BLM 2911 - OBJECT ORIENTED CONCEPTS
September 2017
Asst. Prof. Dr. Yunus Emre SELÇUK
GENERAL INFORMATION

SCORING

- 1st midterm: %20
- 2nd midterm: %20
- Midterm makeup
- Final exam: %40
- Project: %10 **(TBA)**
- Lab: %10

- Highlights (TBA: to be announced) :
  - Dedicated midterm weeks **(TBA)**
  - Exams are held in common between groups
  - Students who have previously failed are exempt from labs (except who got F0)
  - If a student has lab exemption / project is not announced, lab grade / project percentage will be distributed to exams

SUGGESTED BOOKS:

- Java Programming:
  - Java How to Program, Harvey M. Deitel & Paul J. Deitel, Prentice-Hall.
    - 7th ed. or newer
  - Core Java 2 Volume I, C. S. Horstmann and G. Cornell, Prentice-Hall.
    - 7th ed. or newer
- UML:
  - UML Distilled, 3rd ed. (2003), Martin Fowler, Addison-Wesley.

1

GENERAL INFORMATION

COURSE OUTLINE

- General Outline of the Java Programming Language
- Objects and Classes
- UML Class Schemas
- Object State, Behaviour and Methods
- Object and Class Collaborations and Relations
- UML Interaction (Sequence) Diagrams
- Inheritance and Abstract Classes
- Interfaces and Multiple Inheritance
- Polymorphism, Method Overriding and Overloading

2

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

JAVA EXECUTION ENVIRONMENT

- Standard Edition (JSE):
  - Suitable for developing any kind of application except applications for mobile devices
- Micro Edition (JME):
  - Suitable for developing applications for mobile devices, smartphones, etc.
  - Contains a subset of libraries of JSE.
- Enterprise Edition (JEE):
  - Contains JSE and an application server software (App. server)
    - App. server gives several services to applications coded by JSE.
    - More complex applications such as multi-tiered applications, web services, etc. need these services.
    - Transaction support is one of these services.
  - The basic App. Server is named "Sun Java System Application Server".
  - But there are other compatible services as well:
    - IBM Websphere
    - BEA WebLogic
    - Apache Tomcat
    - …

3

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

JAVA EDITIONS

- The old and the new way of naming Java:

Developer Version (Old way)	Product Version (New way)
Java 1.0	
Java 1.1	
Java 1.2	Java 2 Platform
Java 1.3	Java 2 SE 3 (J2SE3)
Java 1.4	J2SE4
Java 1.5	J2SE5
Java 1.6 {Sun}	Java Platform Standard Edition, version 6 (JSE6)
Java 1.7 {Oracle}	Java Platform Standard Edition, version 7 (JSE7)
Java 1.8	Java Platform Standard Edition, version 8 (JSE8)

4

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

JAVA EDITIONS

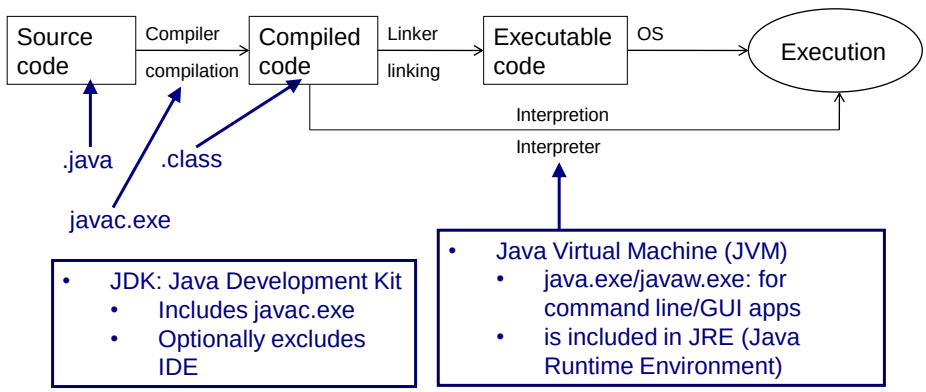
- Versioning details:
  - JDK 1.8.0_144:
    - Java Version 8.0, update 144.
    - Update:
      - Small bug fixes and security improvements.
      - Updated multiple times a year.
- Where to obtain:
  - oracle.com/java (scroll all the way down!)
    - http://www.oracle.com/technetwork/java/javase/downloads/index.html
  - Select the correct version: x86(i586) vs. x64
  - Download and unzip the documentation as well.

5

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

JAVA EXECUTION ENVIRONMENT

- Java is an interpreted language



- JDK is for developers and JRE is for end-users
  - JDK includes JRE

6

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

FREE JAVA DEVELOPMENT TOOLS

- IDE: Integrated Development Environment
- Eclipse: http://www.eclipse.org
  - Downloaded separately
  - You need to install a plug-in (such as eUML2) for drawing UML schemas.
  - You need to install a plug-in for writing GUI applications.
  - No need to have administrator rights on the computer, just unzip it.
- NetBeans:
  - Download separately or optionally with JSE.
  - You need to install a plug-in (suggestions?) for drawing UML schemas.
  - Has built-in GUI editor.
  - Needs administrator rights for installation.

FREE UML MODELING TOOLS

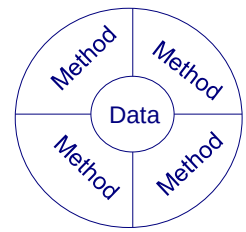
- Violet UML: Lightweight, enough for this course.
- Argo UML

7

CLASSES, OBJECTS AND MEMBERS

OBJECT

- Object: The main programming element.
  - Contains attributes and tasks.
  - Object ≈ a real-world entity.
    - Similar to variables but Superman is similar to mere mortals, too!
  - Attributes of an object ≈ Data about this entity.
  - Tasks ≈ actions ≈ methods
    - Similar to functions but …
    - Each function can access:
      - Any attribute of an object and,
      - Any given parameter but …
      - … there are many rules!
      - Our purpose is to master these rules.



- Encapsulation: The data and the methods of an object cannot be separated.
  - Data is accessed through methods.

8

CLASSES, OBJECTS AND MEMBERS

CLASS

- A class is just a template which defines objects.
  - The program is coded as classes, but the real work is done by the objects.
  - You may think a class as a cookie cutter and think objects as cookies!

```
class myClass {
   //program code
}
```

9

CLASSES, OBJECTS AND MEMBERS

OBJECTS AND CLASSES

- An example object: A particular car.
  - Attributes: Model, license plate number, color, etc.
    - Usually, one of the attributes of an object is determined as its logical unique identifier (UID).
    - Such as the plate number of a car.
  - Actions: Query a car about its license plate number, to sell this car, etc.
- An example class: Automobile.
  - A program code which defines the attributes and methods of cars.
- You can create any number of classes from any different classes within an object oriented program.

10

CLASSES, OBJECTS AND MEMBERS

OBJECTS AND CLASSES

- The attributes of an object can be conceptually divided into two groups:
  - Primitives: One unit of information such as integer numbers, real numbers and boolean values.
  - Non-primitives: Any number of objects from any number of classes.

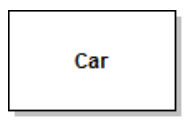
11

CLASSES, OBJECTS AND MEMBERS

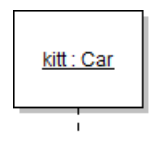
TERMINOLOGY AND REPRESENTATION

- NYP Terminology:
  - Data = **Member field** = field = attribute
  - State: Set of values of all attributes of an object
  - Task = Action = function = Method = **Member method**
  - **Members** of a class/object = Methods + fields
  - **Class** = type.
  - If o is an object of class C, we can also say that o is an **instance of** S.

- UML Representation:



A class shown in a class diagram



An object shown in a sequence diagram

12

CLASSES, OBJECTS AND MEMBERS

TERMINOLOGY AND REPRESENTATION

- There are two kinds of UML interaction diagrams:
  1. Sequence diagrams
  2. Collaboration diagrams
- We will draw sequence diagrams in this course,
  - The name "interaction" reflects the nature of these diagrams so well that I may use "interaction" and "sequence" interchangebly.

13

CLASSES, OBJECTS AND MEMBERS

EACH OBJECT IS A DIFFERENT INDIVIDUAL!

- Consider two objects of the same type:
  - Although both have the same type of attributes, the values of these attributes will be different = The **state**s of these objects will be different.
  - Even if you create two objects having the same state, these two objects will be represented in different areas of the memory.
    - In Java, the JVM creates a unique identifer for this purpose. This process, as well as the other memory management processes, are transparent to the programmer.
      - so transparent that you cannot interfere with
- Example: Any two cars cruising in the street.
  - Some attributes: Model, color, license plate.
  - The models and the colors of these cars will be different.
  - Even if you see the same yellow Anadol STC's, their license plates will be different.
    - Even if there is a conteirfeiting in effect so that their license plates are the same, their drivers will be different!

14

CLASSES, OBJECTS AND MEMBERS

EACH OBJECT IS A DIFFERENT INDIVIDUAL!

- Two different objects will give different answers to the same message, even if they are of the same type.
  - Why? Because their states will be different.
  - Moreover, you can give different parameters to the same method.

15

CLASSES, OBJECTS AND MEMBERS

SENDING MESSAGES TO OBJECTS

- Why do we send a message to an object?
  - In order to have this object to do something
  - To access a member of this object

MEMBER ACCESS

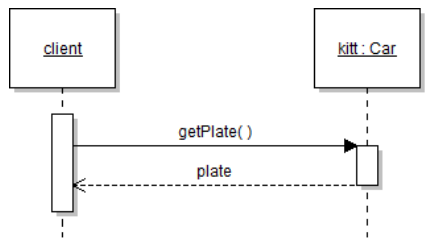
- We access a member field of an object in order to:
  - Change its value (setting)
  - Read its value (getting)
- We access a member method of an object in order to :
  - Run a method, optionally with some parameters
  - Calling a method is similar to calling a function in C.
    - But remember: Unless otherwise, a method of an object works with the members of this object.
      - How come otherwise?
        - Wait until you learn the different kinds of relationships between objects.

16

CLASSES, OBJECTS AND MEMBERS

TERMINOLOGY AND REPRESENTATION

- OOP Terminology:
  - An object accesses a member of another object by sending a message to that other object
  - An object oriented program executes as passed messages between objects.
- UML Representation:



- Code representation:

```
kitt.getPlate();
```

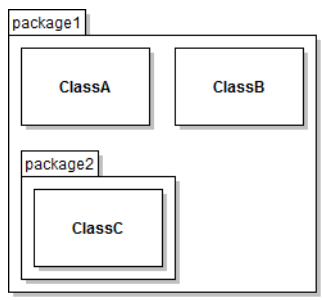
- Meaning of this figure:
  - There is an object named client
  - The class of the client is irrevelant
  - There is an object named kitt
  - Car is the class of the object kitt
  - The class Car has a method named getPlate
  - The client sends the message getPlate to the object kitt
  - the object kitt returns its license plate as the answer to the message

17

CLASSES, OBJECTS AND MEMBERS

PACKAGES

- Classes can be grouped together in abstract containers named packages
- The aim is to group such classes that can be used for a particular common purpose.



- Adding classes in a package to our code:

```
import package1.ClassA;
import package1.*;
import package1.package2.*;
```

  - Classes of package2 are not included when package1 is imported.
- Different classes in different packages may have the same name,
  - yet no conflicts arise.

```
java.io.File
com.fileWizard.File
```

- File path hierarchy must reflect the package hierarchy:

```
com.fileWizard.File -> com\fileWizard\File.java
```

18

CLASSES, OBJECTS AND MEMBERS

VISIBILITY RULES AND INFORMATION HIDING

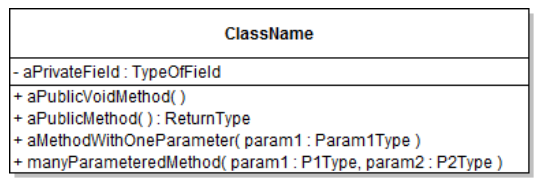
- An object can access all of its members and all members of other objects belonging to the same class (type).
- However, we can hide some members of an object so that they cannot be accessed from objects of different types.
- The information hiding principle:
  - We hide the members that are related with the inner workings of an object from objects of different types.
  - So that an object does not need to know the internal details of another object in order to use that other object.
- Example: It is sufficient to know the universal signs of power, volume and channel switching keys on a remote in order to watch TV.
  - You don't need to know that this TV has a device named cathod tube in it.
  - Moreover, the users need not to be re-educated for using TVs built with new technologies such as LCD, plasma, etc.
- Example: Your friend wants to lend some money from you.
  - You either open your purse and give him/her that money or not.
  - You don't have to tell anything about your salary or your PIN number to your friend!

19

CLASSES, OBJECTS AND MEMBERS

VISIBILITY RULES AND INFORMATION HIDING

- Access modifiers (Visibility rules):
  - public: There are no access restrictions to public members
  - private: Objects of different types cannot access each other's private members
- UML representation:



- Moreover (you are not responsible from those in this class):
  - protected: #
    - Related with inheritance (visible to package and subclasses)
  - package: ~
    - visible to package
    - Default rule in Java

20

CLASSES, OBJECTS AND MEMBERS

VISIBILITY RULES AND INFORMATION HIDING

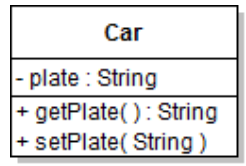
- In practice, the information hiding principle cannot be applied in a perfect way.
  - A change in code of a class not only affects that class but other classes that are related with that class as well.
  - The further you comply with this principle, the easier your coding overhead becomes for completing this change as the number of the affected classes will reduce.
- In order to comply with the information hiding principle:
  - Member fields are defined as private, and…
  - …the necessary access methods are defined as public.
  - At least 5 points for each question will be deduced if you don't comply!
- Access methods (accessors):
  - Setter method: Used for changing the value of a member field of an object.
  - Getter method: Used for reading the value of a member field of an object.
  - Naming convention: getMember, setMember

21

CLASSES, OBJECTS AND MEMBERS

VISIBILITY RULES AND INFORMATION HIDING

- Example:



- You can easily change the permissions to member fields. For example:
  - If you need to restrict the modification of the license plate, remove the setPlate method from code.
  - If you need to permit only the classes in the same package to make this modification, change the visibility of setPlate to package.

22

CLASSES, OBJECTS AND MEMBERS

SPECIAL CASES OF MEMBERS

- Static member fields:
  - The state of each object, even though they are of the same class, is different.
  - However, in some cases, you may need to have **all** objects of a particular type to **share** a common member field.
  - In this case, you define this member field with the **`static`** keyword.
  - Static members are accessed via the class name such as **`ClassName.memberName`**, not via the objects.
  - Example: Each automobile has 4 tires.
- Static member methods:
  - Two different objects of the same type answer the same message differently.
  - However, in some cases, you may need to have **all** objects of a particular type to **share** a common behavior.
  - In this case, you define this member method with the **`static`** keyword.
  - You may only use static members of an object within a static method of this object.
  - They are accessed via the class name, i.e. **`ClassName.aMethod()`**

23

CLASSES, OBJECTS AND MEMBERS

SPECIAL CASES OF MEMBERS

- Final member fields:
  - You may need the value of a member field to stay constant.
  - In this case, you define this member field with the **`final`** keyword
  - You may assign a value to a final member of an object only once
    - This assignment is usually done when that object is created.
  - For example, the chassis number of a car is etched onto it when it is produced in the factory and it cannot be changed afterwards.
- Final member methods:
  - These cannot be overridden (inheritance will be taught later).

POINTS TO CONSIDER

- A member can be both final and static at the same time.
- Do not confuse final and static with each other:
  - Final: Only once
  - Static: Shared usage
- Shown in UML class schemas as: aMember : Type {final,static}

24

CLASSES, OBJECTS AND MEMBERS

CONSTRUCTORS AND FINALIZERS

- Constructor Method:
  - This method is executed explicitly by the programmer when an object is to be created.
  - Constructors are used for assigning the initial values of the member fields of an object.
  - We will pay a significant attention to constructors in this class.
- Finalizing method:
  - This method is executed implicitly by JVM when an object is to be destroyed.
  - The method name is finalize
    - It takes no parameters and it does not return anything.
  - Unlike C/C++, Java programmers mostly need not to handle memory management.
  - As a result, we will not study finalizer methods any more in this class.

25

CLASSES, OBJECTS AND MEMBERS

CONSTRUCTOR METHODS

- Rules for constructors:
  - They are public.
  - Their name is the same with the class
  - Although they are used to create an object,
    - You do not issue a return command within constructor body and
    - you do not give a return type to the constructor method.
  - It's the best place to assign values to final member fields.
  - They are used with the **`new`** keyword.

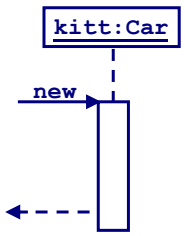
```
kitt = new Car();
```

26

CLASSES, OBJECTS AND MEMBERS

CONSTRUCTOR METHODS

- Defining an object is not enough to use it, you need to ***instantiate*** it by executing its constructor.
- UML Representation:



- Code representation1: As a member field

```
public class AClass {
    private Car kitt;

    someMethod( ) {
        kitt = new Car();
    }
}
```

- Code representation2: As a temporary object/variable

```
public class AnotherClass {
    someMethod( ) {
        Car kitt = new Car();
    }
}
```

27

CLASSES, OBJECTS AND MEMBERS

CONSTRUCTOR METHODS

- Default constructor:
  - The constructor without parameters.
  - JVM implicitly and automatically defines a default constructor if the programmer does not code any constructor.
- Constructors with parameters:
  - The parameters are used for assigning initial values.
  - If the programmer explicitly code a constructor with parameters, the default constructor is not automatically created by JVM.
    - In this case, it's up to the programmer to code a default constructor, if one required.
    - However, business logic often requires parametered constructors and forbids default constructors.
- A class can have more than one constructors having different types and numbers of parameters.
  - This is called constructor **overloading**.
  - Regular methods also can be overloaded.

28

CODING AN OBJECT ORIENTED PROGRAM

CONTROL FLOW

- Control flow is the order of execution of program codes.
  - In the lowest level, a computer program consists of various commands that are executed in a particular order.
  - The order that these commands are written and the order that they are executed are not necessarily the same.
    - In fact, especially in OOP, these two orderings are almost always quite different than each other.
  - Luckily, the starting point of this control flow is easier to determine.

29

CODING AN OBJECT ORIENTED PROGRAM

BEGINNING OF THE CONTROL FLOW

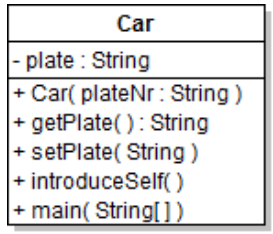
- The control flow of a program should have a starting point.
  - This point is a static method, named main, within a particular class that is determined by the programmer.
    - public static void main(String[ ] args)
      - The array args is used for passing initial parameters to the program from the command line.
      - static: It cannot be otherwise, because:
      - No object is created at the beginning of the control flow.
  - The task of the main method is to create the initial object(s) and to begin the execution of the program.
    - Remember, an OO program consists of messages sent between objects.
  - The existance of a main method in a class does not imply that this method will always be used.
- Terminology: Block/body: A piece of code having multiple instructions.
  - Shown between curly braces: { }

30

CODING AN OBJECT ORIENTED PROGRAM

CREATING YOUR OWN CLASS AND OBJECTS

- UML representation (class diagram)



- Source code (implementation)

```
package ooc01a;
public class Car {
   private String plate;
   public Car( String plateNr ) {
      plate = plateNr;
   }
   public String getPlate() {
      return plate;
   }
   public void setPlate(String plate) {
      this.plate = plate;
   }
   public void introduceSelf( ) {
      System.out.println( "My plate: " + getPlate() );
   }
   public static void main( String[] args ) {
      Car aCar;
      aCar = new Car( "34 RA 440" );
      aCar.introduceSelf( );
   }
}
```

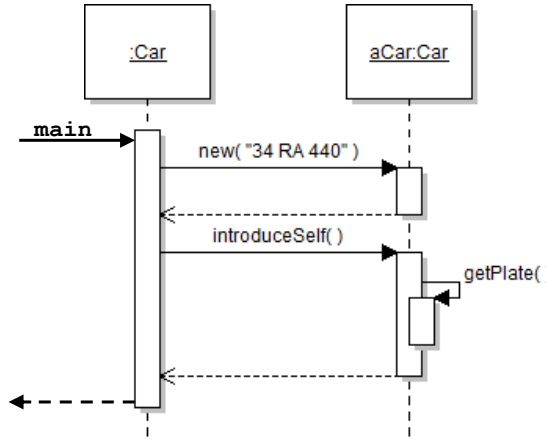
- Draw the class schema first.
- Then map the schema and the code
- Pretty printing, camel casing …

31

CODING AN OBJECT ORIENTED PROGRAM

UML REPRESENTATION

- Representation of the main method in the sample code by a sequence diagram, which is a kind of interaction diagram.
  - Pay utmost attention to the ordering and alignment of the arrows!

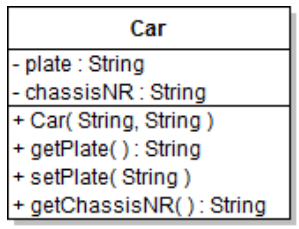


32

CODING AN OBJECT ORIENTED PROGRAM

CREATING YOUR OWN CLASS AND OBJECTS

- Another version of the class Car :




```
package ooc01b;
public class Car {
   private String plate;
   private String chassisNR;
   public Car( String plateNr, String chassisNR ) {
      plate = plateNr;
      this.chassisNR = chassisNR;
   }
   public String getPlate() {
      return plate;
   }
   public void setPlate(String plate) {
      this.plate = plate;
   }
   public String getChassisNR( ) {
      return chassisNR;
   }
}
```

- This second version of the class Car does not have a main method.
  - Therefore, it cannot be run and tested alone.
  - We need to code another class with a main method with these purposes (will be shown later).

33

CODING AN OBJECT ORIENTED PROGRAM

CREATING YOUR OWN CLASS AND OBJECTS

- Pay attention to the constructor:
  - In real world, every car must have a license plate AND a chassis number.
  - Therefore, both fields must be initialized in the same constructor having two parameters.
  - The code at the left is right, the code at the right is wrong.

```
public class Car {
   private String plate;
   private String chassisNR;
   public Car( String plateNr,
               String chassisNR ) {
      plate = plateNr;
      this.chassisNR = chassisNR;
   }
   /* Rest of the code */
}
```

```
public class Car {
   private String plate;
   private String chassisNR;
   public Car( String plateNr ) {
      plate = plateNr;
   }
   public Car(String chassisNR ) {
      this.chassisNR = chassisNR;
   }
   /* Rest of the code */
}
```

- Compile error vs. bug:
  - The code at the right does not compile. If it had, it's logic would be wrong (buggy).
- In real world, the chassis number of a car never changes. Therefore, we didn't code the getter method of that field. If we had, we would introduce another bug! (final fields can be mentioned shortly)

34

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

PRIMITIVE DATA TYPES

Name	Meaning	Range
int	Integer (4 bytes)	(– 2.147.483.648 , + 2.147.483.647) (± 2 billion)
double	Big and precise real number	**(± 1,7 E 308)** (for large numbers and precise calculations)
float	Small real number	**(± 1,7 E 38)** (for saving memory and faster calculations)
boolean	Logical	(false , true)

- Primitive: The basic data type representing a unit of information.
- Variable: Storage area for a primitive value
- Similar to objects, a primitive must be also defined before it can be used
  - int i = 7;
  - Unlike objects, primitives can be used before initialized.
    - Their default value is 0/false when not initialized.
- Pay attention to decimal point!
- boolean: Flag variable.

35

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

PRIMITIVE DATA TYPES

- Operations with primitives:
  - Arithmetic: + - * / %.
    - Operation priority
    - ++, --,
    - Difference between ++i and i++
    - Assignment and operation together: += –= *= /= %=
    - Keep it simple and minimal for sake of clarity
  - Logical: & | ! etc.
- I cut it short: Just like you have learned in previous year

36

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

STRING CLASS

- Some methods of the class String
  - int length()
  - int compareTo(String anotherString)
  - int compareToIgnoreCase(String str)
- System.out.println(String)
  - print / println
- Example:

```
package ooc01c;
public class StringOps01 {
   public static void main( String args[] ) {
       String strA, strB;
       strA = "A string!";
       strB = "This is another one.";
       System.out.println(strA.compareTo(strB));
   }
}
```

- Output of the example: −19

37

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

STRING CLASS (Continued)

- Some methods of the class String (continued):
  - boolean contains(String anotherString)
  - String toUpperCase( )
  - String toLowerCase( )
  - Note: toUpper/LowerCase methods do not change the state of the object.
  - Considering that note, what will the output of the code given in the next slide will be?

38

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

STRING CLASS (Continued)

```
package oop01c;
public class StringOps02 {
   public static void main( String args[] ) {
       String strA = "İstanbul", strB = "Yıldız";
       System.out.println(strA.contains(strB));
       strB = "tan";
       System.out.println(strA.contains(strB));
       strB.toUpperCase();
       System.out.println(strB);
       System.out.println(strA.contains(strB));
       strB = strB.toUpperCase();
       System.out.println(strB);
       System.out.println(strA.contains(strB));
   }
}
```

39

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MATH CLASS

- This class has static methods for common mathematical functions.
  - public static double Math.random( )
    - Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0
    - Example code:

```
package ooc01c;
public class MathOps01 {
    public static void main(String[] args) {
        double value = Math.random();
        System.out.println("The generated random value is: " + value);
    }
}
```

40

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MATH CLASS (Continued)

- This class has static methods for common mathematical functions (cont'd:)
  - variations of public static <primitive> Math.abs(<primitive> a)
    - returns the absolute value of parameter a where <primitive> is any primitive type, i.e.
    - public static double Math.abs(double a)
  - variations of public static <primitive> Math.max(<primitive> a, b)
    - returns the value of the greater of the two parameters where <primitive> is any primitive type, i.e.
    - public static double Math.max(double a, double b)
  - variations of public static <primitive> Math.min(<primitive> a, b)
    - returns the value of the smaller of the two parameters where <primitive> is any primitive type, i.e.
    - public static double Math.min(double a, double b)

41

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MATH CLASS (Continued)

- This class has static methods for common mathematical functions (cont'd:)
  - public static double Math.ceil(double a)
    - returns the upwards-rounded value of a (i.e. 3.1 → 4.0)
  - public static double Math.floor(double a)
    - returns the downwards-rounded value of a (i.e. 3.9 → 3.0)
  - public static double Math.round(double a)
    - returns the correctly-rounded value of a (i.e. 3.5 → 4.0, 3.1 → 3.0)
  - public static double Math.sqrt(double a)
    - returns the correctly rounded positive square root of a

42

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MORE ON RANDOM VALUES

- All computer systems rely on pseudo-random number generators.
  - It is modeled by java.util.Random class
  - If Math.random( ) is used directly, JRE automatically generates a Random object and uses it in the entire lifetime of the JVM
  - Random class has some useful non-static methods to obtain random values of desired primitives:
    - public boolean nextBoolean( ) returns [false, true]
    - public double nextDouble( ) returns [0.0, 1.0)
    - public float nextFloat ( ) returns [0.0, 1.0)
    - public int nextInt( ) returns [0, 2^{32})
    - public int nextInt(int bound) returns [0, bound)
    - public long nextLong( ) returns [0, 2^{48})

43

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MORE ON RANDOM VALUES

- Example code:

```
package ooc01c;
import java.util.*;
public class RandomOps {
    public static void main(String[] args) {
        Random generator = new Random();
        int intVal = generator.nextInt();
        System.out.println("I have got " + intVal + " pebbles.");
        int bounded = generator.nextInt(11);
        System.out.println("I have painted my " + bounded + " fingers.");
    }
}
```

44

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

COMMAND LINE I/O

- Output with System.out object:
  - The out member of System is a public and static member
    - The object out can therefore be used directly.
  - Methods for command line output:
    - printLn, print: We have learned those
    - printf: Used just as the C programmers know

45

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

COMMAND LINE I/O

- Input operations with the java.util.Scanner class: with JDK 5.0 and later!
  - Initialization: Scanner in = new Scanner(System.in);
  - System.in : A public static member of type java.io.InputStream.
  - Methods for obtaining input (one element at a time):
    - String nextLine()
    - int nextInt()
    - float nextFloat()
    - …

```
package ooc01c;
import java.util.Scanner;
public class ConsoleIOv1 {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      System.out.print("What is your name? ");
      String name = in.nextLine();
      System.out.print("How old are you? ");
      int age = in.nextInt();
      System.out.println("Hello, " + name +
      ". Next year, you'll be " + (age + 1) + ".");
      in.close();
   }
}
```

46

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

COMMAND LINE I/O

- A bug in the Scanner class:
  - If you get input for a string after getting input for a primitive by using nextInt, nextFloat, etc., that string goes to void!
  - As a workaround, issue an empty nextLine command in such cases.

```
package ooc01c;
import java.util.Scanner;
public class ConsoleIOv2 {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      System.out.print("How old are you? ");
      int age = in.nextInt();
      in.nextLine(); //workaround for the bug
      System.out.print("What is your name? ");
      String name = in.nextLine();
      System.out.println("Hello, " + name +
      ". Next year, you'll be " + (age + 1) + ".");
      in.close();
   }
}
```

47

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

COMMAND LINE I/O

- Let's change the main method so that the license plate of the car is obtained from the user:

Car
- plate : String
+ Car(plateNr : String) + getPlate( ) : String + setPlate(String) + introduceSelf( ) + main(String[ ])

```
package ooc01d;
import java.util.*;
public class Car {
   private String plate;
   public String getPlate() {
      return plate;
   }
   public void setPlate(String plate) {
      this.plate = plate;
   }
   public Car( String plateNr ) {
      plate = plateNr;
   }
   public void introduceSelf( ) {
      System.out.println( "My plate: " + getPlate() );
   }
   public static void main( String[] args ) {
      Car aCar;
      Scanner input = new Scanner( System.in );
      System.out.print("Enter a license plate: ");
      String plateNr =  input.nextLine( );
      aCar = new Car( plateNr );
      aCar.introduceSelf( );
      input.close();
   }
}
```

48

ALTERING THE CONTROL FLOW

- The structures you are familiar with since BBG2 also exists in Java with similar syntax.
- A short summary is given below. Refer to a Java book if you feel yourself uncomfortable with these satements.

DECISION MAKING – THE IF STATEMENT

```
if (condition) {...} else if (condition) {...} ... else (condition) {...}
```

- About the condition part:
  - Comparison: < > <= >= == !=
  - Double operator is used in logical operations: && ||

LOOPS

```
for( initialStatement; conditionStatement; incrementStatement ) { ... }
while( condition ) { ... }
do { ... } while( condition );
switch / case ...
```

49

RELATIONS BETWEEN OBJECTS

RELATIONS BETWEEN OBJECTS

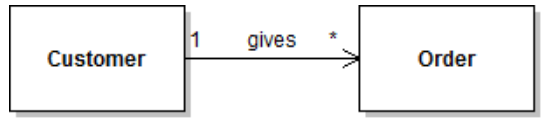
- We have learned that an object oriented program executes, i.e. runs, by sending messages to objects.
- In order to have an object to send a message to (i.e. use) another object, there must be some kind of relationship between these objects.
- Types of relations:
  - Association
  - Dependency
  - Aggregation
  - Composition
  - Inheritance
- These relations are shown in class diagrams but they should actually be read as relations between instances of classes, i.e. objects.

50

RELATIONS BETWEEN OBJECTS

ASSOCIATION

- The essence of association is **ownership**.
- The object that can send a message has the receiver of the message as a member field.
- Example: A customer and his/her orders
  - The logical name and the quantities of the relation are also shown in this diagram.

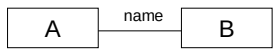


51

RELATIONS BETWEEN OBJECTS

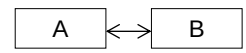
ASSOCIATION

- Representation:



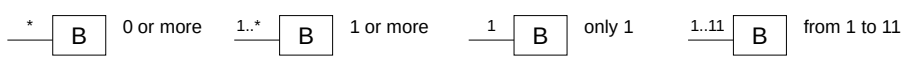
Association

A owns B = instances of A can send messages to instances of B



A owns B and B owns A = two-way connection

- The direction of the Arrow is important, it determines who can send messages to whom.
- If no arrows are drawn, this means:
  - either there is a two-way connection,
  - or the direction has not been considered by the architect yet.
- There may be numbers on the edges of the relation.
  - These numbers represent cardinality,
  - i.e. they show the number of objects at that edge's side.

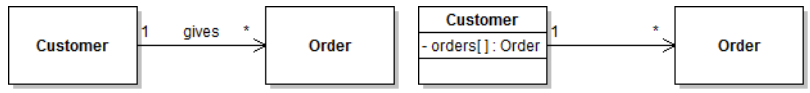


52

RELATIONS BETWEEN OBJECTS

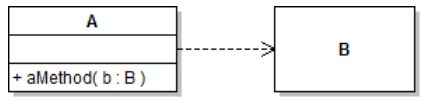
HIDDEN INFORMATION

- If a relation is shown by lines and arrows, you may omit the details within a class.
  - i.e., the two diagrams below are the same.



DEPENDENCY

- The essence of dependency is either **being a parameter of a method** or **temporary usage**, without ownership.
  - Representation:



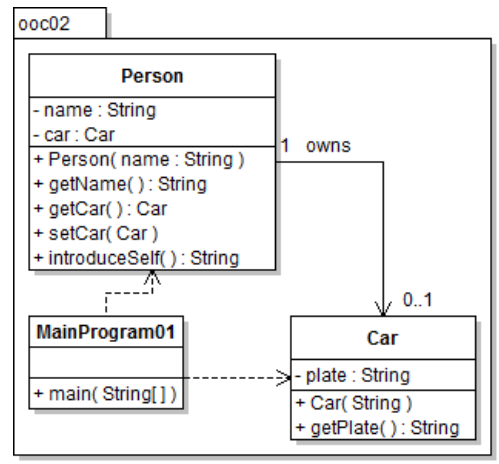
A depends on B = instances of A can send messages to instances of B in the body of aMethod.

53

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- Let's create a domain model where each person can have a car…
- …and include a program that uses the domain model (must have a main method in order to be run).
- The domain model and the program should reside in different packets.
- Question: Why 0..1 at the association?
- Hidden information: Check the constructor of class Car.



54

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- Source code of class Car:

```
package ooc02;

public class Car {
   private String plate;
   public Car( String plateNr ) {
       plate = plateNr;
   }
   public String getPlate() {
       return plate;
   }
}
```

- According to the code, a license plate number is assigned to a car when its created and this number cannot be changed.
- This was easy, lets move on to the class Person:

55

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- Source code of class Person:

```
package ooc02;
public class Person {
   private String name;
   private Car car;

   public Person( String name ) {
         this.name = name;
   }
   public String getName( ) { return name; }
   public Car getCar( ) { return car; }
   public void setCar( Car car ) { this.car = car; }

   public String introduceSelf( ) {
         String intro;
         intro = "Hello, my name is " + getName();
         if( car != null )
             intro += "and I have a car with license plate "
             + car.getPlate()+ ".";
         return intro;
   }
}
```

Attention!

56

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- Did you notice in the class diagram that the Car end of the ownership relation between Person and Car is 0..1?
  - This means that not every person may have a car.
- Moreover:
  - When you add a method to a class, there is no guarantee (*) about in what order the methods will run. They may even not be run anyway.
    - (*) except the special rules about constructors and the finalizer.
- As a result, one may create a person but he/she does not have to assign a car to that person.
  - How can one learn the license plate of his/her car when there is not any?
  - In this case, you will encounter with a "NullPointerException" error.
  - Our responsibility is to create solid (without errors and resistant to bugs) code. Therefore:
    - We should check whether a person has a car or not. We should access the license plate of his/her car only if he/she has a car.
      - If a person does not have a car, the value of that member field is `null`, meaning that this field is not assigned yet, i.e. it is not initialized.

57

CODING THE RELATIONS BETWEEN OBJECTS

TESTING FOR INITIALIZATION

- When an object is initialized, we can say that this object is now active.
- We can check whether an object1 is initialized or not as follows:

	Expression	Value
Initialized (active)	`object1 == null`	`false`
	`object1 != null`	`true`
Not initialized (inactive)	`object1 == null`	`true`
	`object1 != null`	`false`

58

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- We can code the MainProgram01 at last:

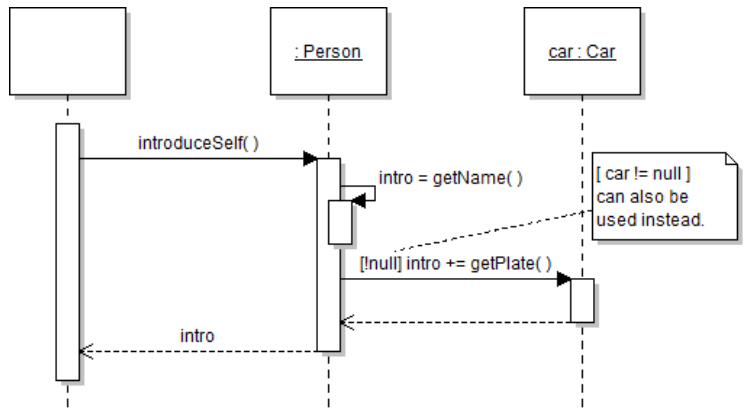
```
package ooc02;

public class MainProgram01 {
   public static void main(String[] args) {
       Person oktay;
       oktay = new Person( "Oktay Sinanoğlu" );
       Car rover = new Car( "34 OS 1934" );
       oktay.setCar( rover );
       System.out.println( oktay.introduceSelf() );
       Person aziz = new Person( "Aziz Sancar" );
       System.out.println( aziz.introduceSelf() );
   }
}
```

59

CODING THE RELATIONS BETWEEN OBJECTS

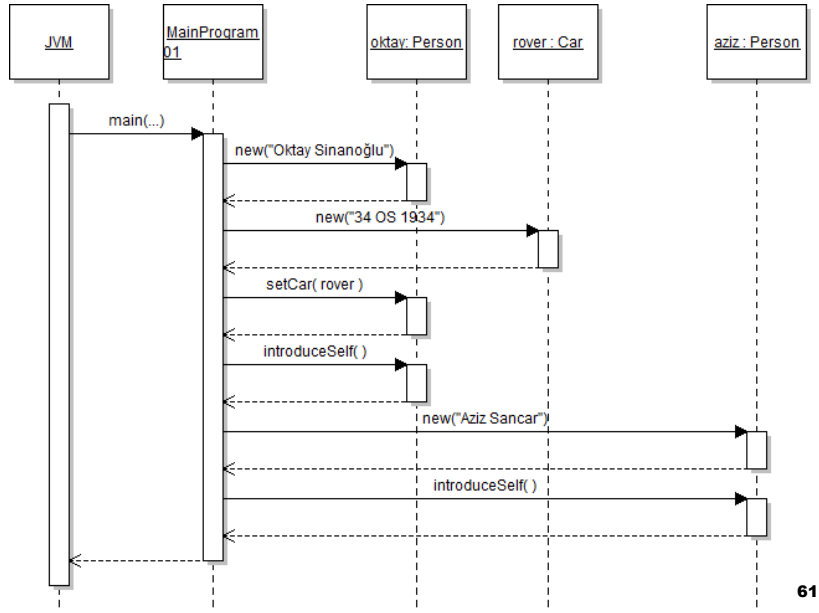
ASSOCIATION : ONE WAY

- Sequence diagram of Person.introduceSelf( ) method:



60

- The sequence diagram for the execution of MainProgram01

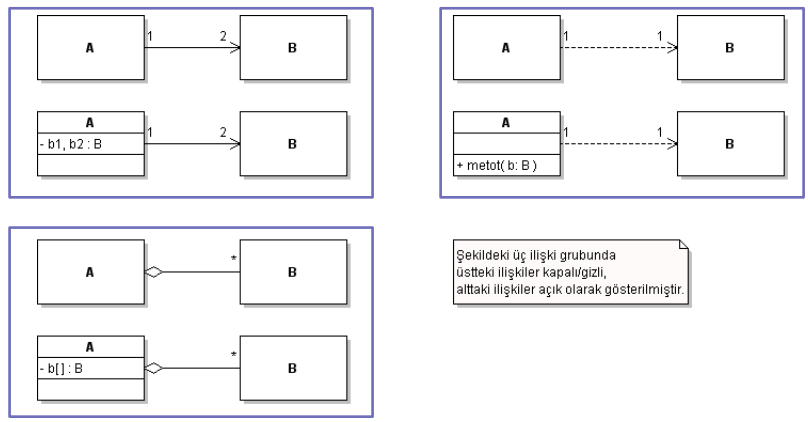


61

RELATIONS BETWEEN OBJECTS

HIDDEN INFORMATION

- Some implementation details may be hidden in class diagrams.
- In the 3 groups below, the pair on top implies the pair on bottom

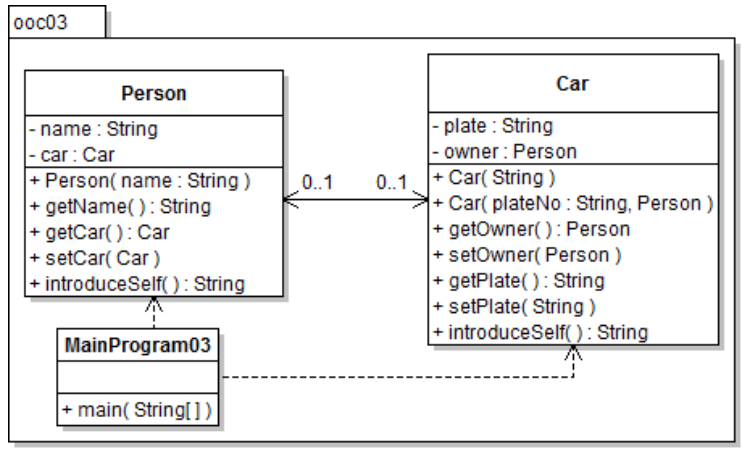


62

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- If we need to be able to find out the owner of a car, as well as being able to assign a car to a person, a two-way association must be constructed.



63

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Have you noticed the differences between one-way and two-way associations in the respective class schemas ?
  - We had to change the Car class.
  - The class Person stayed the same.
- We have put the new example into a different package as the Car class needed to be changed.
- About 0..1 on Person side:
  - A car object can be created without an owner in current design (because of Car(plate:String) constructor.
  - Do not be confused: 0..1 on Person side means that zero or more person can be associated with a car, that 0..1 is the quantity of car objects.

64

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- The source code of the new Car class:

```
package ooc03;
public class Car {
   private String plate;
   private Person owner;

   public Car( String plate ) { this.plate = plate; }
   public Car( String plate, Person owner ) {
         this.plate = plate;
         this.owner = owner;
   }
   public void setOwner( Person owner ) { this.owner = owner; }
   public Person getOwner() { return owner; }
   public String getPlate( ) { return plate; }
   public void setPlate( String plate ) { this.plate = plate; }
   public String introduceSelf( ) {
         String intro;
         intro = "[CAR] My license plate is " + getPlate();
         if( owner != null )
                  intro += " and my owner is " + owner.getName();
         return intro;
   }
}
```

Attention!

65

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Why did we have to code the if statement emphasized with the red?
  - Answer: Because one may call the constructor Car(String) and forget to call the setOwner method.
  - Should we remove the Car(String plate) constructor then?
    - No, a car does not have an owner in the real world as soon as it gets out of the factory.

66

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Let's try what we have done by coding a main method:

```
01  package ooc03;
02  public class MainProgram02 {
03  public static void main(String[] args) {
04          Person oktay = new Person("Oktay Sinanoğlu");
05          Car rover = new Car("06 OS 1934");
06          oktay.setCar(rover);
07          rover.setOwner(oktay);
08          System.out.println( oktay.introduceSelf() );
09          System.out.println( rover.introduceSelf() );
10
11          Person aziz = new Person("Aziz Sancar");
12          Car honda = new Car("47 AZ 1946");
13          aziz.setCar(honda);
14          honda.setOwner(aziz);
15          System.out.println( aziz.introduceSelf() );
16          System.out.println( honda.introduceSelf() );
17     }
18  }
19
20
```

67

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Can you see a problem in the main method?
  - Why do we have to code both the lines 6 and 7?
  - What if we forget writing any of those lines?
  - What if we mistakenly make a crossover between (oktay, rover) – (aziz, honda)?
  - etc.
- All those defects can be removed by making the two-way association stronger.
  - Which parts of the program should we change?

68

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Modifications to classes Person and Car (in a new package):

```
package ooc03b;
public class Person {
        /*the rest is the same*/
        public void setCar( Car car ) {
                this.car = car;
                if( car.getOwner() != this )
                        car.setOwner(this);
        }
}

package ooc03b;
public class Car {
        /*the rest is the same*/
        public void setOwner( Person owner ) {
                this.owner = owner;
                if( owner.getCar() != this )
                        owner.setCar(this);
        }
}
```

Attention!

Attention!

69

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

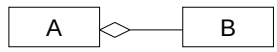
- How about giving the same flexibility to the second constructor of class Car?
  - Have the car to inform its owner in Car(String,Person)
  - (Instructor does that in class)
- Results:
  - Two-way relations are stronger than one-way relations but they are harder to code.
  - Therefore, if you don't need a two-way relation, code it only one way.
  - What if we need it two-way later?
    - Don't loose time with it, do it later. You will already be busy coding the other requirements.

70

RELATIONS BETWEEN OBJECTS

AGGREGATION

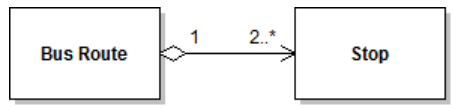
- Represents a weak **whole-part relationship**.
- Representation:



aggregation

- A instances has multiple B instances.
- A: Whole, B: Part.

- Even if it is not shown in the diagrams, aggregation implies the following:
  - 1 on the diamond end
  - * (multiplicity) and arrow on the other end
- Agrregation is stronger than association, but only conceptually.
  - It implies that this relation has stronger rules than a regular association.
  - For example, a bus route consists of at least 2 stops and there are rules for adding a new stop to a route.



71

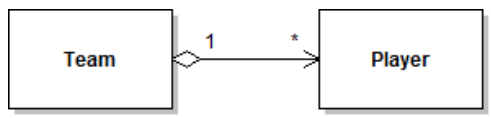
RELATIONS BETWEEN OBJECTS

COMPOSITION

- Similar to aggregation, but represents a **stronger whole-part** relation.

Composition

- The strength of composition over aggregation is that in composition, the part can only belong to one whole at the same time.
- Example:



72

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

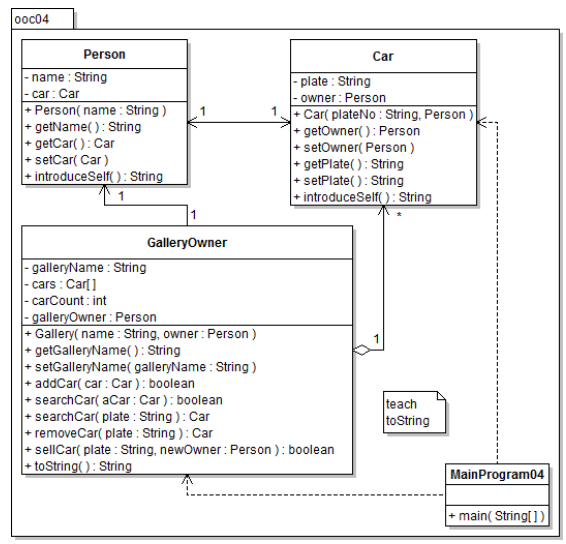
- Example: Let's create a Gallery class associated with multiple cars to be sold.
  - We will reuse the Car class, too.
    - You may want to add a constructor with having only the license plate as parameter. However, I opted for not doing this: We can assign the gallery an owner, a Person instance, as some kind of temporary owner.
  - I opted for representing the relation between the gallery and cars with the composition relation.
    - You can choose to show this relation with 1..* association, it does not matter.
    - This example also demonstrates the arrays and the for loop.

73

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- UML class diagram:



74

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- Source code of the class Gallery (to be cont'd in the next slides):

```
package ooc04;
public class Gallery {
        private String galleryName;
        private Car[] cars;
        private int carCount;
        private Person galleryOwner;

        public Gallery( String galleryName, Person galleryOwner ) {
            this.galleryName = galleryName; this.galleryOwner = galleryOwner;
            carCount = 0;
            cars = new Car[30];
        }
        public String getGalleryName() { return galleryName; }
        public void setGalleryName(String galleryName) {
           this.galleryName = galleryName;
        }
        public String toString( ) {
           String intro = "This is a car gallery named "+ galleryName;
           intro += ",  owned by " + galleryOwner.getName();
           intro += ". There are currently " + carCount + " cars to sell.";
           return intro;
        }
```

Note: No constructor is run here, this is just a memory allocation for the array

75

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- Source code of the class Gallery (cont'd):

```
public boolean addCar( Car aCar ) {
   if(!searchCar(aCar) && carCount < cars.length ){
      cars[ carCount ] = aCar;
      carCount++;
      return true;
   }
   return false;
}//end addCar
//code of class Gallery will continue in
```

You can learn the size of the array this way, but not the element count of the array. Therefore we don't need a variable such as maxCar, but we need a variable such as carCount.
Note:
public final static maxCar = 30;
private int carCount;

- In the next slides (you can try coding the following by yourself first):
  - It is wise to first check whether the car to be added already exists in the array by calling another method, searchCar.
  - How to sell a car? Implement the sellCar method.
- Excercise/HW: What if you need to take the money issues into account?

76

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- Source code of the class Gallery (cont'd):

```
public boolean searchCar( Car aCar ) {
    for( Car car : cars )
        if( car == aCar )
            return true;
    return false;
}
public Car searchCar( String plate ) {
    for( int i = 0; i < carCount; i++ )
        if( cars[i].getPlate().compareTo(plate) == 0 )
            return cars[i];
    return null;
}
```

- You can overload the searchCar method, too.
- Source code of the class Gallery will continue in the next slide.

77

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- Source code of the class Gallery (cont'd):

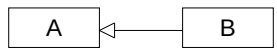
```
public Car removeCar( String plate ) {
    for( int i = 0; i < carCount; i++ ) {
        if( cars[i].getPlate().compareTo(plate) == 0 ) {
            Car theCar = cars[i];
            for( int j = i; j < carCount; j++ )
                cars[j] = cars[j+1];
            cars[carCount] = null;
            return theCar;
        }
    }
    return null;
}
public boolean sellCar( String plate, Person newOwner ) {
    Car soldCar = removeCar(plate);
    if( soldCar != null ) {
        soldCar.setOwner(newOwner);
        return true;
    }
    return false;
}
} //end class Gallery
```

78

RELATIONS BETWEEN OBJECTS

INHERITANCE

- Real world: A child inherits genetic properties from his/her parents.
- OOP: A means of creating new classes from an existing class, in a way that is similar with the real world.
- Representation:



Inheritance

- Pay attention to the direction of the arrow!

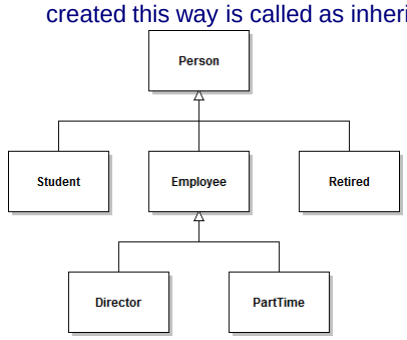
- A:
  - Parent class
  - Super class
  - Base class
- B:
  - Child class
  - Sub class
  - Derived class
- How inheritance works:
  - All member fields and methods of the parent are transferred to the child
    - However, children cannot access the inherited private members
  - Protected members and inheritance:
    - Those members can be accessed by children but they are inaccessible for other classes

79

RELATIONS BETWEEN OBJECTS

INHERITANCE

- Rules of the inheritance mechanism:
  - Sub classes cannot reject a member from the super class.
  - However, bodies of inherited methods can be changed
    - This is called **overriding**.
    - Attention: **final methods cannot be overridden.**
  - New members can be added to sub classes.
  - A sub class can be the parent of other classes. The tree structure created this way is called as inheritance hierarchy or as inheritance tree.



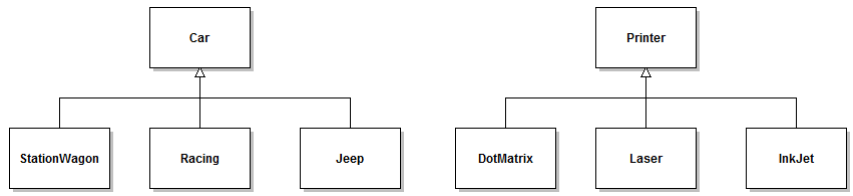
- Do not make the tree too deep
  - It leads to the fragile super class problem.
  - When you change the upmost super class, this change will affect all other sub classes.
  - This is similar to a tree that has a rotten root, it can easily collapse with the wind.

80

RELATIONS BETWEEN OBJECTS

INHERITANCE

- Effects of inheritance
  - Inheritance is also called as the generalization – specialization relation
    - The child is a specialized and more capable version of its parent.
    - Likewise, the parent is a generalized version of its children with less capabilities
  - Substitutability:
    - The child can be used wherever the parent is expected.
    - Therefore inheritance is also called as IS-A relationship.

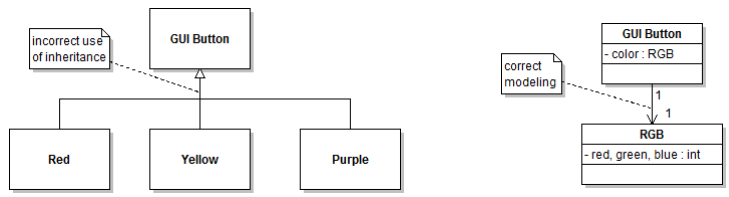


81

RELATIONS BETWEEN OBJECTS

INHERITANCE

- Misuse of inheritance

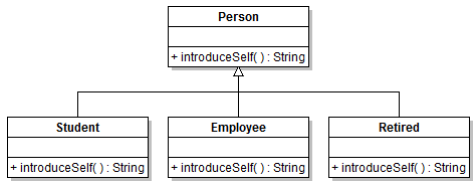


82

SPECIAL TOPICS in INHERITANCE

POLYMORPHISM and OVERRIDING

- We have learned that the body of an inherited method can be changed, and this act is named overriding.
  - Remember: final methods cannot be overridden.
- We have also learned that an instance of a subclass can be used wherever an instance of its superclass is expected.



- An example inheritance tree is shown on the left
  - The introduceSelf() method is overridden in the subclasses.

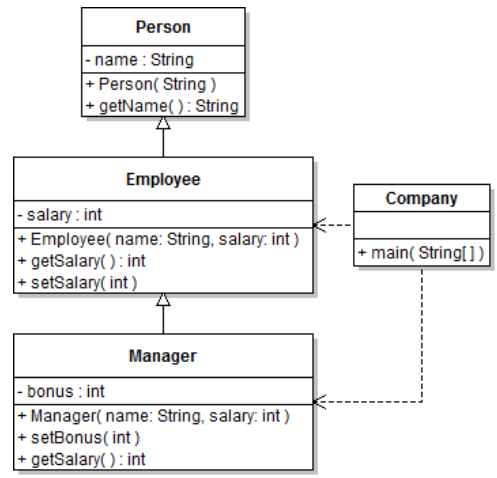
- Consider an array of type Person, having mixed instances of all classes above. What happens if we run the introduceSelf method of all members in the array?
  - The correct version of the introduceSelf method is executed in runtime.
  - This mechanism is called **polymorphism**.
- In this case, what should we do if we need to access an overridden method's previous version, i.e. as it is coded in the super class?
  - We can access that particular method via the `super` pointer!

83

SPECIAL TOPICS in INHERITANCE

POLYMORPHISM and OVERRIDING

- UML class schema of an example which consists of:
  - An inheritance tree (Person – Employee – Manager),
  - And a class using them (Company)



84

SPECIAL TOPICS in INHERITANCE

POLYMORPHISM and OVERRIDING

- Source codes:

```
package ooc05;
public class Person {
        private String name;
        public Person( String name ) { this.name = name; }
        public String getName( ) { return name; }
}
```

```
package ooc05;
public class Employee extends Person {
        private int salary;
        public Employee( String name, int salary ) {
                super( name );
                this.salary = salary;
        }
        public int getSalary( ) { return salary; }
        public void setSalary( int salary ) { this.salary = salary; }
}
```

Mention the important role of super

85

SPECIAL TOPICS in INHERITANCE

POLYMORPHISM and OVERRIDING

- Source codes (continued):

```
package ooc05;

public class Manager extends Employee {
    private int bonus;

    public Manager( String name, int salary ) {
        super( name, salary );
        bonus = 0;
    }
    public void setBonus( int bonus ) {
        this.bonus = bonus;
    }
    public int getSalary( ) {
        return super.getSalary( ) + bonus;
    }
}
```

Don't cause an error by writing:
super(name)
super(salary)

Remember visibility rules.
Cannot simply write:
salary + bonus

- Attention: You can use **super** only once. You cannot write **super.super**
- Attention: The **super** reference must be the first statement in the constructor..

86

SPECIAL TOPICS in INHERITANCE

POLYMORPHISM and OVERRIDING

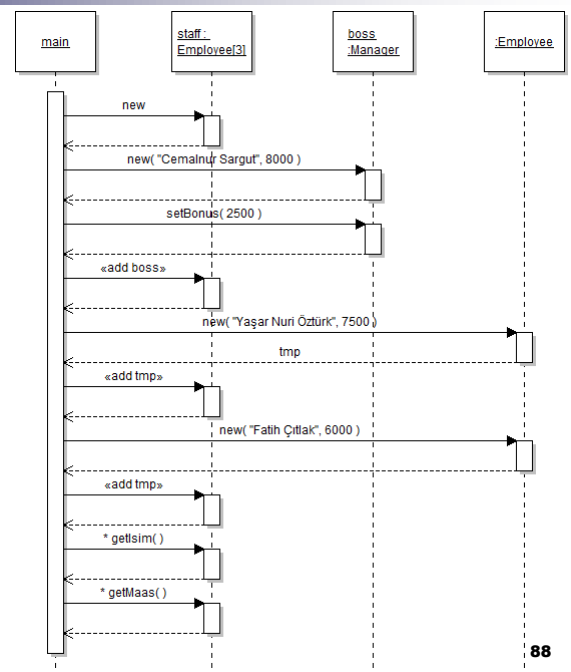
- Source codes (continued):

```
package ooc05;
public class Company {
    public static void main(String[] args) {
       Employee[] staff = new Employee[3];
       Manager boss = new Manager( "Cemalnur Sargut", 8000 );
       boss.setBonus( 2500 );
       staff[0] = boss;
       staff[1] = new Employee( "Yaşar Nuri Öztürk", 7500 );
       staff[2] = new Employee( "Fatih Çıtlak", 7000 );
       for( Employee author : staff )
           System.out.println( author.getName() + " " +
              author.getSalary( ) );
   }
}
```

- Have you noticed the syntax of the for loop?

87

- Sequence diagram of the main method:



88

SPECIAL TOPICS in OOP

OVERLOADING and OVERRIDING

- Do not confuse overriding and overloading:
  - Override: Modifying the body of inherited method. Overriding is closely related with inheritance.
  - Overload: Have multiple methods having same names but with different parameters. Overloading is not related with inheritance.
- Overriding example: The getSalary method in class Manager.
- Overloading example: Let's overload the constructor of the class Manager by adding the following constructor method:

```
public Manager( String name, int salary, int bonus ) {
       super( name, salary );
       this.bonus = bonus;
}
```

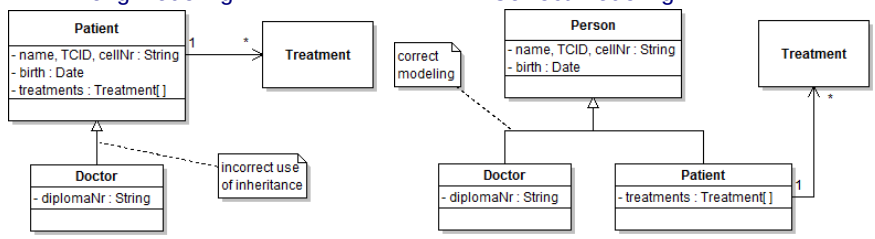
  - Now the class Manager has two constructors.

89

RELATIONS BETWEEN OBJECTS

INHERITANCE

- An excerpt from a requirements documentation:
  - We should keep track of the patients' names, TC ID numbers, birth dates and cell numbers. This information should be stored for doctors, too.In addition, it is required by law to keep the diploma numbers of doctors. We should keep records of treatments applied to a patient, too.
- Wrong modeling: Correct modeling:



- Patients should have treatments, not the doctors. In the incorrect case, the treatments array is transferred to doctors because of the inheritance relation.

90

CODING THE RELATIONS BETWEEN OBJECTS

INHERITANCE AND THE COSMIC SUPER CLASS IN JAVA

- In Java, the class java.lang.Object is the implicit super class of all classes.
- You can override some methods of this class in your classes for your own purposes:
  - public String toString(): You can return a string representation of the object that is easy for a person to read.
    - Just as you have done in `public String Car.introduceSelf()`
    - The advantage of overriding toString is that you can print the instance directly.

91

SPECIAL TOPICS in OOP

ABSTRACT CLASSES

- An abstract class is such a class that it is used as a base class and it represents a template for its regular sub classes.
  - Regular classes we have coded so far can be called concrete.
  - If a class is abstract, we identify it with the keyword **`abstract`**.
- It is forbidden to create instances of an abstract class.
- One can create instances of concrete subclasses of an abstract class.
- Abstract classes can have member fields, just like the concrete classes.
- Abstract classes can have both concrete and abstract member methods.
  - An abstract method has only definiton together with the keyword **`abstract`**, it does not have a body.
- The bodies of inherited abstract methods must be defined in the concrete subclasses.
  - Otherwise, those subclasses should also be defined as abstract.

92

SPECIAL TOPICS in OOP

ABSTRACT CLASSES

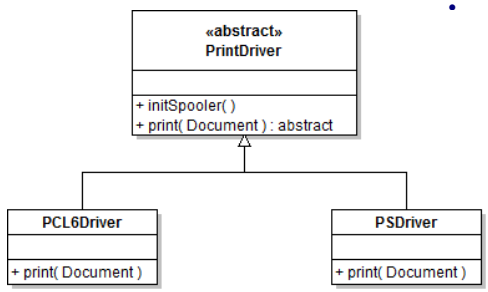
- When do we need abstract classes?
  - The more we climb upwards in a class hierarchy, the more the classes become generalized. At a point, the classes may become so generalized that we don't need them to be instantiated.
  - We said that you may use an abstract class as a template. In this case:
    - If you need to be make sure that a particular class must have some particular methods, you can define these methods in an abstract super class and you introduce an inheritance relationship between the aforementioned classes.
- You can mark the abstract classes in UML class schemas in italics or by adding the <<Abstract>> stereotype.
  - <<…>>: This is called a stereotype and used in any kind of UML schema whenever a symbol is used without its regular meaning.

93

SPECIAL TOPICS in OOP

ABSTRACT CLASSES

- Example:



- In the super class, we know how to initialize the spooler but we don't know the exact details of the printing process. We only know that a printer can print documents.
  - Therefore, PrintDriver instances are of no use to us.
  - However, the PrintDriver class removes the burden of coding the spooler initialization method from the coders of the subclasses.

- The details of the printing process are coded in the sub classes.
- Our design allows to install multiple printers of types PCL6, PS and printers of any other future types can be installed at the same computer. All those different printer types can be accessed in a uniform fashion represented by the PrintDriver class.

94

SPECIAL TOPICS in OOP

ABSTRACT CLASSES

- Source codes of the example:

```
public abstract class PrintDriver {
    public void initSpooler( ) {
       /* necessary codes*/
    }
    public abstract void print( Document doc );
}
```

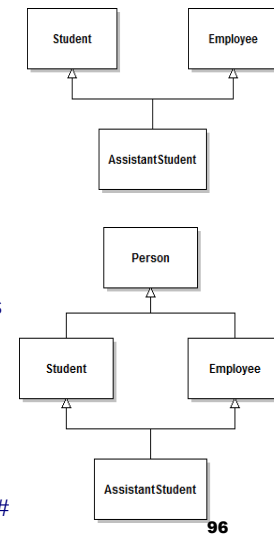
```
public class PCL6Driver extends PrintDriver {
    public void print(Document doc) {
       //necessary code is inserted here
    }
}
```

95

SPECIAL TOPICS in OOP

MULTIPLE INHERITANCE

- If a sub class can have more than one super class, this is called multiple inheritance.
  - For example, we have classes Student and Employee.
  - Our client asked for adding an AssistantStudent type to the software
  - At the first sight, it can be convenient to create this new class via inheriting from existing classes.
- However, multiple inheritance comes with an added complexity, i.e. the Diamond Inheritance Problem:
  - Consider the toString method, overriden in classes Student and Employee.
  - Which version of the toString method will be inherited by our new class?
  - In this case, we absolutely have to override the toString method in the new class.
- Due to such added complexities, multiple inheritance is not supported in recent languages such as Java and C#

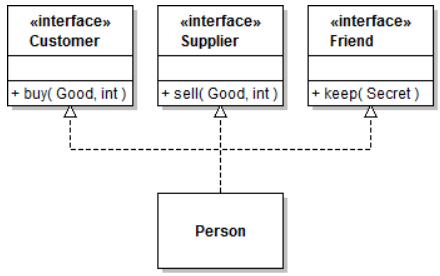


96

SPECIAL TOPICS in OOP

INTERFACES

- Interfaces can be thought as abstract classes without members.
  - If you wish, you may add "public final static" member fields only.
- 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
   }
}
```

97

SPECIAL TOPICS in OOP

INTERFACES

- We use interfaces …
  - in order to group responsibilites 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.

98

SPECIAL TOPICS in OOP

INTERFACES

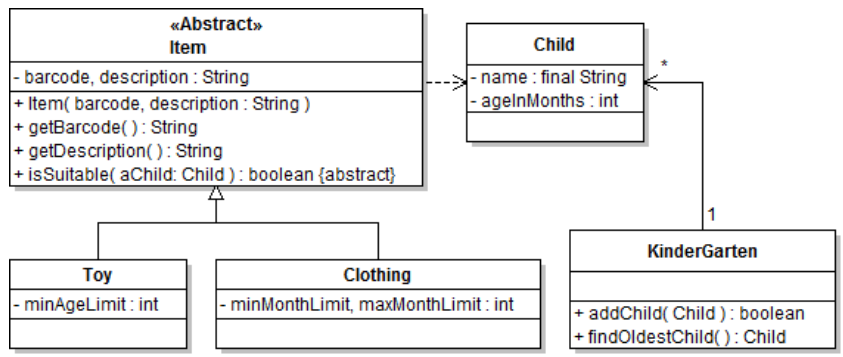
- Rules related to interfaces:
  - A class should code the bodies of all the methods of the implemented interfaces.
  - Regular member fields cannot be defined in interfaces. Interfaces can only have "public final static" member fields.
  - Only public methods can be defined in interfaces.
  - Interfaces cannot have constructors.
  - A class can implement multiple interfaces.
  - I suggest you to begin naming interfaces with I (capital i).

99

SPECIAL TOPICS in OOP

DESIGNING AND CODING AN ABSTRACT CLASS

- Consider items for children:
  - Not every item is suitable for every child.
    - Toys have lower age limit, usually measured in months
    - Clothes have both lower and higher age limits.
- How should we model this case?



100

SPECIAL TOPICS in OOP

DESIGNING AND CODING AN ABSTRACT CLASS

- Source code of class Item:

```
package ooc06;
public abstract class Item {
    private String barcode, description;
    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);
}
```

- The logic for determining the suitability of an Item is different for a Toy and a Clothing. Therefore we have left the isSuitable method as abstract here.
- However, we have coded the common operations in the abstract base class so that we don't have to code them again in sub classes.
  - "Say a word only once and at the right time! "

101

SPECIAL TOPICS in OOP

DESIGNING AND CODING AN ABSTRACT CLASS

- Source code of concrete subclasses:

```
package ooc06;
public class Clothing extends Item {
    private int minMonthLimit, maxMonthLimit;

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

102

SPECIAL TOPICS in OOP

DESIGNING AND CODING AN ABSTRACT CLASS

- Source code of concrete subclasses:

```
package ooc06;
public class Toy extends Item {
    private int minAgeLimit;

    public Toy(String barcode, String description, int minAgeLimit) {
        super(barcode, description);
        this.minAgeLimit = minAgeLimit;
    }
    public boolean isSuitable(Child aChild) {
        if( aChild.getAgeInMonths()/12 >= minAgeLimit )
            return true;
        return false;
    }
}
```

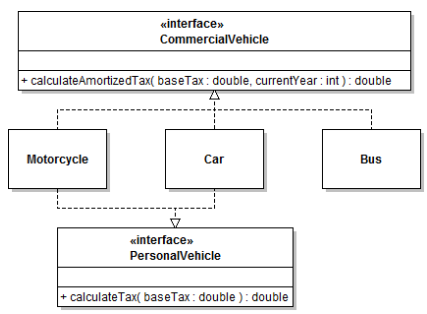
- You can implement the class Kindergarten with the given methods and more as exercise
- You can build different relationships between Item instances at one end and Kindergarten/Child at the other end(s)

103

SPECIAL TOPICS in OOP

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, etc) 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.

104

SPECIAL TOPICS in OOP

DESIGNING AND CODING INTERFACES

- Coding the interfaces:

```
package ooc07;
public interface CommercialVehicle {
    public double calculateAmortizedTax( double baseTax, int currentYear );
}
```

```
package ooc07;
public interface PersonalVehicle {
    public double calculateTax( double baseTax );
}
```

105

SPECIAL TOPICS in OOP

DESIGNING AND CODING INTERFACES

- Coding the class car:

```
package ooc07;
public class Car implements CommercialVehicle, PersonalVehicle {
    private String plate;
    private int modelYear;
    private double engineVolume;
    public Car(String plate, int modelYear, double engineVolume) {
         this.plate = plate; this.modelYear = modelYear;
         this.engineVolume = engineVolume;
    }
    public double calculateTax( double baseTax ) {
         return baseTax * engineVolume;
    }
    public double calculateAmortizedTax( double baseTax, int currentYear ) {
         //Tax can be reduced %10 for each year as amortization
         int age = currentYear - modelYear;
         if( age < 10 )
             return baseTax * engineVolume * (1-age*0.10);
         return baseTax * engineVolume * 0.10;
    }
    public String getPlate() { return plate; }
    public int getModelYear() { return modelYear; }
    public double getEngineVolume() { return engineVolume; }
}
```

106

SPECIAL TOPICS in OOP

DESIGNING AND CODING INTERFACES

- Coding the class bus:

```
package ooc07;
public class Bus implements CommercialVehicle {
    private String plate;
    private int modelYear;
    private double tonnage;
    public Bus(String plate, int modelYear, double tonnage) {
           this.plate = plate; this.modelYear = modelYear; this.tonnage = tonnage;
    }
    public double calculateAmortizedTax( double baseTax, int currentYear ) {
           double ratioT, ratioA;
           if( tonnage < 1.0 )
               ratioT = 1.0;
           else if( tonnage < 5.0 )
               ratioT = 1.2;
           else if( tonnage < 10.0 )
               ratioT = 1.4;
           else
               ratioT = 1.6;
           ratioA = (currentYear - modelYear) * 0.05;
           if( ratioA > 2.0 )
               ratioA = 2.0;
           return baseTax * ratioT * ratioA;
    }
    public String getPlate() { return plate; }
    public int getModelYear() { return modelYear; }
    public double getEngineVolume() { return tonnage; }
}
```

107

SPECIAL TOPICS in OOP

DESIGNING AND CODING INTERFACES

- The state of a car, bus or a motorcycle instance being a personal or commercial vehicle will be saved in a Container object:

VehicleRegistrationSystem
- commercialVehicles[ ] : CommercialVehicle - personalVehicles[ ] : PersonalVehicle
+ registerCommercialVehicle(CommercialVehicle) : boolean + registerPersonalVehicle(PersonalVehicle) : boolean + searchCommercialVehicle(plate : String) : CommercialVehicle + searchPersonalVehicle(plate : String) : PersonalVehicle + unregisterCommercialVehicle(plate : String) : boolean + unregisterPersonalVehicle(plate : String) : boolean

- How must the logic be? How should we implement that?

108

SPECIAL TOPICS in OOP

DESIGNING AND CODING INTERFACES

- If the tax calculation for different vehicles were similar (i.e. parametrized), using two abstract base classes instead of interfaces would be a better choice.
- Likewise, if the tax calculation for commercial and personal vehicles were similar, using only one abstract base class and choosing appropriate method parameters would be a better choice.
- Those cases are left as exercises to the students for experimenting with.

109