



School of Computer Science



Lesson 4: Object interaction



Introduction to Programming

Academic year 2023-2024

Topics

- Abstraction
- Modularization
- Creating objects
- Method calls
- Class diagrams
- Object diagrams
- Using the debugger

Introduction

- ❑ To build actual applications we cannot rely on **objects working on isolation**.
- ❑ **Objects must cooperate** to fulfill the common task.
- ❑ To **solve complex problems** we must identify **components** which can be transformed into **independent classes**.

Abstraction and Modularization

- **Modularization** is the process of **dividing a whole problem** into **well-defined parts** which can be built and examined separately.
- **Abstraction** is the **ability to ignore details** of parts to **focus attention** on higher levels of the problem.
- Modularization and abstraction mutually complement.

Abstraction and Modularization

- To see the “**big picture**” in complex programs you must:
 - **Identify components** as independent entities.
 - **Use components as simple parts** ignoring their internal complexity.
- In OOP both **components** and **subcomponents** are **objects**.

Example: A digital clock

```
public class NumberDisplay {  
    private int limit;  
    private int value;  
    // constructors and methods omitted  
}
```

Classes define new types. If a field (or variable) type is a class then that field can store references to objects from that class.

```
public class ClockDisplay {  
    private NumberDisplay hours;  
    private NumberDisplay minutes;  
    private String displayString; // simulates the display  
    // constructors and methods omitted  
}
```

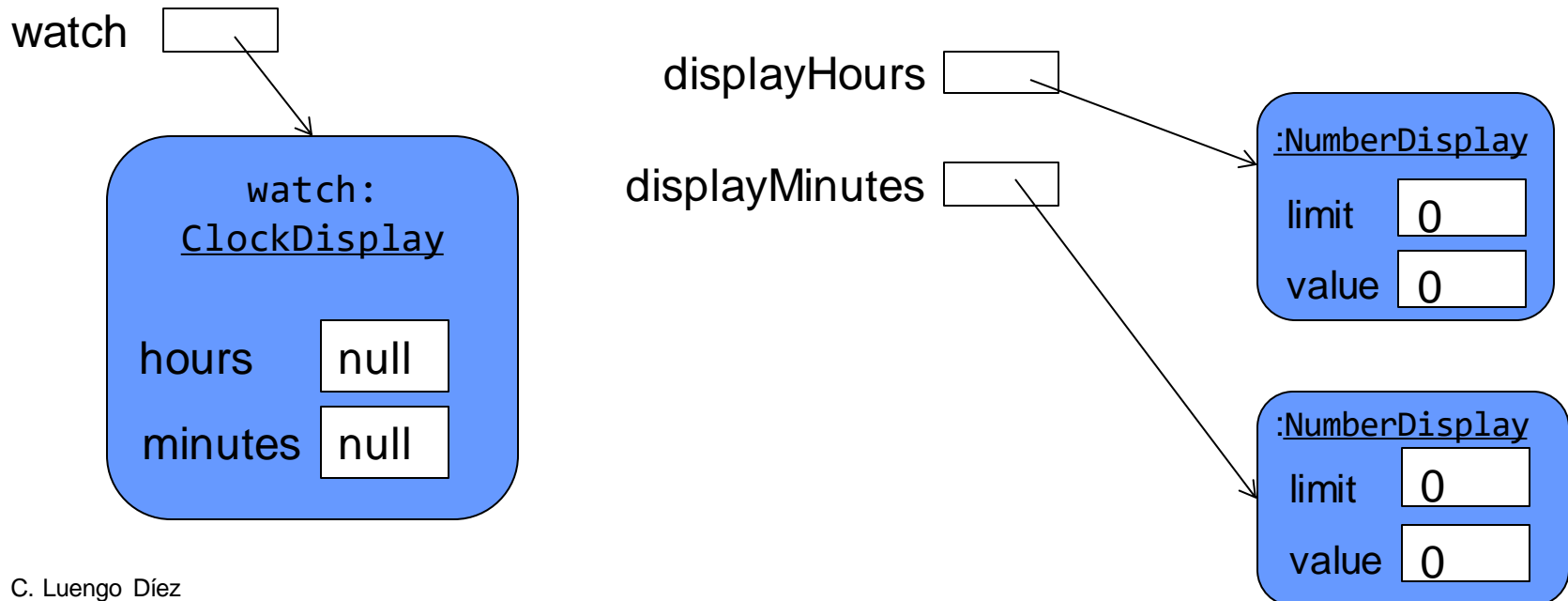
Objects

What objects are created when executing the following code?

```
ClockDisplay watch = new ClockDisplay();
```

```
NumberDisplay displayHours = new NumberDisplay();
```

```
NumberDisplay displayMinutes = new NumberDisplay();
```



```

public class ClockDisplay {
    private NumberDisplay hours;
    private NumberDisplay minutes;

    public ClockDisplay() {
        hours = new NumberDisplay (24);
        minutes = new NumberDisplay (60);
    }
    ...
}

```

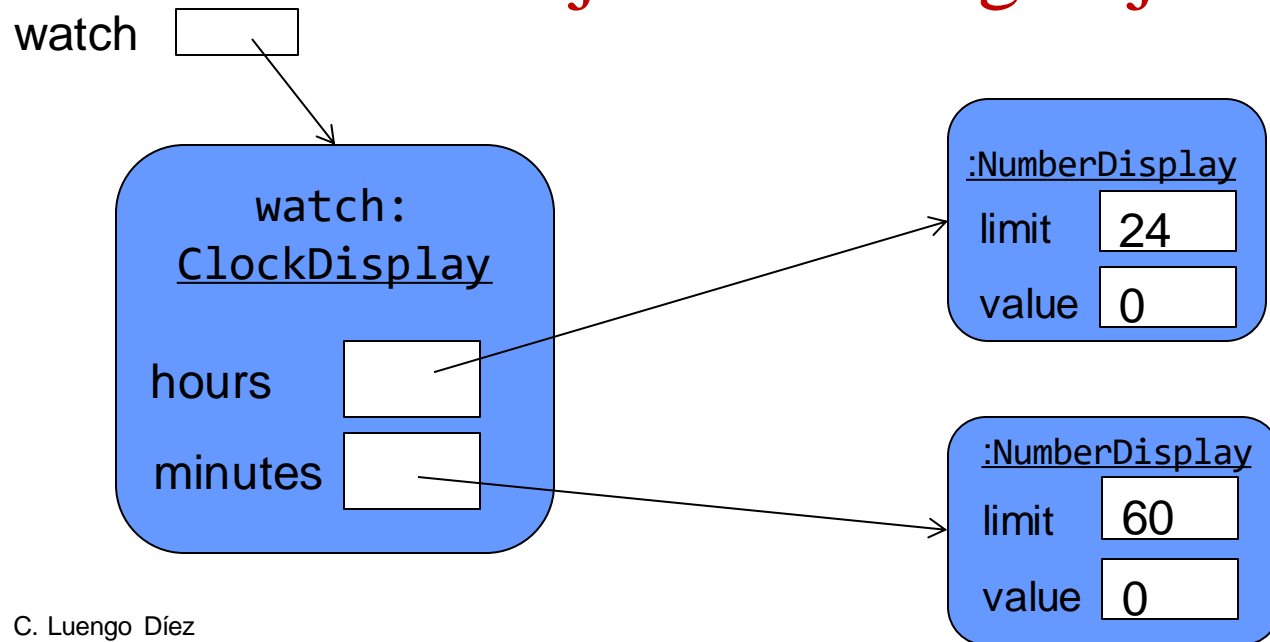
```

public class NumberDisplay {
    private int limit;
    private int value;

    public NumberDisplay (int maxLimit){
        setLimit(maxLimit);
        setValue(0);
    }
    ...
}

```

Objects creating objects



Multiple constructors

- **Overloading.** A class can provide several constructors or methods with the same name provided they have a different set of parameters to tell them apart.
 - Additionally, return data type can be different.

```
public ClockDisplay(){  
    hours = new NumberDisplay (24);  
    minutes = new NumberDisplay (60);  
}
```

```
public ClockDisplay(int hour, int minute) {  
    //hours = new NumberDisplay (24);  
    // minutes = new NumberDisplay (60);  
    this();  
    setTime(hour, minute);  
}
```

Method calls (I)

□ Internal method calls

- Methods can contain invocations to other methods **of the same class.**

Syntax: `methodName (parameter-list)`

Optionally: `this.methodName (parameter-list)`

- Example:

```
public ClockDisplay(int hour, int minute) {  
    this();  
    setTime(hour, minute);  
    // this.setTime(hour, minute)  
    // Using 'this' in the previous invocation is  
    // optional  
}
```

Method calls

```
public class ClockDisplay
{   private NumberDisplay hours;
    private NumberDisplay minutes;

    public ClockDisplay(int hour, int minute) {
        hours = new NumberDisplay (24);
        minutes = new NumberDisplay (60);
        setTime(hour, minute);
    }
    /**
     * Method that sets the hour on the screen
     * given the hour and minutes as parameters
     * @param hour new value for the hour
     * @param minute new value for the minutes
     */
    public setTime (int hour, int minute)
    {
        hours.setValue(hour);
        minutes.setValue(minute);
        this.updateDisplay(); // updates 'displayString'
    }
}
```

```
public class NumberDisplay {
    private int limit;
    private int value;
    // ...
    public NumberDisplay (int
                           maxLimit) {
        setLimit(maxLimit);
        setValue(0);
    }
}
```

```
public void setValue (int nValue) {
    if (nValue >= 0 &&
        nValue <= getLimit())
        value = nValue;
}
```

Methods of the
NumberDisplay class

Method calls (II)

□ External method calls

- Methods can call **methods on other objects** using the **dot . operator**.

Syntax:

`object.methodName(parameter-list)`

- Example:

Methods from the class
NumberDisplay

```
public void timeTick() {  
    // ClockDisplay method  
    minutes.increment();  
    if (minutes.getValue() == 0) // It is zero again!  
        hours.increment();  
    this.updateDisplay();  
}
```

The NumberDisplay class

```
public class NumberDisplay {
    private int limit;
    private int value;
    // ...
    public NumberDisplay (int maxLimit) {
        setLimit(maxLimit);
        setValue(0);
    }

    /**
     * Increments by 1 the displayed
     * value, setting it to zero
     * if the limit value is reached
     */
    public void increment() {
        setValue((getValue()+ 1) % limit);
    }

    public void setValue (int newValue) {
        if (newValue>=0 &&
            newValue <= getLimit())
            value = newValue;
    }

    public int getValue () {
        return value;
    }
}
```

Object interaction

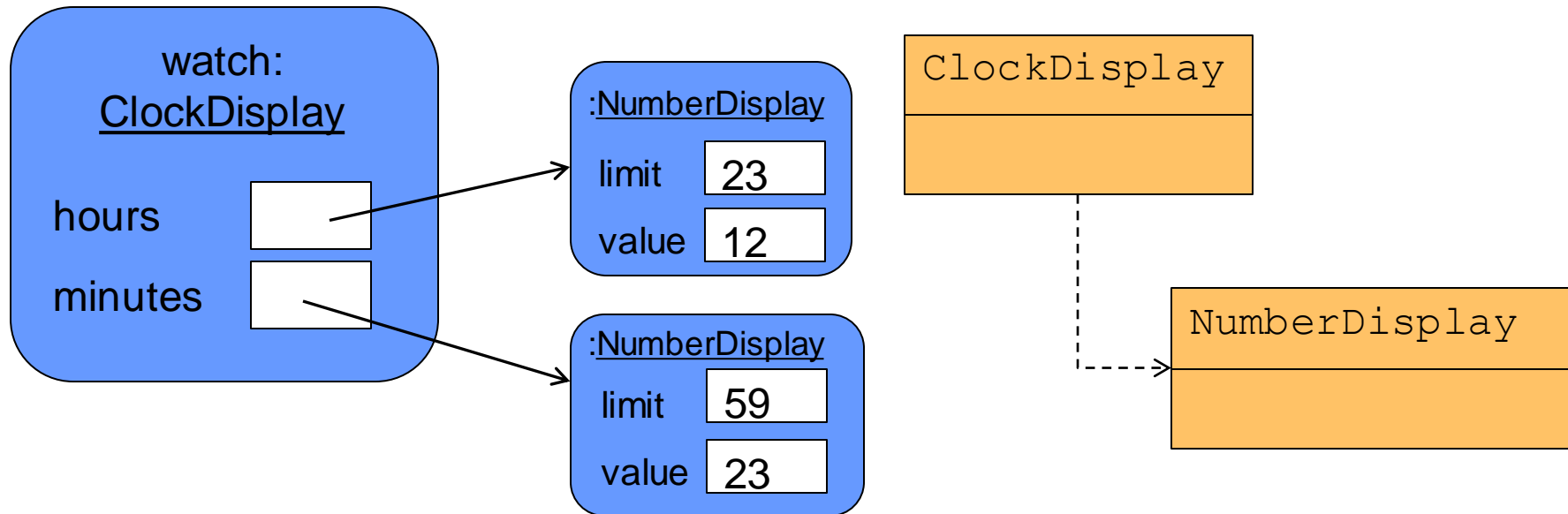
- Objects can **communicate among them** by **invoking (public) methods on other objects.**
- Objects can **create other objects** and **invoke their (public) methods.**

Example

```
public class Bike {  
    private Person owner;  
    private Wheel frontWheel;  
    private Wheel backWheel;  
    // ...  
    public Bike(Person owner) {  
        setOwner(owner);  
        setFrontWheel(new Wheel());  
        setBackWheel(new Wheel());  
    }  
  
    public void inflateTires() {  
        frontWheel.inflate(7.0);  
        backWheel.inflate(6.5);  
    }  
}
```

```
public class Wheel{  
    private float pressure; //bars  
    private int inches; //inches  
  
    // ...  
    public Wheel() {  
        setPressure(5.0);  
        setInches(26)  
    }  
  
    public void inflate(float pressure){  
        setPressure(pressure);  
    }  
}
```

Class diagrams and object diagrams



Object diagram

It shows the **objects and their relations** at a given moment of the program execution

It deals with the **dynamic** part of a program.

Class diagram

It shows the **classes** of a given program and the **relations** among them.

It deals with the **static** part of a program.

Class diagrams and object diagrams

- The **class diagram** provides a **static view**.

- The arrow shows that the `ClockDisplay` class uses the `NumberDisplay` class (it appears in the source code of `ClockDisplay`).

`ClockDisplay` depends on `NumberDisplay`

- The **object diagram** provides a **dynamic view**. That is, the situation at a given point of the execution.

Notation

- There are numerous notations.
- Nowadays, the most common for OO is **UML**.
 - UML (Unified Modelling Language) is a **notation**.
- It is a standard by the OMG (*Object Management Group*)



<http://www.uml.org/>

Notation: in a nutshell

□ Comments

This is a
comment about
the cloud

Whatever

□ Classes

MyClass

Book

title: String
isbn : String

getChapterTitle(chapter: int) : String

□ Objects

myobject :
MyClass

Notation (static view): Classes

- Class name
- Attributes / methods
- Types :
 - Attributes
 - Parameters
 - Return
- Visibility
 - Private: -
 - Protected: #
 - Public: +
- Stereotypes: << >>

| Dictionary |
|---|
| - editionDate: Date - title : String - sourceLanguage: String - targetLanguage: String |
| + <<constructor>> Dictionary + lookUpWord(word : String): String |

Notation (static view): Classes

Dictionary

- editionDate: Date
- title: String
- originLanguage: String
- destinationLanguage: String

+ <<constructor>> Dictionary
+ lookUpWord(word:String): String

```
public class Date
{ private int day;
  private int month;
  private int year;
  ...
```

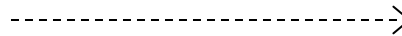
```
public class Dictionary {
  private Date    editionDate;
  private String  title;
  private String  originLanguage;
  private String  destinationLanguage;

  public Dictionary()
  {...}
  public String lookUpWord(String word)
  {...}
```

Notation: relations

A line implies a connection between classes

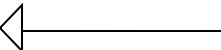
Dependency (use)

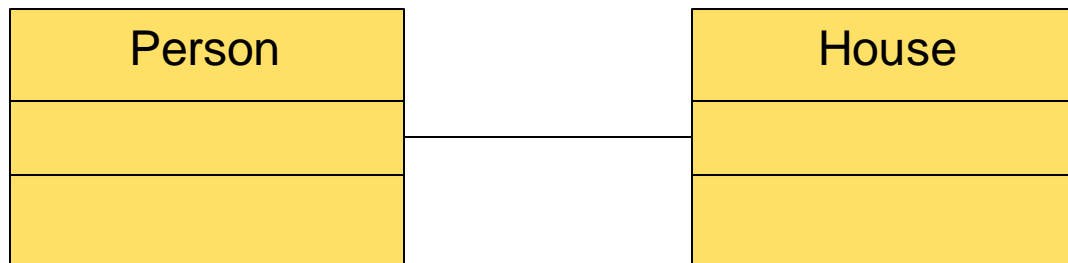


Association

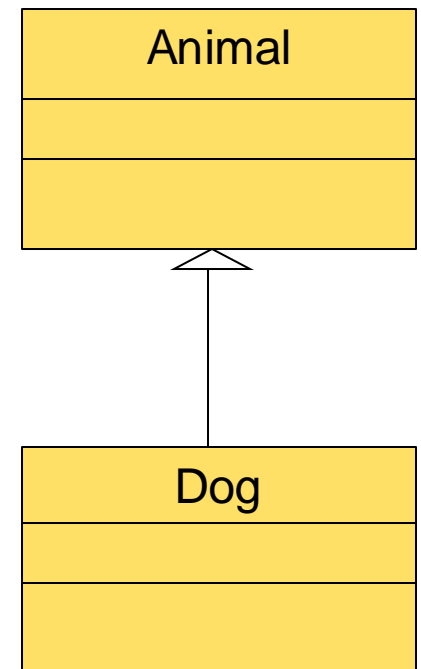


Inheritance

Base  Derivative



Association



Inheritance

Notation: relations

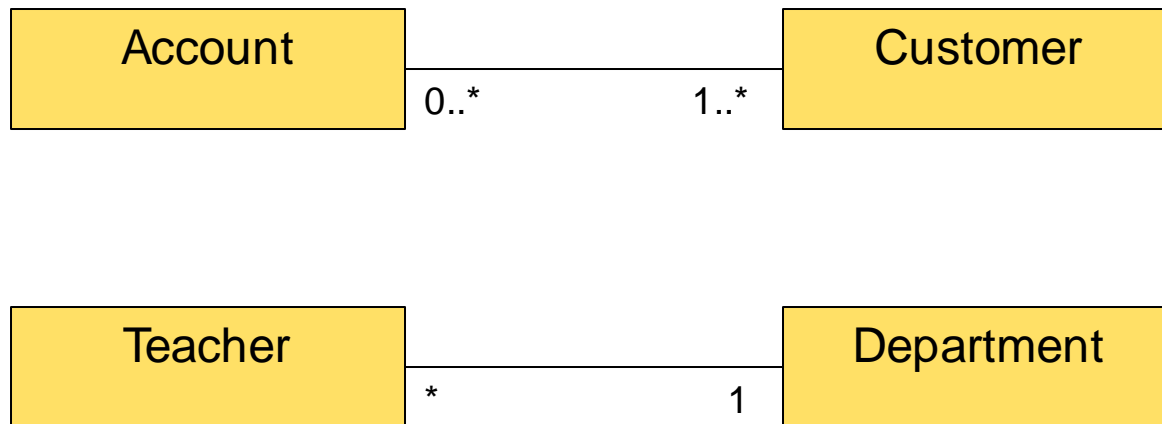
Multiplicity

- The number of instances of a given class related with ONE instance of the other class
- Each association has two multiplicity values (one at each end).
- To specify multiplicity you can use the following values.

| | |
|------|----------------------------|
| 1 | One and just one |
| 0..1 | One or none. |
| 0..* | Zero or more. |
| 1..* | One or more (at least one) |
| n..m | From n to m |
| * | Zero or more |
| + | One or more (at least one) |

Notation: relations

Example

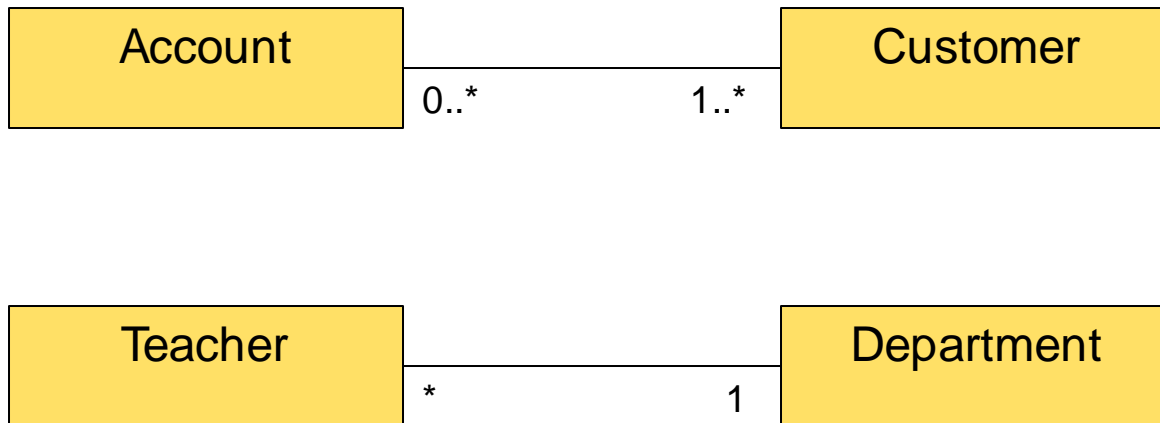


Notation: relations

Navigability in associations

Bidirectional association

The association can be navigated in both directions

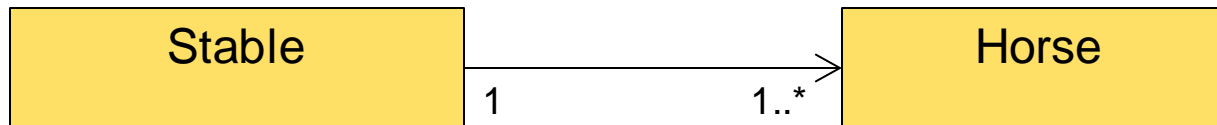


By default, **associations** allow **bidirectional navigability**

Notation: relations

Unidirectional association

Navigability is valid in just one direction.



To show the only meaningful direction we use an arrow

In a given context, it makes sense to reach the horses from a given stable but it is not needed to find the stable for a given horse.

Exercise 1(modeling)

- Three teachers share IP. A course is defined by its name and the year it belongs to. Teachers are defined by their name and the kind of classes they give (lab, theory or seminar).
 - Draw a class diagram.
 - Suppose a Course object and three Teacher objects are created. Draw an object diagram for that situation.
- In which situation could a class diagram change? How does it change?
- In which situation could an object diagram change? How does it change?

Exercise 2 (object interaction)

Assume the following sentence is successfully executed:

```
Printer p1 = new Printer();
```

with the `Printer` class having the following methods:

```
public void print(String fileName,  
                  boolean doubleSide)  
public int checkStatus(int wait)
```

Write down two possible calls to those methods.

Exercise 3 (object interaction)

Given the following class

```
public class Screen {  
    public Screen(int resX, int resY)  
    {...}  
    public int numberOfPixels()  
    {...}  
    public void clear()  
    {...}  
}
```

Write the code needed to create a `Screen` object and then clear the screen only if the number of pixels is greater than 1000.

Exercise 4 (object interaction)

```
public class Person {  
    private String name;  
  
    public Person(String name)  
    {...}  
    public String getName()  
    {...}
```

```
public class Vehicle {  
    Person driver;  
  
    public Vehicle()  
    {...}  
    public void print()  
    {...}
```

1. Draw the objects created by the execution of the Vehicle constructor.
2. Write the necessary code inside the print() method of the Vehicle class to print on the screen the first two characters of the name of the driver in upper case.

Exercise 5 (object interaction)

| Picture |
|---|
| <ul style="list-style-type: none">- wall: Square- roof: Triangle |
| <ul style="list-style-type: none">+ draw(): void |

| Square |
|---|
| <ul style="list-style-type: none">- color : String- isVisible: boolean- size: int |
| <ul style="list-style-type: none">+makeVisible():void+changeSize(s:int):void+moveVertical(v:int):void |

| Triangle |
|--|
| <ul style="list-style-type: none">- color : String- isVisible: boolean- height: int- width: int |
| <ul style="list-style-type: none">+makeVisible():void+changeSize(h:int, w:int):void+moveVertical(v:int):void+moveHorizontal(h:int):void |

Exercise 5 (object interaction)

```
public void draw() {  
    wall = new Square();  
    wall.changeSize(100);  
    wall.moveVertical(80);  
    wall.makeVisible();  
  
    roof = new Triangle();  
    roof.changeSize(50, 140);  
    roof.moveHorizontal(60);  
    roof.moveVertical(70);  
    roof.makeVisible();  
    ...  
}
```


Exercise 5 (object interaction)

What is the result of the execution of the following code?

Draw the created objects.

```
Picture demo = new Picture();
```

```
demo.draw();
```

Using the debugger

- A **debugger** is a program that allows programmers to execute an application in a controlled way.
- It provides functionality such as:
 - Stop and resume a program execution in previously selected source code points.
 - Execute the code sentence by sentence.
 - Track the variable values.

Object interaction: testing and debugging

- Unit test example with object interaction and debugger use.

The screenshot displays an IDE window titled "Control - interaccion objetos" containing the following Java code:

```
Control X
[Compilar] [Deshacer] [Cortar] [Copiar] [Pegar] [Buscar...] [Cerrar]

    tv = new TV();
}

public TV getTv()
{
    return tv;
}

public void changeCH (char car)
{
    if (car == '+' || car == '-')
        tv.changeChannel(car);
}
}
```

The debugger window, titled "Blue: Depurador", is open on the right. It shows the following information:

- Opciones:** Hilos: main (en breakpoint)
- Secuencia de Llamado:** Control.changeCH
- Variables estáticas:** (Empty)
- Variables de Instancia:** private TV tv = <object reference>
- Variables locales:** char car = '+'

At the bottom of the IDE, a status bar indicates "Clase compilada - no hay errores de sintaxis". The debugger toolbar at the bottom right includes buttons for "Detener", "Paso", "Entrar en", "Continuar", and "Terminar".