Cannot divide by zero."); } } }
```

DECLARING EXCEPTIONS :

- An interface is a class that contains only constants and abstract methods.
- Every method must state the types of checked exceptions it might throw. This is known as declaring exceptions.
- Ex :

```
public void myMethod()
throws IOException
```

THROWING AND CATCHING AN EXCEPTION :

- An exception is an error represented as an object in Java :

```
new ArithmeticException("Divisor cannot be zero")
```

- This object is created from a specific exception class, such as java.lang.ArithmeticException
- When an exception occurs, a throw statement is used to create and send this exception object :

```
throw new ArithmeticException("Divisor cannot be zero")
```

- The message passed to the exception’s constructor describes the cause of the error, helping with debugging.
- Throwing an exception means interrupting the normal flow of the program and transferring control to an error handler.

THROWING AND CATCHING AN EXCEPTION :

- The method that might throw an exception is wrapped in a try block.
  - The try block contains the code that runs under normal conditions but may potentially cause an exception.
- If an exception is thrown inside the try block, Java looks for a matching catch block to handle it.
  - The catch block receives the exception object as a parameter :

    catch (ArithmeticException e)
- The catch block contains code that handles the exception, allowing the program to respond appropriately.
- After the catch block finishes execution, the program continues with the next statement after the entire try-catch structure.
- You can think of the throw statement as similar to a method call, but instead of calling a method, it transfers control to the catch block.
- Unlike a method call, the control does not return to the throw point after the exception is handled, it moves forward from the catch block.

NOTES :

- Java requires you to handle checked exceptions either by catching them or by declaring them to be thrown, so the program doesn’t crash unexpectedly.
- If a method can handle the exception where it occurs, there's no need to throw it. But if the caller should handle it, you must create and throw an exception object.

TYPES OF EXCEPTIONS :

- Throwable : The top-level superclass for all errors and exceptions in Java. It is part of ' java.lang '.
- Throwable has two main subclasses:
  1 . Exception
  2 . Error

1. **EXCEPTION :**

- Represents events that a program can detect and handle (file not found, invalid input).
- The Exception class itself has many subclasses designed to handle specific types of exceptions.
- Common subclasses include:
  - IOException : issues during file or data input/output.
  - SQLException : problems with database access.
  - RuntimeException : base class for unchecked exceptions.

CONSTRUCTORS OF EXCEPTION CLASS:

- Exception() : creates an exception object with no detail message.
- Exception(String message) : creates an exception with a custom error message.

GENERIC HANDLING USING EXCEPTION CLASS:

- You can catch all types of exceptions (both checked and unchecked) using a single catch block:

```
try { // }
catch (Exception e) { // handle any exception}
```

CHECKED EXCEPTIONS

- Checked at compile time. The compiler ensures that these exceptions are either handled using try-catch or declared using throws.
- Examples:
  - IOException
  - SQLException
  - FileNotFoundException

UNCHECKED EXCEPTIONS

- Include RuntimeException, Error, and all their subclasses.
- Not checked at compile time. Java does not require explicit handling of these exceptions.
- Typically result from programming logic errors (invalid operations or improper input).
- These exceptions should be prevented by writing correct code, not by trying to catch them.
- Examples:
  - ArithmeticException : division by zero.
  - NullPointerException : using an object that hasn't been initialized.
  - IndexOutOfBoundsException : accessing array or list elements out of range.
  - InputMismatchException : reading input in an unexpected format.
  - IllegalArgumentException : passing an illegal argument to a method.

Note :

Unchecked exceptions can occur anywhere in a program. To reduce overuse of try-catch blocks, Java does not force the programmer to catch them.

2. **ERROR :**

- Represents serious problems that are not meant to be handled by normal applications.
- These are thrown by the Java Virtual Machine (JVM) and indicate system-level failures.
- Examples:
  - OutOfMemoryError
  - StackOverflowError
  - VirtualMachineError

GETTING INFORMATION FROM EXCEPTIONS :

An exception object holds useful details about the error, and you can use methods from the Throwable class to access this information.

java.util.Throwable
+ getMessage (): String **Returns the message that describes this exception object.** + toString (): String **Returns the concatenation of three Strings: the full name of the exception class, (a colon and a space), the getMessage method** + printStackTrace (): void **Prints the Throwable object and its call stack trace information on the console**

- Ex 1 : NullPointerException & IndexOutOfBoundsException

```
public class Student{
private int id;
private String name;
public Student(int i, String n) {
id = i;
name = n; }
public void print() {
System.out.println("The ID is" + id + " The name is " + name);}}

import java.util.Scanner;
public class Application {
public static void main(String[] args) {
Scanner scan = new Scanner(System.in);
Student s[]=new Student[10];
s[0]=new Student(1,"A");
s[1]=new Student(2,"B");
int index;
System.out.println("Enter an index:");
index = scan.nextInt();
try {
s[index].print(); }
catch(IndexOutOfBoundsException e1) {
System.out.println("Index Out of bounds"); }
catch (NullPointerException e) {
System.out.println("Attempting to access a null reference"); }
System.out.println("The end");
scan.close(); }}
```

Output :

- If the user enters an index less than 0 or greater than 9 : "Index Out of bounds."
- If the user enters an index from 2 to 9 (cells with null values) : "Attempting to access a null reference."
- If the user enters index 0 or 1 (valid student objects) : Student ID and Name

- Ex 2 : InputMismatchException

```
import java.util.*;
public class Main {
public static void main(String[] args) {
 int n, m, sum;
Scanner scan = new Scanner(System.in);
boolean repeat = true;
System.out.print("Enter two integers: ");
do {
try {
n = scan.nextInt();
m = scan.nextInt();
sum = n + m;
System.out.println("sum is: " + sum);
repeat = false; }
catch (InputMismatchException ime) {
scan.nextLine(); // To skip the invalid input
System.out.print("Enter two integers: "); }
} while (repeat);
System.out.println("The end."); }
}
```

Output :

- If the user enters two valid integers : Sum of the two values, then "The end."
- If one or both values are not integers : The program keeps asking for two inputs until both are integers,
- then it shows the sum and "The end."

- Ex 3 : IllegalArgumentException

```
public class Student{
private int id;
private String name;
public Student(int i, String n) throws IllegalArgumentException {
if (i < 0)
  throw new IllegalArgumentException("ID cannot be negative.");
if (n.length() < 2)
  throw new IllegalArgumentException("Name should have at least
2 letters.");
id = i;
name = n; }
public void print() {
System.out.println("ID: " + id + " Name: " + name); } }

import java.util.Scanner;
public class Main {
public static void main(String[] args) {
Scanner scan = new Scanner(System.in);
System.out.println("Enter the name and the id: ");
String name = scan.next();
int id = scan.nextInt();
try {
Student s = new Student(id, name);
s.print(); }
catch(IllegalArgumentException iae) {
System.out.println(iae.getMessage()); } }}
```

Output :

- If the ID is negative : "ID cannot be negative."
- If the name has less than 2 letters : "Name should have at least 2 letters."
- If both ID and name are valid : Student ID and Name

CREATING CUSTOM EXCEPTION CLASS :

- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending Exception or a subclass of Exception.

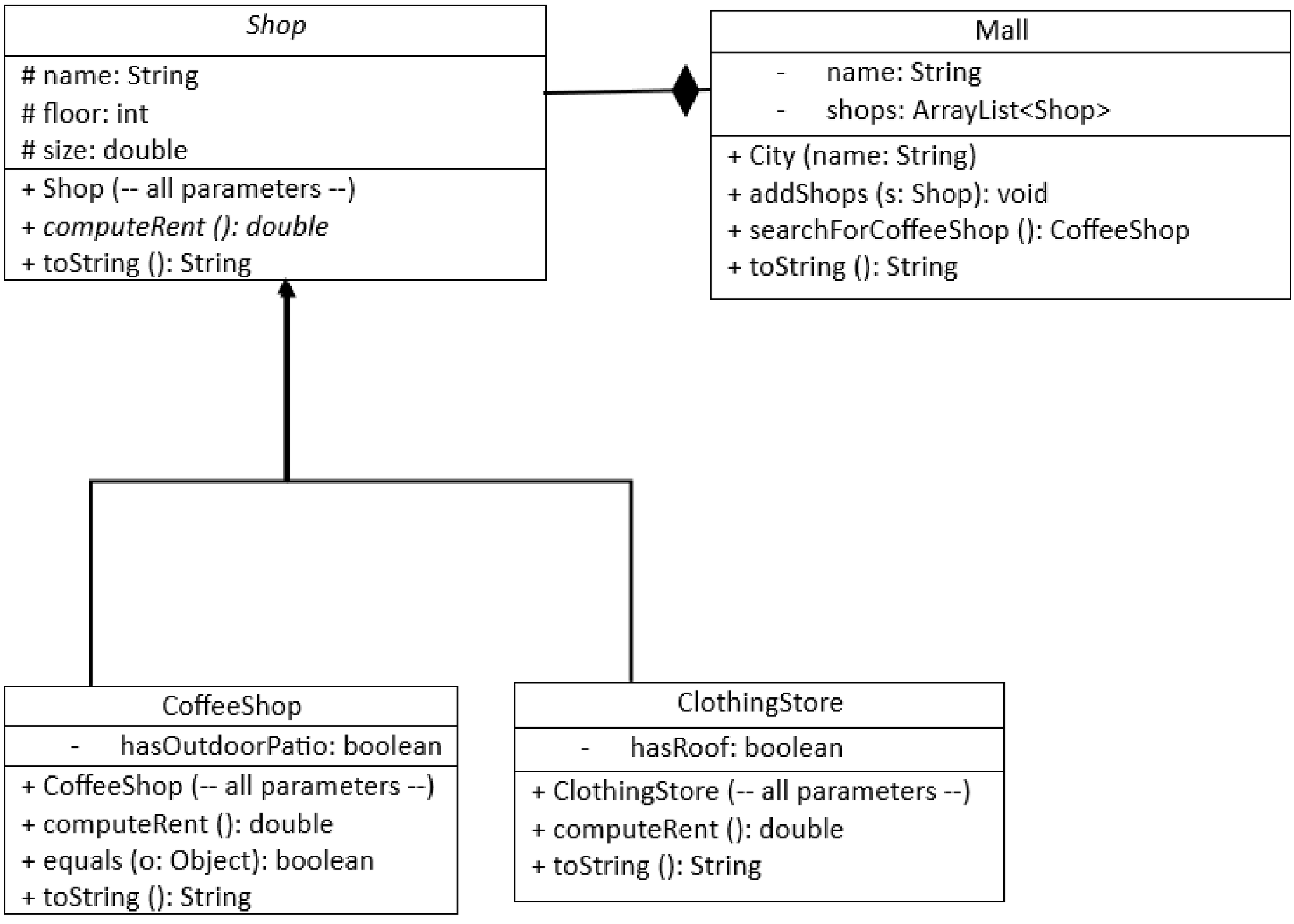
EX :

```
public class InvalidRadiusException extends Exception {
public InvalidRadiusException(String message) {
super(message); // calling the constructor of Exception } }

public class Circle {
private double radius;
public Circle(double r) throws InvalidRadiusException {
if (r < 0)
  throw new InvalidRadiusException("The radius cannot be
negative");
radius = r; }
public void print() {
System.out.println("The radius is " + radius); } }

import java.util.Scanner;
public class Test {
public static void main(String[] args) {
Scanner scan = new Scanner (System.in);
double r;
System.out.println("Enter a radius: ");
r = scan.nextDouble();
try {
Circle c1 = new Circle(r);
c1.print(); }
catch (InvalidRadiusException ex) {
System.out.println(ex); } }}
```

Question :

The following is the UML diagram with four classes. You are asked to implement the classes Shop, CofeeShop and Mail.

- Don't implement the class ClothingStore but assume it exists.
- Don't implement any getter or setter of any class but assume they exist.



- Implement the abstract class Shop as follows:
  - Define its attributes.
  - The constructor creates a Shop object with the given values of all attributes. It throws an IllegalArgumentException if the given value of the floor attribute is less than 0 or greater than 3.
  - Define the abstract method computerRent.
  - toString : It returns all attributes as a String separated by spaces.

~~Chapter 06 Exception Handling~~

Question :

- Implement the class CoffeeShop which is a subclass of Shop:
  - Override the method computeRent to return the rent of the CoffeeShop object which is equal to its size multiplied by 20. If it has an outdoor patio, then 100 is added to the rent.
  - Override the equals method: Two CoffeeShop objects are equal if they have the same rent.
  - toString : It returns all attributes of the CoffeeShop class as well as all the attributes of its superclass.
- Remember that a Shop object could be either a CoffeeShop or a ClothingStore object. Implement the class Mall which has a composition relationship with the Shop class:
  - The constructor creates a Mall object with the given name. It also creates the ArrayList<Shop> shops which stores all shops of the mall.
  - The method addShop adds the Shop object s to the shops ArrayList.
  - The method searchForCoffeeShop() searches the shops ArrayList for a CoffeeShop object with an outdoor patio. The first CoffeeShop object that meets this criterion is returned. If none is found, null is returned.
  - toString: returns the description of the mall as a String.
- Implement the Test class as follows:
  - This class handles the IllegalArgumentException thrown by the constructor of the Shop's class.
  - Create a Mall object with any data of your choice.
  - Create one CoffeeShop object of any data of your choice. Add it to the mall created.
  - Create one ClothingStore object of any data of your choice. Add it to the mall created.
  - Search the mall for a CoffeeShop that has an outdoor patio and display its name if found. If none is found, display the message " None is found ".
  - Display the description of the mall using the toString method.

- **Solution :**
  - Shop class :

```
public abstract class Shop {
protected String name;
protected int floor;
protected double size;
public Shop(String name, int floor, double size) {
if (floor < 0 || floor > 3) {
  throw new IllegalArgumentException("Floor must be
between 0 and 3."); }
this.name = name;
this.floor = floor;
this.size = size; }
public String getName() {
return name; }
public abstract double computeRent();
public String toString() {
return "Name: "+name+", Floor: "+floor+", Size: "+size;
}
```

- CoffeeShop class :

```
public class CoffeeShop extends Shop {
private boolean hasOutdoorPatio;
public CoffeeShop(String name, int floor, double size,
boolean hasOutdoorPatio) {
super(name, floor, size);
this.hasOutdoorPatio = hasOutdoorPatio; }
public boolean hasOutdoorPatio() {
return hasOutdoorPatio; }
public double computeRent() {
double rent = size * 20;
if (hasOutdoorPatio) {
  rent += 100; }
return rent; }
public boolean equals(Object o) {
if (o instanceof CoffeeShop) {
  CoffeeShop other = (CoffeeShop) o;
  return this.computeRent() == other.computeRent(); }
return false; }
public String toString() {
return hasOutdoorPatio + " " + super.toString(); }
}
```

- Mall class :

```
import java.util.ArrayList;
public class Mall {
private String name;
private ArrayList<Shop> shops;
public Mall(String name) {
this.name = name;
this.shops = new ArrayList<Shop>(); }
// Getters, Setters
public void addShop(Shop s) {
shops.add(s); }
public CoffeeShop searchForCoffeeShop() {
for (int i=0; i<shops.size(); i++) {
 Shop s = shops.get(i);
 if (s instanceof CoffeeShop) {
  CoffeeShop cs = (CoffeeShop) s;
    if (cs.hasOutdoorPatio()) { return cs; } } }
return null; }
public String toString() {
String s = "Mall:" + name;
for(int i=0; i<shops.size(); i++)
 s+= "\n" + shops.get(i).toString();
return s; }
}
```

- Test class :

```
public class Test {
public static void main(String[] args) {
try {
Mall mall = new Mall("Lebanon Mall");
CoffeeShop coffeeShop = new CoffeeShop("StarCafe",
1, 30.5, true);
mall.addShop(coffeeShop);
ClothingStore clothingStore = new
ClothingStore("StyleZone", 2, 45.0, true);
mall.addShop(clothingStore);
CoffeeShop found = mall.searchForCoffeeShop();
if (found != null) {
  System.out.println("Found CoffeeShop: " +
found.getName()); }
else
  System.out.println("None is found");
System.out.println("\n Mall Description:");
System.out.println(mall.toString()); }
catch (IllegalArgumentException e) {
System.out.println("Error creating shop: " +
e.getMessage()); } }
}
```