Chapter 5: Classes and Objects in Depth

Information Hiding

Objectives

- Information hiding principle
- Modifiers and the visibility
- UML representation of a class
- Methods
- Message passing principle
- Passing parameters
- Getters and setters
- Constructors
- Overloading

Object Oriented Basic Principles

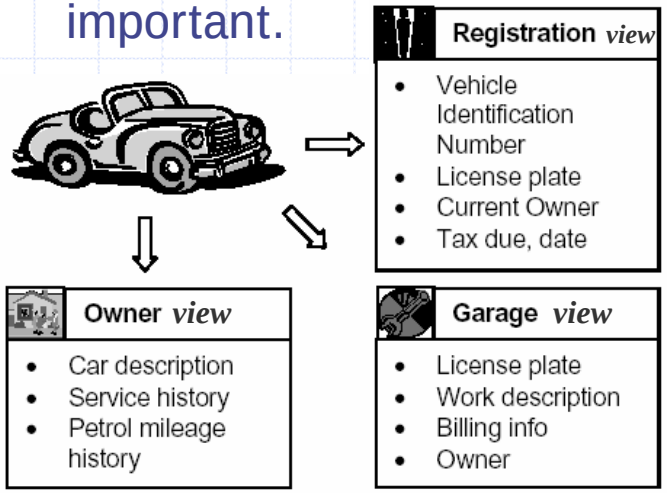
- Abstraction
- Encapsulation
- Information Hiding
- Message Passing
- Overloading

- Inheritance
- Overriding
- Polymorphism
- Dynamic Binding

- Information hiding and Message passing are discussed in this chapter.
- Overloading is discussed in chapter 6.
- Inheritance, Polymorphism, Overriding and Dynamic binding are discussed in CSC 113.

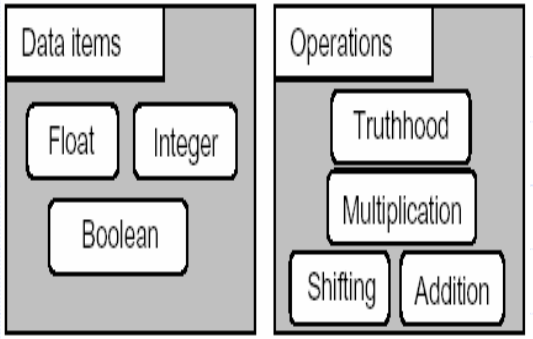
Abstraction Principle

- Data Abstraction
  - In order to process something from the real world we have to extract the essential characteristics of that object.
  - Data abstraction is the process of:
    - Refining away the unimportant details of an object,
    - Keeping only the useful characteristics that define the object.
  - For example, depending on how a car is viewed (e.g. in terms of something to be registered, or alternatively something to be repaired, etc.) different sets of characteristics will emerge as being important.
- Functionality Abstraction
  - Modeling functionality suffers from
    - unnecessary functionality may be extracted,
    - or alternatively, an important piece of functionality may be omitted.
  - Functionality abstraction is the process of determining which functionality is important.

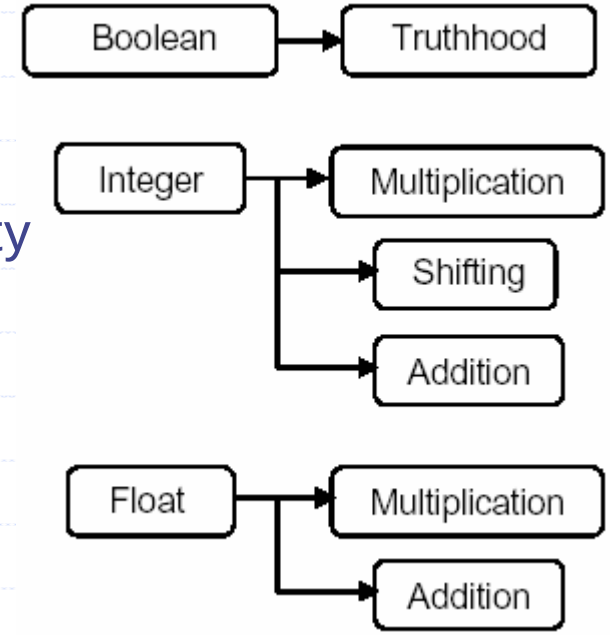


Encapsulation Principle

- Abstraction involves reducing a real world entity to its abstraction essential defining characteristics.



- Encapsulation extends this idea by also modeling and *linking* each data of an entity to the appropriate functionality of that entity.



Encapsulation Gives Classes

- OOP makes use of encapsulation to ensure that data is used in an appropriate manner.
  - by preventing from accessing data in a non-intended manner (e.g. asking if an Integer is true or false, etc.).
- Through encapsulation, only a predetermined appropriate group of operations may be applied (have access) to the data.
- Place data and the operations that act on that data in the same class.
- Encapsulation is the OO principle that allows objects containing the appropriate operations that could be applied on the data they store.
  - My Nokia-N71 cell-phone stores:
    - My contacts,
    - Missed calls
    - ... etc.
  - My Nokia-N71 may perform the following operations on the data it contains:
    - Edit/Update/Delete an existing contact
    - Add a new contact
    - Display my missed calls.
    - ...etc.

Information Hiding Principle

- Limit access to data only to internal operations that need it.
- OO classes hide the data as private data members and use public accessor operations to get at it.
  - The scope of the data is limited to the class.

Information Hiding Objectives

- Information hiding protects from exposing:
  - data items (attributes).
  - the difference between stored data and derived data.
  - the internal structure of a class.
  - implementation details of a class.

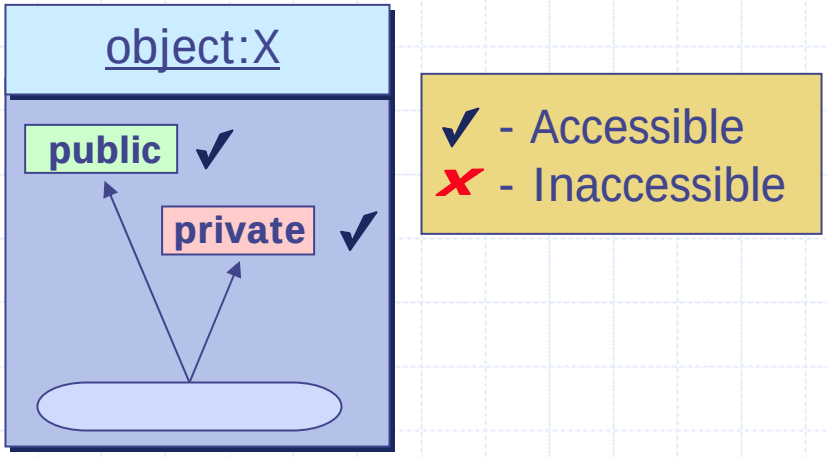
Encapsulation and Information Hiding

- Encapsulation (is a language construct that) facilitates the bundling of data with the operations acting on that data.
  - **Place data and the operations that perform on that data in the same class**
- Information hiding is a design principle that strives to shield client classes from the internal workings of a class.
- Encapsulation facilitates, but does not guarantee, information hiding.
- Smearing the two into one concept prevents a clear understanding of either.

public and *private* modifiers

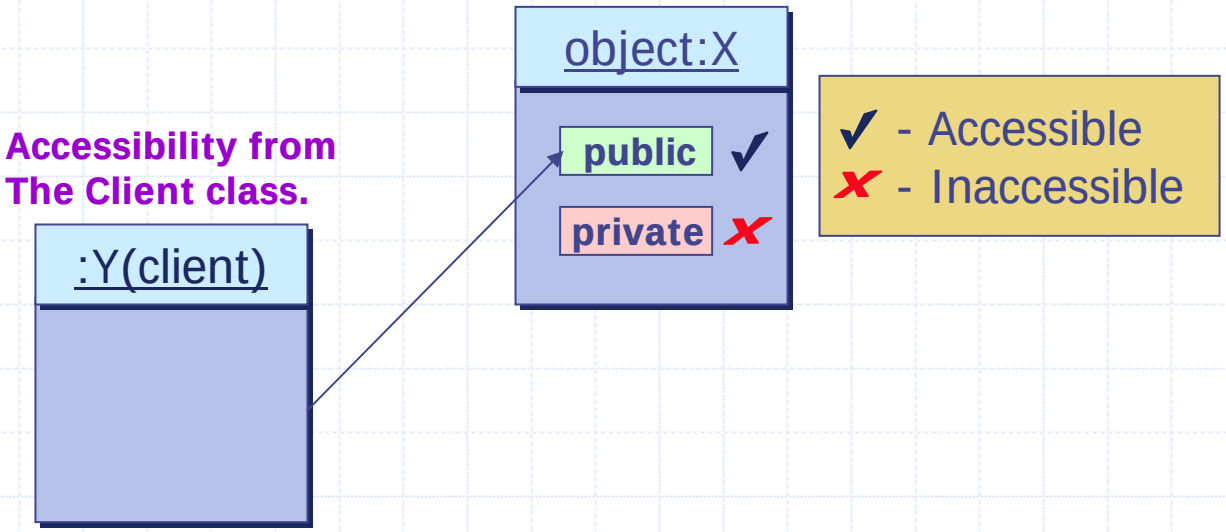
- Let's consider a class X.
- Let's consider Y a client class of X.
  - Y is a class that uses X.
- Attributes (and methods) of X declared with the *public* modifier are accessible from instances of Y.
  - The public modifier does not guarantee the information hiding.
- Attributes (and methods) of X declared with the private modifier are not accessible from instances of Y.
  - The private modifier guarantee the information hiding.

Accessibility from Inside (the Instance itself)



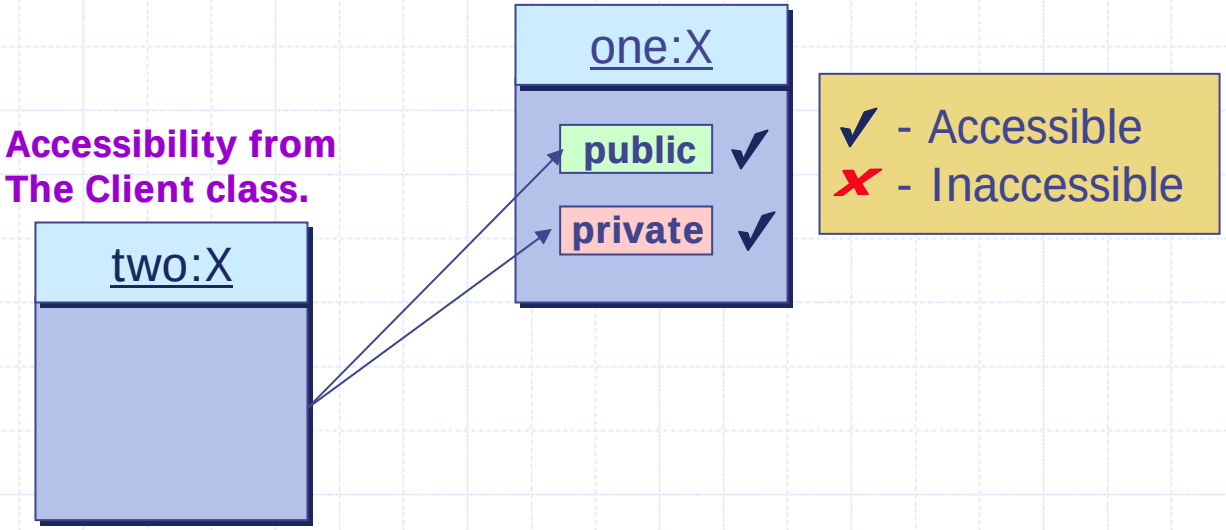
All members of an instance are accessible from the instance itself.

Accessibility from an Instance of another Class



Only public members
Are visible from outside.
All else is hidden from
Outside.

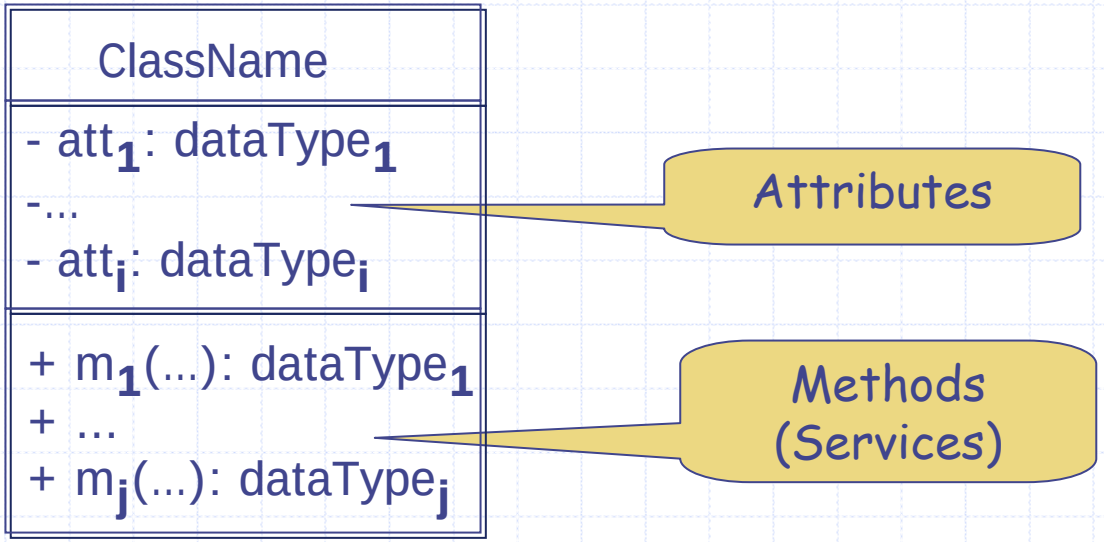
Accessibility from an Instance of the same Class



If a member is accessible from an instance, it is also accessible from other instances of the same class.

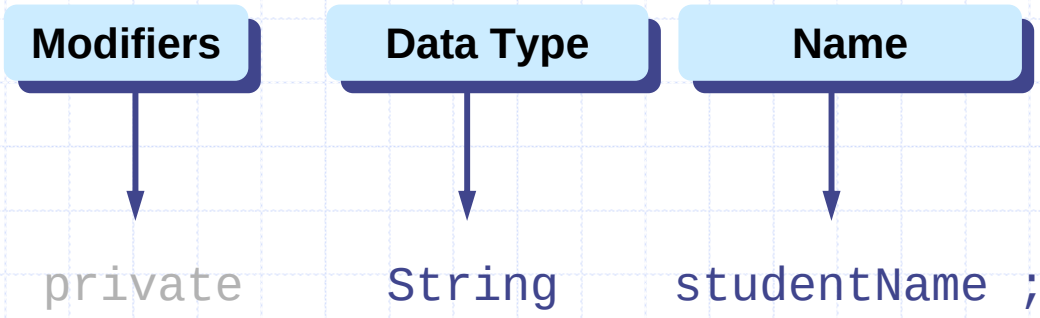
UML Representation of a Class (UML Class Diagram)

- UML uses three symbols to represent the visibility of the class' members.
  - **+** : mentions that the member is *public*.
  - **-** : mentions that the member is private.
  - # : introduced in the CSC 113.

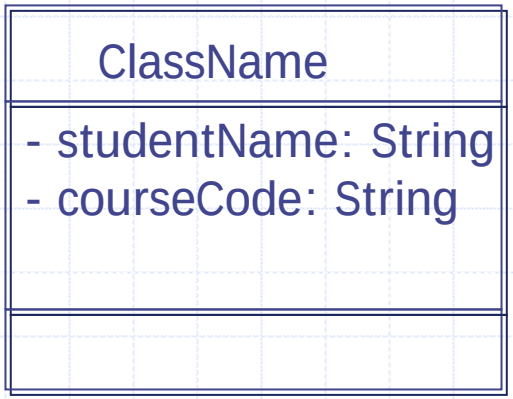


Declaring Private Attributes

```
<modifiers>  <data type> <attribute name> ;
```



Example of a Class with Private attributes




```
public class Course {

    // Attributes
    private String studentName;
    private String courseCode ;
    // No method Members

}
```

```
class Course {

    // Data Member
    private String studentName;
    private String courseCode ;

}
```

```
public class CourseRegistration {
    public static void main(String[] args) {
        Course course1, course2;
//Create and assign values to course1
        course1 = new Course( );
        course1.courseCode= “CSC112“;
        course1.studentName= “Majed AlKebir“;

//Create and assign values to course2
        course2 = new Course( );
        course2.courseCode= “CSC107“;
        course2.studentName= “Fahd AlAmri“;

        System.out.println(course1.studentName + " has the course “+
                                                course1.courseCode);
        System.out.println(course2.studentName + " has the course “+
                                                course2.courseCode);

    }
}
```

Accessibility Example

```
…
Service obj = new Service();

obj.memberOne = 10;

obj.memberTwo = 20;

obj.doOne();

obj.doTwo();
```

```
class Service {
    public  int memberOne;
    private int memberTwo;

    public void doOne() {

    …

    }
    private void doTwo() {

    …

    }
}
```

Client

Service