OOP

Interfaces Polymorphism

Part 2

softserve

AGENDA

- Java OOPs Concepts
- Interface
- Polymorphism
- Sorting
- *Class Diagram

Java OOPs Concepts

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Java OOPs Concepts

Inheritance

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: cat speaks meow, dog barks woof etc.

Abstraction

Hiding internal details and showing functionality is known as an abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.

Interfaces

- An **interface** is a reference type in Java, it is similar to class, it is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
- Along with *abstract methods* an interface may also contain *constants*, *default methods*, *static methods*, and *nested types*. Method bodies exist only for default methods and static methods.
- An interface is essentially a **type** that can be satisfied by any class that implements the interface.
- Any class that implements an interface must satisfy 2 conditions
  - It must have the phrase "implements *Interface_Name*" at the beginning of the class definiton;
  - It must implement all of the method headings listed in the interface definition.

Interfaces

```
public interface Worker {
    int getSalary(); // public abstract
}

public class Director implements Worker {
   public int getSalary() {
       // method definition here
   }
}
```

Interfaces

- Example

```
interface Volumetric {
    double PI = 3.14;

    double getVolume();

    static double getPI() {
        return Volumetric.PI;
    }
    default String info() {
        return definition() +
            "1 litre = (10 cm)^3 = 1000 cubic centimetres = 0.001 cubic metres";
    }
    private String definition() {
        return "Volume is the quantity of three-dimensional space" +
            "enclosed by a closed surface.\n";
    }
}
```

Interfaces

- To access the interface methods, the interface must be "implemented by another class with the **implements** keyword.
- Example:

```
public class Ball extends Shape implements Volumetric {
    private double radius;
    public Ball(double radius, String name) {
        super(name); this.radius = radius;
    }
    @Override
    public double getArea() {
        return 4 * Math.PI * radius * radius;
    }
    @Override
    public double getVolume() {
        return 4.0 / 3 * Volumetric.PI * Math.pow(radius, 3);
    }
}
```

Multiple interfaces implementation

- To **implement *multiple interfaces***, separate them with a comma:

```
class Cube extends Shape implements Vertexable, Volumetric {
    private double side;
    public Cube(double side, String name) {
        super(name);
        this.side = side;
    }
    @Override
    public double getArea() {
        return 12 * side;
    }
    @Override
    public double getVolume() {
        return Math.pow(side, 3);
    }
    @Override
    public int getNumberOfVertex() {
        return 8;
    }
}
```

```
interface Vertexable {
    int getNumberOfVertex();
}

interface Volumetric {
    double getVolume();
}
```

Interfaces with same method

- If a type implements ***two interfaces***, and each interface define a method that has ***identical signature***, then in effect there is only one method, and they are ***not distinguishable***.

```
public interface Vertexable {

    ...

    void info();
}
```

```
public interface Volumetric {

    ...

    void info();
}
```

```
public class Cube extends Shape implements Vertexable, Volumetric {

    ...

    @Override
    public void info() { ... }
}
```

Note that there is ***only one*** **`@Override`** ***necessary***. This is because **`Vertexable.info`** and **`Volumetric.info`** are "Override-equivalent"

softserve

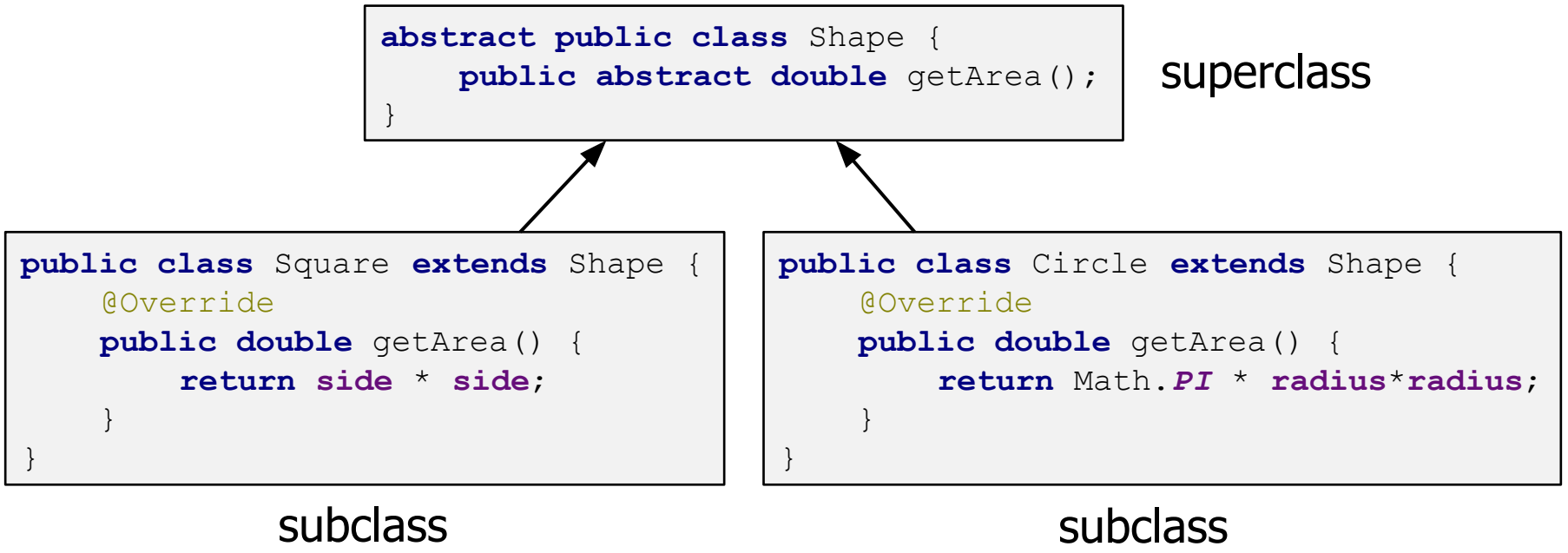
Extending Interfaces

- An interface can **extend *another interface*** in the same way that a class can extend another class.
- The **`extends`** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

```
interface A { }
interface B { }

interface Volumetric extends A, B {
    double getVolume();
}
```

Polymorphism

- Polymorphism is the concept, according to which a ***common interface*** used for data processing ***various specialized types***.



- Java code uses ***late binding*** technique to support polymorphism and the method to be invoked is decided at runtime

The `instanceof` Operator

- The **`instanceof`** operator allows you determine the type of an object.

```
Shape shapes[] = {
        new Square(7.2, "MySquare"),
        new Circle(5.8, "MyCircle"),
        new Cube(6.7, "MyCube"),
        new Ball(6.3, "MyBall")
};

for (Shape shape : shapes) {
    double area = shape.getArea();
    System.out.println(area);

    if (shape instanceof Volumetric) {
        double volume = ((Volumetric) shape).getVolume();
        System.out.println(volume);
    }
}
```

Polymorphism

```
public abstract class ACar {
    private double maxSpeed;

    public double getMaxSpeed( ) { return maxSpeed; }

    public void setMaxSpeed(double maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    abstract void carRides( );
}
```

Polymorphism

```
public class BmwX6 extends ACar {
    public BmwX6( ) { }

    @Override
    public void carRides( ) {
        setMaxSpeed(200);
        System.out.println("Car Rides");
        workedEngine( );
        workedGearBox( );
    }
```

Polymorphism

Are private fields and methods inherited?

```
 public void workedEngine( ) {
        System.out.println("BmwX6: Engine Running on Petrol.");
        System.out.println("BmwX6: Max Speed: " + getMaxSpeed( ));
    }

 private void workedGearBox( ) {
         System.out.println("BmwX6: Worked GearBox.");
 }

 public void lightsShine( ) {
     System.out.println("BmwX6: Halogen Headlights.");
 }
}
```

Polymorphism

```
public class BmwX6mod extends BmwX6 {
    public BmwX6mod( ) { super( ); }

    @Override
    public void workedEngine( ) {
        System.out.println("BmwX6mod: Engine Running on Diesel.");
        System.out.println("BmwX6mod: Max Speed: " + getMaxSpeed( ));
    }

    @Override
    public void lightsShine( ) {
        System.out.println("BmwX6mod: Xenon Headlights.");
        super.lightsShine();
    }
}
```

Polymorphism

```
public class Appl {
    public static void main(String[ ] args) {
        ACar carX6 = new BmwX6( );
     carX6.carRides( );
        ((BmwX6)carX6).lightsShine( );

     ACar carX6mod = new BmwX6mod( );
        carX6mod.carRides( );
        ((BmwX6)carX6mod).lightsShine( );

     BmwX6 carX6mod2 = new BmwX6mod( );
        carX6mod2.carRides( );
        carX6mod2.lightsShine( );
    }
}
```

Sorting

```
public static void main(String[] args) {
      int[] x = new int[10];
      Random rand = new Random();
      for (int i = 0; i < x.length; i++) {
         x[i] = rand.nextInt(10);
      }
      Arrays.sort(x);
      for (int i = 0; i < x.length; i++) {
         System.out.println(x[i]);
      }
   }
```

What is wrong in the code

- Write a new code for type double, etc.
- Do you need to constantly create "bicycle" ?
- You may use an existing solution

Class Arrays. Sorting

```
public static void main(String[ ] args) {
        Student[ ] students = new Student[3];
        students[0] = new Student(52645, "Oksana");
        students[1] = new Student(98765, "Bogdan");
        students[2] = new Student(1354, "Orest");
        Arrays.sort(students);
        for (int i = 0; i < students.length; i++) {
            System.out.println(students);
    }
}
```

What will happen?

Compare elements

To specify the order of the following interfaces: **Comparable** and **Comparator**

```
public class MyType implements Comparable {
    String name;
    public int compareTo(Object obj) {
        return name.compareTo(((MyType)obj).name);
    }
}
```

Comparable to specify only one order.
Method **compareTo** can return
0, if objects are equal
<0 (-1), if first object is less than second object
>0 (1), if first object is great than second object

Interface **`Comparable`**

- Interface **`Comparable`** allows ***custom sorting*** of objects when implemented.
- When a class implements this interface, we must add the public method **`compareTo(Object o)`**.

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    @Override
    public int compareTo(Person p) {
        if (this.name.compareTo(p.name) != 0 )
            return this.name.compareTo(p.name);
        else
            return Integer.compare(this.age, p.age);
    }
}
```

Interface **Comparable**

- Example:

```
Person people[] = {
        new Person("Bill", 34),
        new Person("Tom", 23),
        new Person("Alice", 21),
        new Person("Bill", 27)
};

for (Person person : people) {
    System.out.println(person);
}

Arrays.sort(people);

for (Person person : people) {
    System.out.println(person);
}
```

```
Name: Bill, age: 34
Name: Tom, age: 23
Name: Alice, age: 21
Name: Bill, age: 27
```

```
Name: Alice, age: 21
Name: Bill, age: 27
Name: Bill, age: 34
Name: Tom, age: 23
```

Interface **`Comparator`**

- Interface Comparator allows ***custom sorting*** of objects when implemented.
- When a class implements this interface, we must add the public method **`compare(Object o1, Object o2)`**.
- Methods compare can throw an exception *ClassCastException*, if the object types are not compatible in the comparison.

Example 1

```
public class Employee {
    int tabNumber;
    String name;
    public Employee(String name, int tabNumber) {
        this.name = name;
        this.tabNumber = tabNumber;
    }
    @Override
    public String toString() {
        return "Employee [tabNumber=" + tabNumber + ", name=" + name + "]";
    }
}
```

Example 1

```
import java.util.Comparator;
public class NameComparator implements Comparator<Employee>{
    @Override
    public int compare(Employee o1, Employee o2) {
        return o1.name.compareTo(o2.name);
    }
}
```

```
import java.util.Comparator;
public class TabComparator implements Comparator<Employee>{
    @Override
    public int compare(Employee o1, Employee o2) {
        return o1.tabNumber - o2.tabNumber;
    }
}
```

Example 1

```
import java.util.ArrayList;
import java.util.List;
public class Main {
    public static void main(String[] args) {
      List<Employee> list = new ArrayList<Employee>();

      list.add(new Employee("Vasya", 15));
      list.add(new Employee("Anna", 2));
      list.add(new Employee("Alina", 40));
```

Example 1

```
        list.sort(new NameComparator());
            for (Employee employee : list) {
                System.out.println(employee);
            }

        list.sort(new TabComparator());
            for (Employee employee : list) {
                System.out.println(employee);
            }
    }
}
```

Example 2

Add get() and set() methods

```
public class Employee {
    int tabNumber;
    String name;
    static NameComparator nameComparator = new NameComparator( );
    static TabComparator tabComparator = new TabComparator();
    public static Comparator getNameComparator( ) {
        return nameComparator;
    }
    public static Comparator getTabComparator( ) {
        return tabComparator;
    }
```

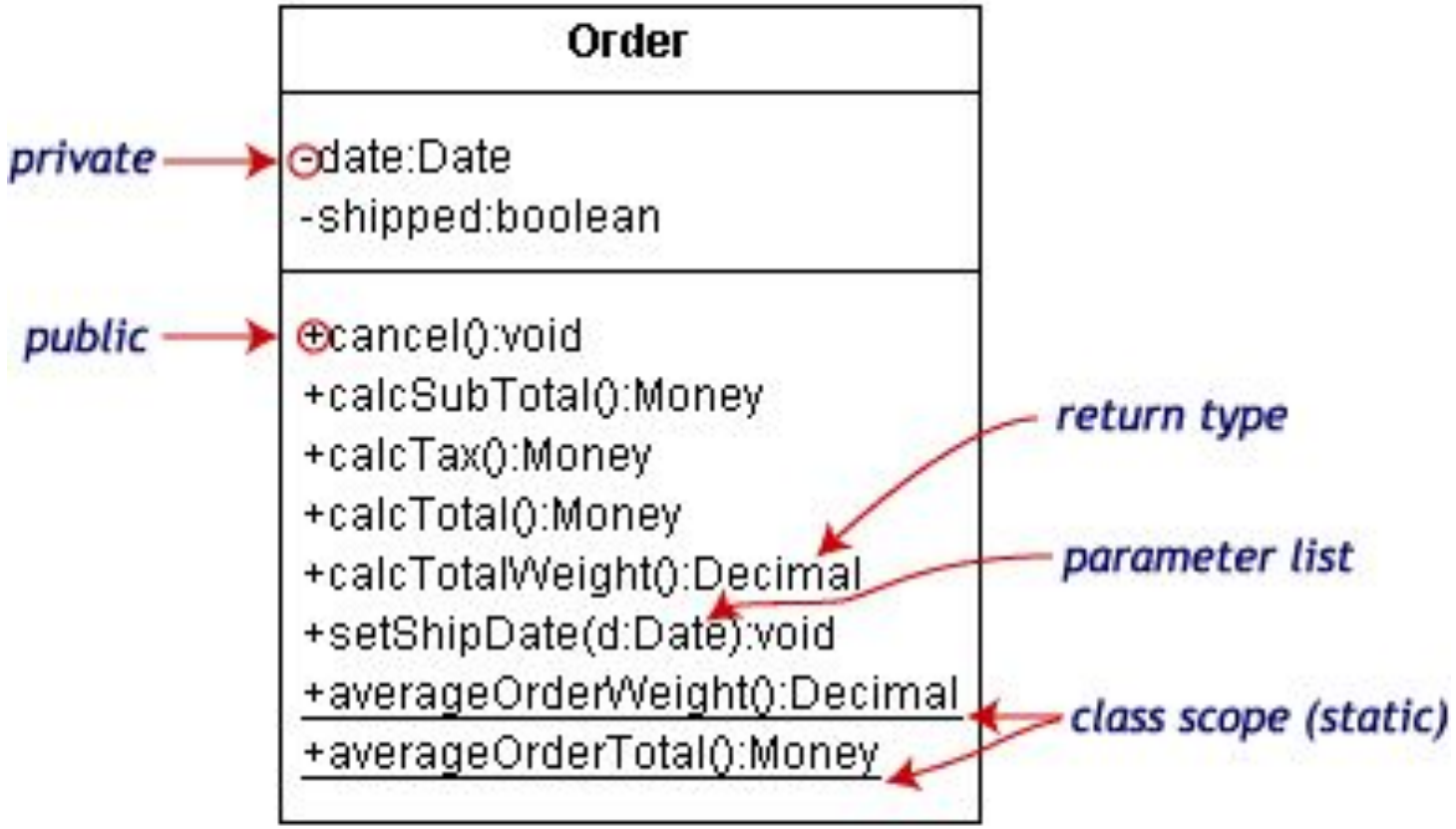
Example 2

```
static class NameComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    return ((Employee)o1).getName().compareTo(((Employee)o2).getName());
  }
}

static class TabComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    return ((Employee)o1).getTabNumber() – ((Employee)o2).getTabNumber();
  }
} . . . }
```

Example 2

```
public static void main(String[] args) {
   Set<Employee> set = new TreeSet(Employee.getNameComparator());
   set.add(new Employee(15, "Vasya"));
   set.add(new Employee(2, "Anna"));
   set.add(new Employee(40, "Alina"));
   System.out.println(set);

   Set<Employee> set1 = new TreeSet(Employee.getTabComparator());
   set1.addAll(set);
   System.out.println(set1);
}
```

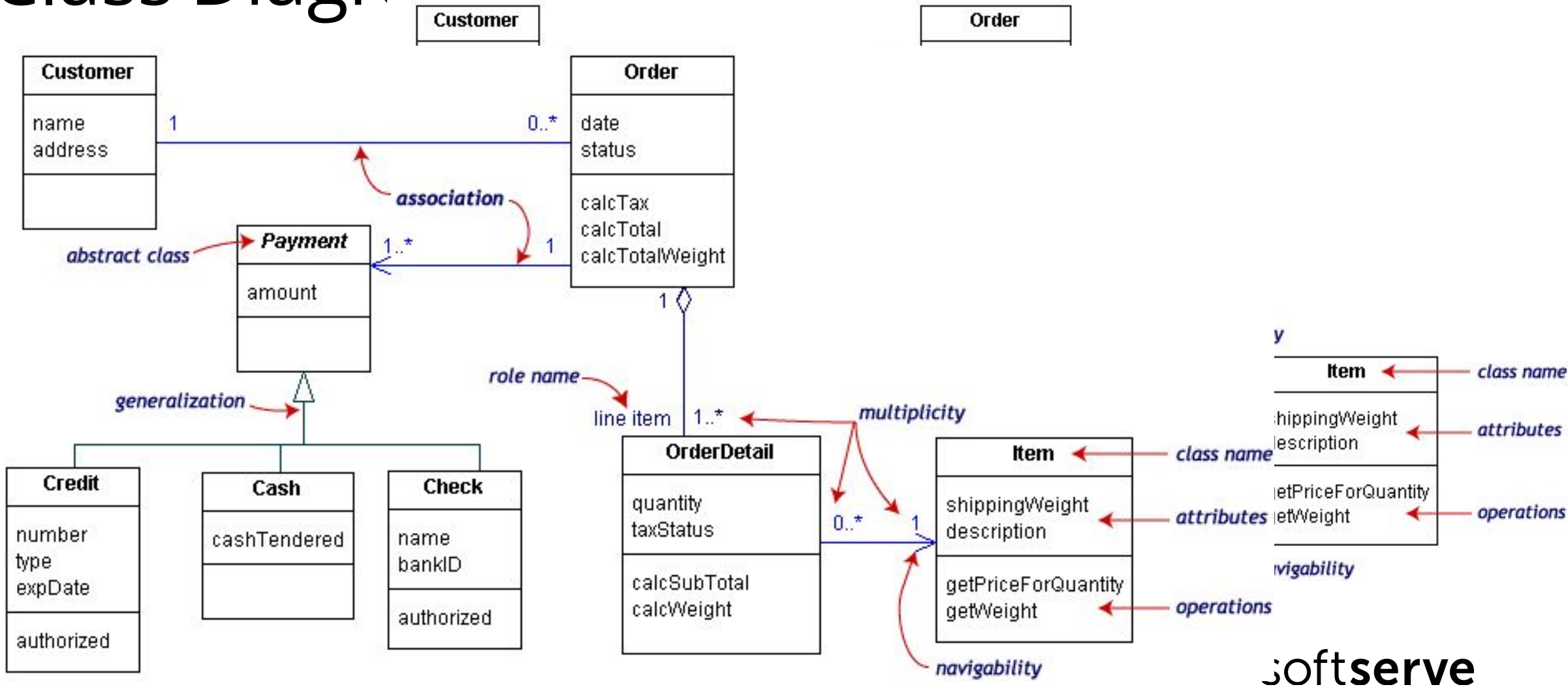
Class Diagram. Visibility and scope



Symbol	Access
+	public
-	private
#	protected

softserve

Class Diagram



softserve

Class Diagram

Our class diagram has three kinds of relationships.

association -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.

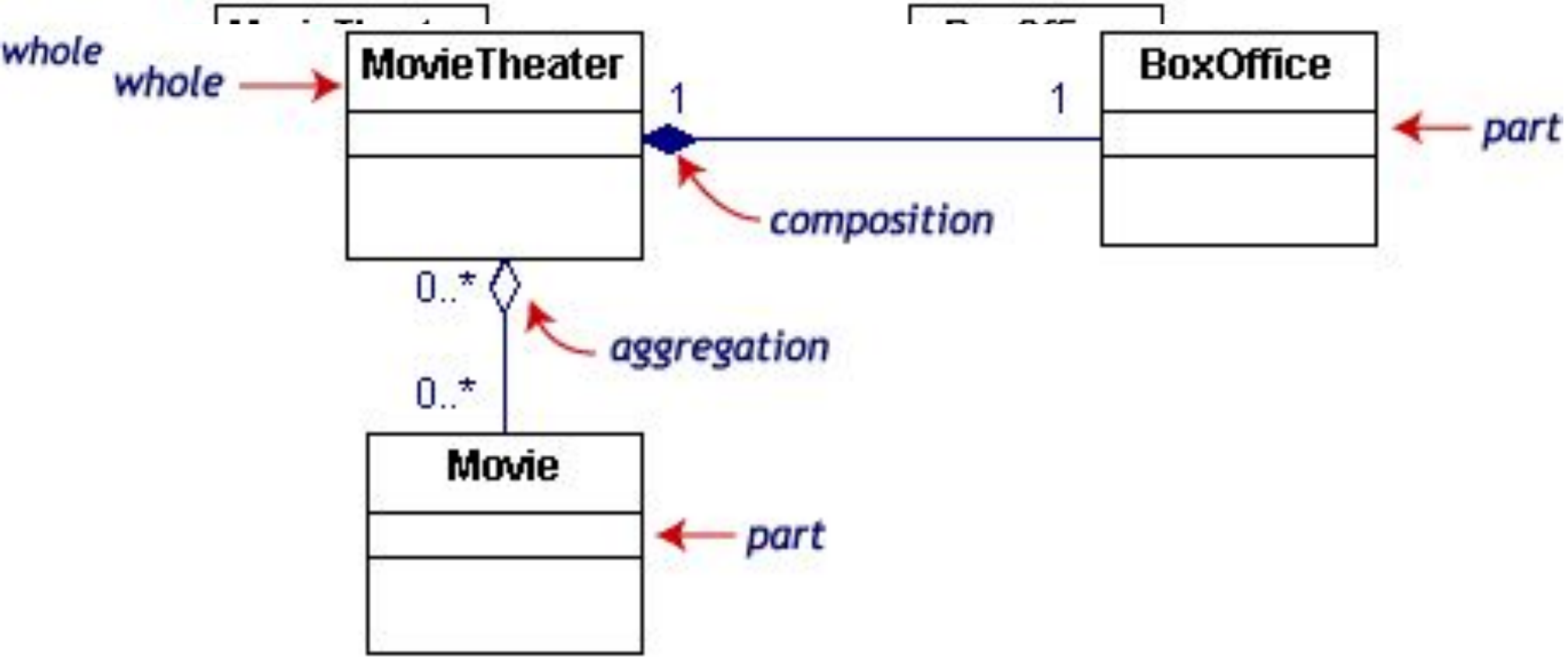
aggregation -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. In our diagram, Order has a collection of OrderDetails.

generalization -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass. Payment is a superclass of Cash, Check, and Credit.

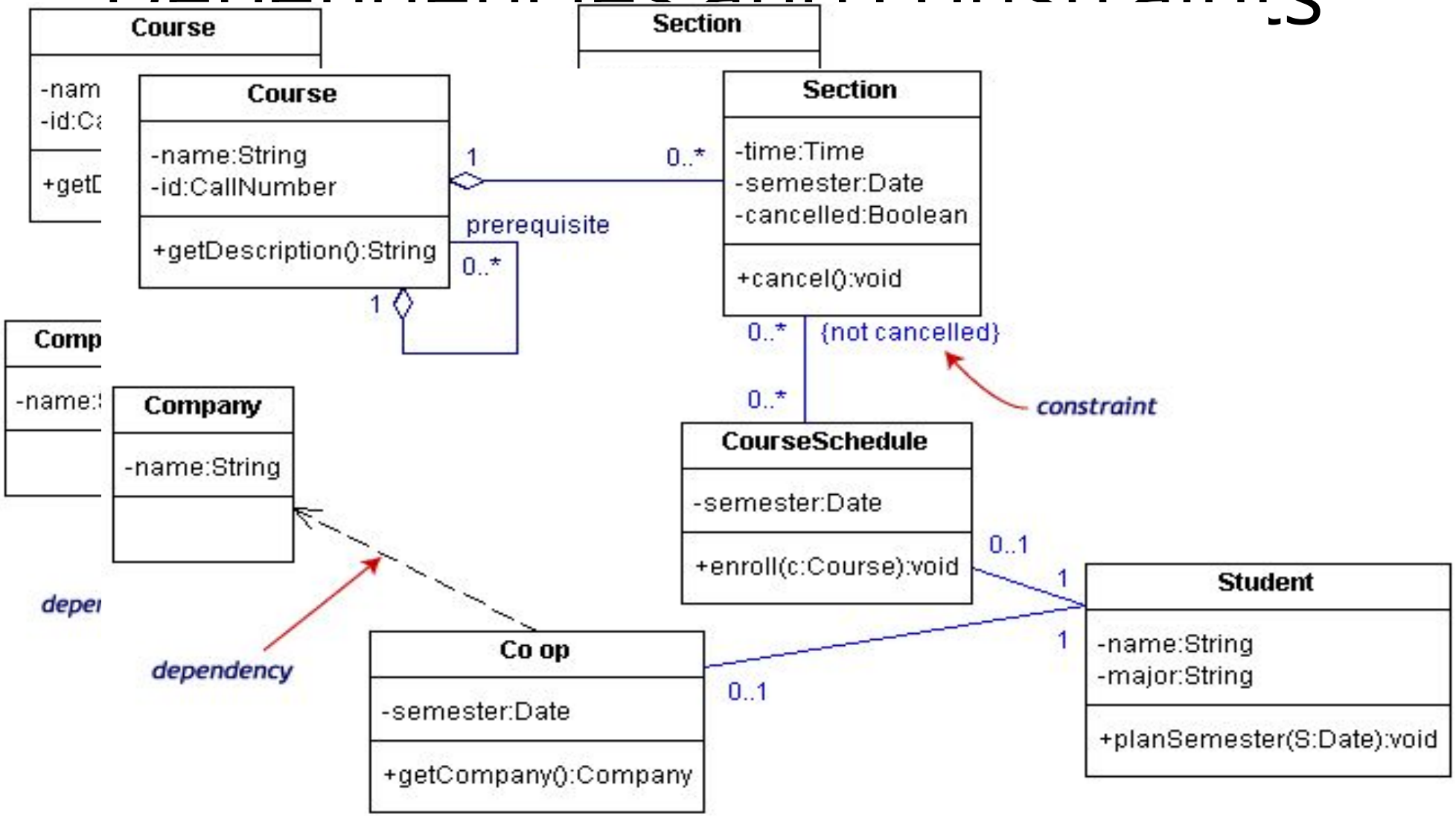
Class Diagram. Multiplicities

Multiplicities	Meaning
0..1	zero or one instance. The notation n . . M indicates n to m instances.
0..* or *	no limit on the number of instances (including none).
1	Exactly one instance
1..*	at least one instance

Composition and aggregation

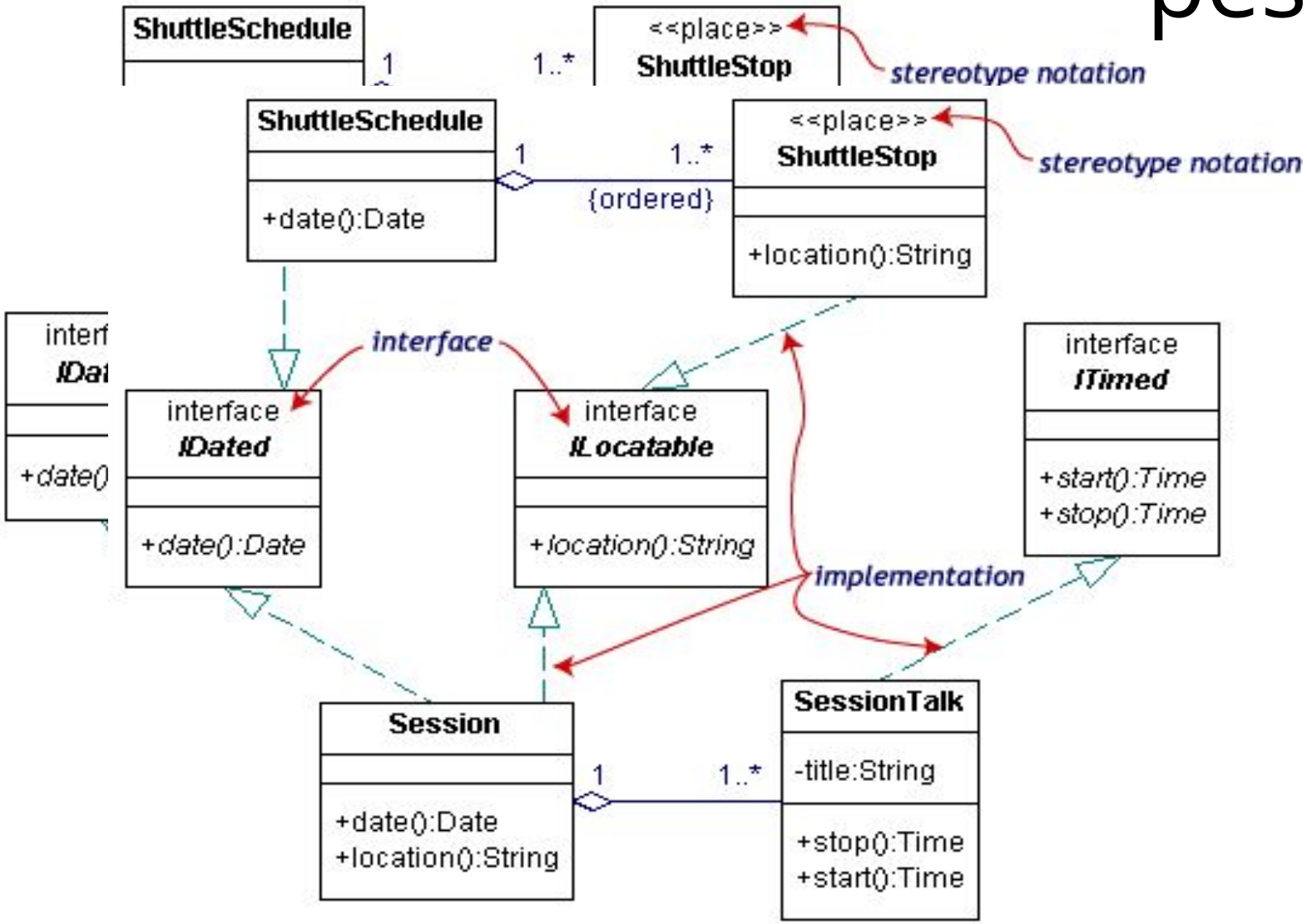


Dependencies and constraints



softserve

Interfaces and stereotypes



softserve

final

A ***final variable*** can only be assigned once and its value cannot be modified once assigned.

Constants are variables defined

```
final double RADIUS = 10;
```

A ***final method*** cannot be overridden by subclasses

```
public final void myFinalMethod() {...}
```

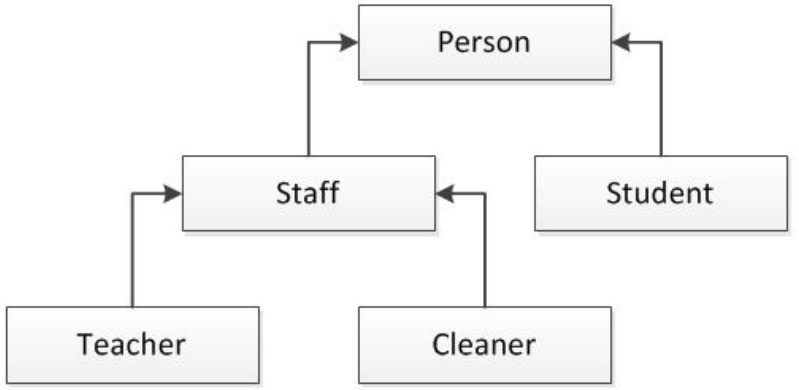
A ***final class*** cannot extend

```
public final class MyFinalClass {...}
```

Practical tasks

1. Create interface ***Animal*** with methods ***voice***() and ***feed***(). Create two classes ***Cat*** and ***Dog***, which implement this interface. Create array of Animal and add some Cats and Dogs to it. Call ***voice***() and ***feed***() method for all of it

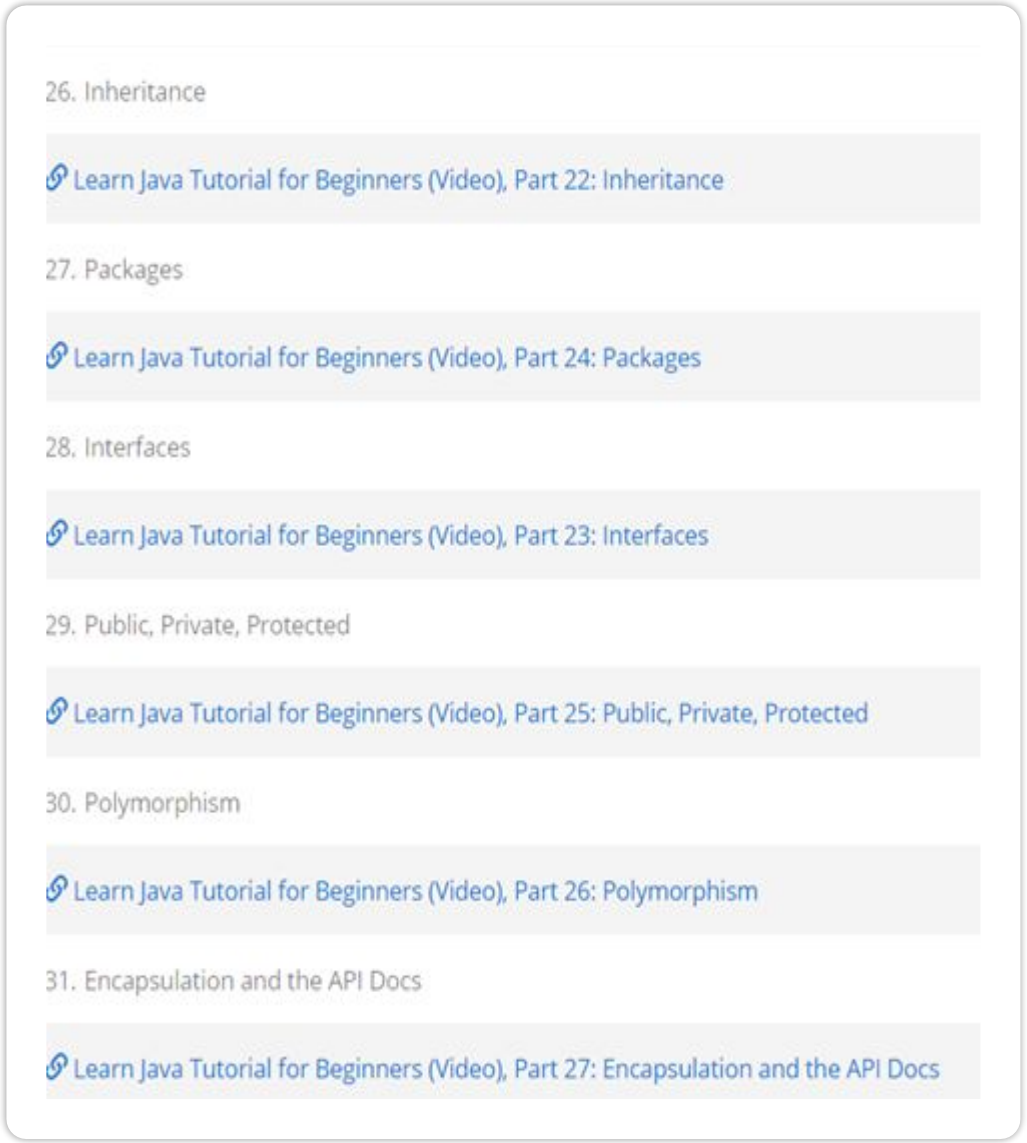
2. Create next structure. In abstract class ***Person*** with property ***name***, declare abstract method ***print***(). In other classes in body of method *print*() output text “I am a …”. In class Staff declare abstract method ***salary***(). In each concrete class create constant *TYPE_PERSON*. Output type of person in each constructors. Create array of *Person* and add some Teachers, Cleaners and Students to it. Call method *print*() for all of it. Call method *salary*() for all Teachers and Cleaner



softserve

HomeWork (online course)

- UDEMY course "Java Tutorial for Complete Beginners": https://www.udemy.com/java-tutorial/
- Complete lessons 26-31:



softserve

Homework

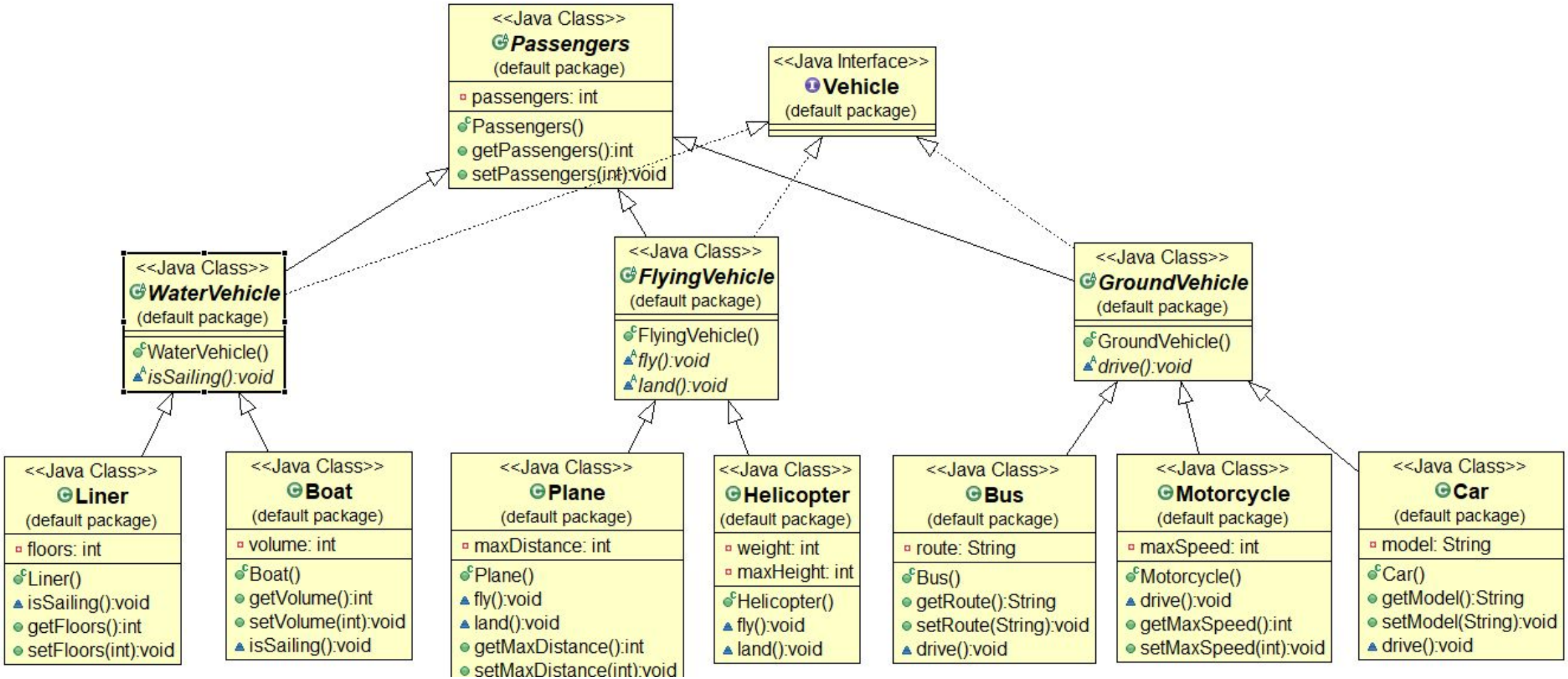
1. Create ***Payment*** interface with the method *calculatePay()*, the base class Employee with a string variable *employeeId.*
   Create two classes ***SalariedEmployee*** and ***ContractEmployee***, which implement interface and are inherited from the base class.
   - Describe hourly paid workers in the relevant classes (one of the children), and fixed paid workers (second child).
   - Describe the string variable *socialSecurityNumber* in the class SalariedEmployee .
   - Include a description of *federalTaxIdmember* in the class of ContractEmployee.

Homework

- The calculation formula for the "time-worker“ is: *the average monthly salary* = *hourly rate* * *number of hours worked*
- For employees with a fixed payment the formula is: the average monthly *salary* = *fixed monthly payment*
- Create an array of employees and add the employees with different form of payment.
- Arrange the entire sequence of workers descending the average monthly wage. Output the employee ID, name, and the average monthly wage for all elements of the list.

Homework

2. Develop and test a program's structure corresponding to the next schema



THANKS

softserve