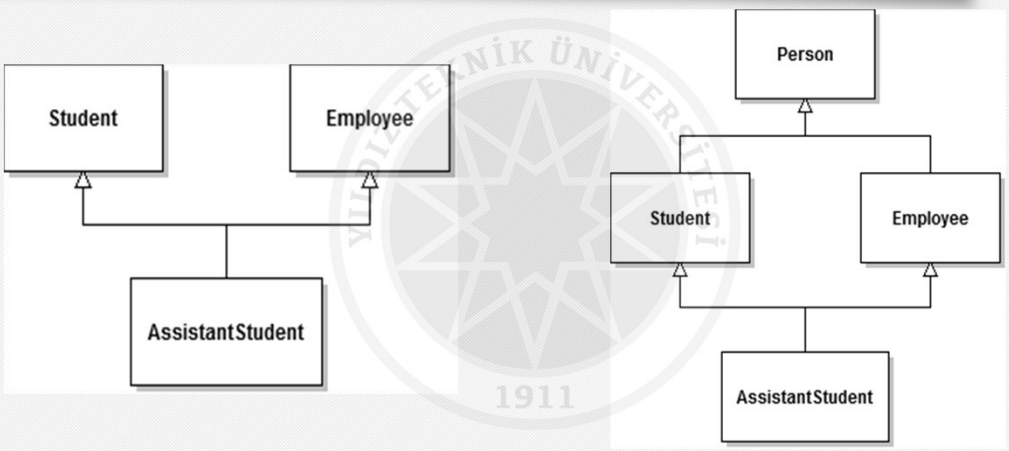
Nesneye Yönelik Programlam
BLM2012

Öğr. Grv. Furkan ÇAKMAK

Ders Tanıtım Formu ve Konular

Hafta	Tarih	Konular
1	01.03.2022	Dersin ve Java Dilinin Genel Tanıtımı, Sınıflar, Nesneler, Üyeler, Final ve Static Kavramları
2	08.03.2022	UML Sınıf Şemaları, Kurucular ve Sonlandırıcılar, Denetim Akışı, Nesneleri Oluşturulması
3	15.03.2022	Kurucuların ve Metotların Çoklu Tanımlanması, İlkeller, String ve Math Sınıfları
4	22.03.2022	Sahiplik ve Kullanma İlişkileri, Tek Yönlü ve İki Yönlü Sahiplik Kavramları
5	29.03.2022	Kalıtım, Metotların Yeniden Tanımlanması ve Çoklu Metot Tanımlamadan Farkı
6	05.04.2022	NYP'da Özel Konular: Abstract Classes, Interfaces, Enum Sınıfları
7	12.04.2022	Exception Handling, Unit Test
8	19.04.2022	1. Ara Sınav
9	26.04.2022	Temel Veri Yapılarının Jenerik Sınıflar Eşliğinde Kullanımı (Liste ve Eşleme Yapıları).
10	03.05.2022	Ramazan Bayramı
11	10.05.2022	Dosyalar ve Akışlar ile Çalışmak (Serileştirme ve Ters İşlemi)
12	17.05.2022	Tip dönüşüm, Enum Sınıfları, İç Sınıflar
13	24.05.2022	2. Ara Sınav
14	31.05.2022	Paralel Programlamaya Giriş

Multiple Inheritance



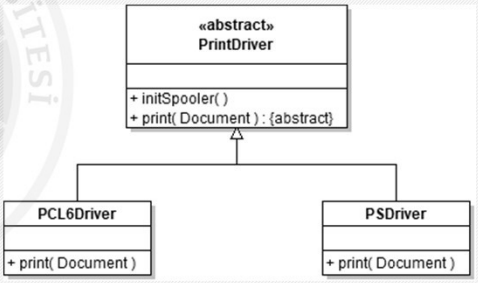
Abstract Classes

- An abstract class is such a class that it is used as a basee class and it represents a template for its regular sub classes.
  - If a class is abstract, we identify it with the keyword abstract.
- It is forbidden to create instances of an abstract class.
- Abstract classes can have member fields, just like the cconcrete classes.
- Abstract classes can have both concrete and abstract mmember methods.
  - An abstract method has only definition together with the keywword abstract, it does not have a body.

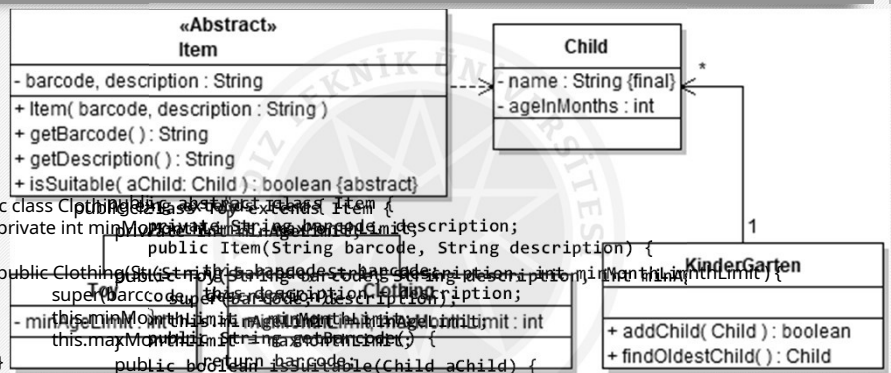
Abstract Classes (Con't)

- When do we need abstract classes?
- You can mark the abstract classes in UML class schemas in italics or by adding the «Abstract» stereotype.
  - <<...>>: This is called a stereotype.

```
public abstract class PrintDriver {
      public void initSpooler( ) {
     /* necessary codes*/
      }
      public abstract void print( Document doc );
}
public class PCL6Driver extends PrintDriver {
      public void print(Document doc) {
     //necessary code is inserted here
      }
}
```



Abstract Classes: Example



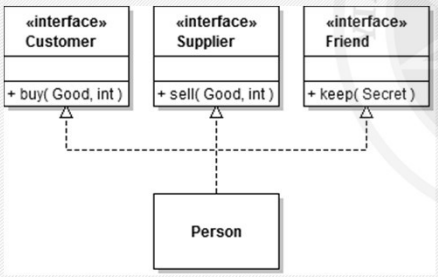

```
public class Clothing extends Item {
    private int minMonthLimit, maxMonthLimit;

public Clothing(String barcode, String description, int minMonthLimit) {
    super(barcode, description);
    this.minMonthLimit = minMonthLimit;
}
public boolean isSuitable(Child aChild) {
    if( aChild.getAgeInMonths() >= minMonthLimit && aChild.getAge() <= maxMonthLimit )
        return true;
    return false;
}
}
```

```
public Item(String barcode, String description) {
    this.barcode = barcode;
    this.description = description;
}
public String getBarcode() {
    return barcode;
}
public String getDescription() {
    return description;
}
public abstract boolean isSuitable(Child aChild);
}
```

Interfaces

- Interfaces can be thought as abstract classes **without m**members.
  - **If you wish, you may add "public final static" member fields o**nly.
- An interface is a named collection of methods.
- UML representation and source code of an example:



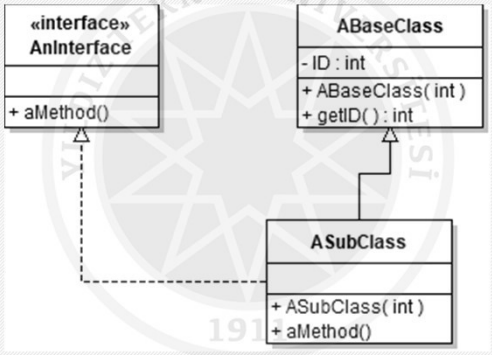

```
public interface Customer {
    public void buy( Good aGood, int quantity );
}
public interface Supplier {
    public void sell( Good aGood, int quantity );
}
public interface Friend {
    public void keep( Secret aSecret );
}
public class Person implements Customer,
        Supplier, Friend {
    public void buy( Good aGood, int quantity ) {
        //related code
    }
    public void sell (Good aGood, int quantity ) {
        // related code
    }
    public void keep( Secret aSecret ) {
        // related code
    }
}
```


Interfaces (Con't)

- We use interfaces ..
  - in order to group responsibilities of entities,
  - in order to give objects multiple views,
  - instead of inheritance,
    - Because inheritance is a "heavy weight" relation that should be used only when it is absolutely necessary.
  - instead of multiple inheritance.
- Rules related to interfaces:
  - A class should code the bodies of all the methods of the implemented interfaces.
  - Regular member fields cannot be defined in interfaces. Interfaces can only have "public final static" member fields.
  - Only public methods can be defined in interfaces.
  - Interfaces cannot have constructors.
  - A class can implement multiple interfaces.
  - Suggestion: Begin naming interfaces with I (capital i).

Interfaces (Con't)

- Interface implementation and inheritance can be used together when needed:

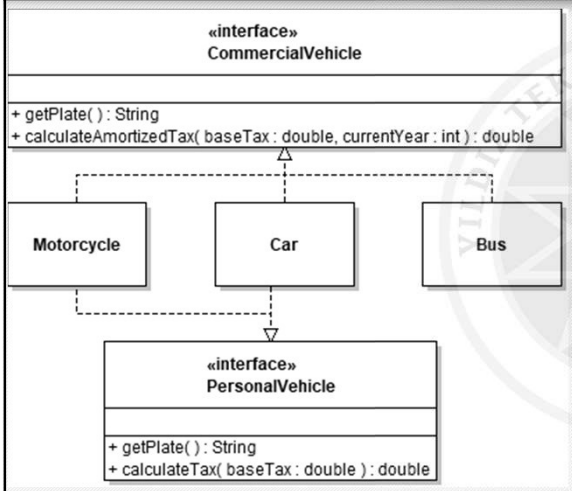



```
public class ASubClass extends ABaseClass implements AnInterfac
    // it should be be easy to code the rest for you
}
```


DESIGNING AND CODING INTERFACES

- Consider the following requirement about calculating the taxes of vehicles:
  - Taxation of commercial and personal vehicles is different.
  - Motorcycles, cars and buses can be registered as commercial vehicles.
  - Only motorcycles and cars can be registered as personal vehicles .
  - Only taxes of commercial vehicles can be amortized.
  - Commercial or not, calculation of the tax of different vehicles (car, bus, eetc.) are very different.
- How can we model this requirement?
- Hint: If the tax calculation for different vehicles were similar (i.e. parametrized), using one abstract base class instead of interfaces would be a better choice.
- PS: We will code a year limit for maximum amortizement in the CommercialVechile interface but VioletUML doesn't let us draw this in here.

DESIGNING AND CODING INTERFACES (CON'T)

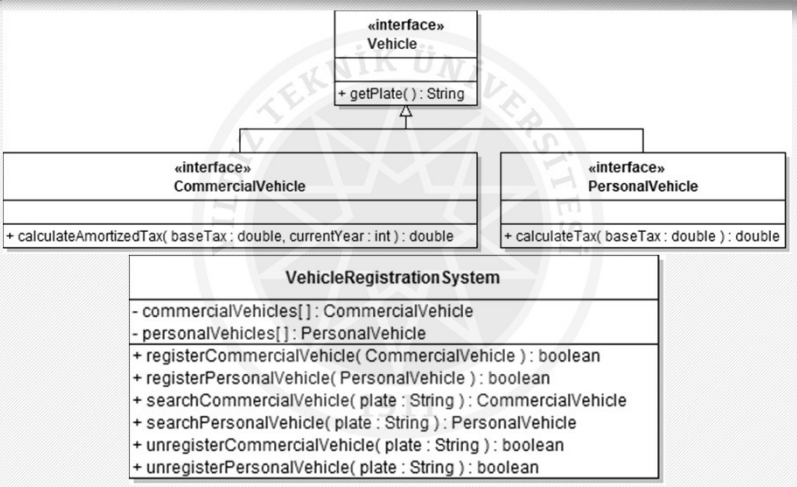



```
public class Bus implements CommercialVehicle, PersonalVehicle {
    private String plate;
    private int modelYear;
    private double tonnage;
    private double engineVolume;

    public Car(String plate, int modelYear, double tonnage, double engineVolume) {
        this.plate = plate;
        this.modelYear = modelYear;
        this.tonnage = tonnage;
        this.engineVolume = engineVolume;
    }
    public double calculateAmortizedTax(double baseTax, int currentYear ) {
        double ratioTonnage, ratioAge;
        if( tonnage < 1.0 )
            ratioTonnage = 1.0;
        else if( tonnage > 10.0)
            ratioTonnage = 2.0;
        else
            ratioTonnage = 1.4;
        int age = currentYear - modelYear;
        if( age < CommercialVehicle.yearLimit )
            ratioAge = (double) age * 0.05;
        else
            ratioAge = (double) CommercialVehicle.yearLimit;
        return baseTax * ratioTonnage * ratioAge;
    }
    public String getPlate() { return plate; }
    public int getModelYear() { return modelYear; }
    public double getEngineVolume() { return engineVolume; }
    public double getEngineVolume() { return tonnage; }
}
```

```
public interface CommercialVehicle {
    public final static int yearLimit = 10;
    public String getPlate();
    public double calculateAmortizedTax(double baseTax, int currentYear );
}
public interface PersonalVehicle {
    public String getPlate();
    public double calculateTax(double baseTax);
}
```


DESIGNING AND CODING INTERFACES (CON'T)



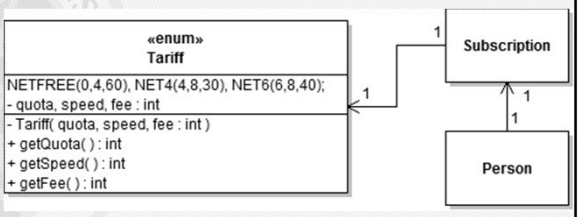
PRIMITIVE ENUMERATIONS and ENUM CLASSES


```
public enum Size {
    SMALL, MEDIUM, LARGE, EXTRA_LARGE;
}

Size s = Size.MEDIUM;
```

```
public enum Tariff {
    NETFREE(0,4,60), NET4(4,8,30), NET6(6,8,40);
    private int quota, speed, fee;
    private Tariff( int quota, int speed, int fee ) {
        this.quota = quota; this.speed = speed; this.fee = fee;
    }
    public int getQuota() { return quota; }
    public int getSpeed() { return speed; }
    public int getFee() { return fee; }
}

public class Test {
    public static void main(String[] args) {
        Tariff tariff4 = Tariff.NET4;
        Person yunus = new Person("Yunus Emre");
        yunus.subscribeTo(tariff4);
        Person berkin = new Person("Berkin Gülay");
        berkin.subscribeTo(Tariff.NETFREE);
        System.out.println(yunus);
        System.out.println(berkin);
    }
}
```



Sabırla Dinlediğiniz İçin Teşekkürler