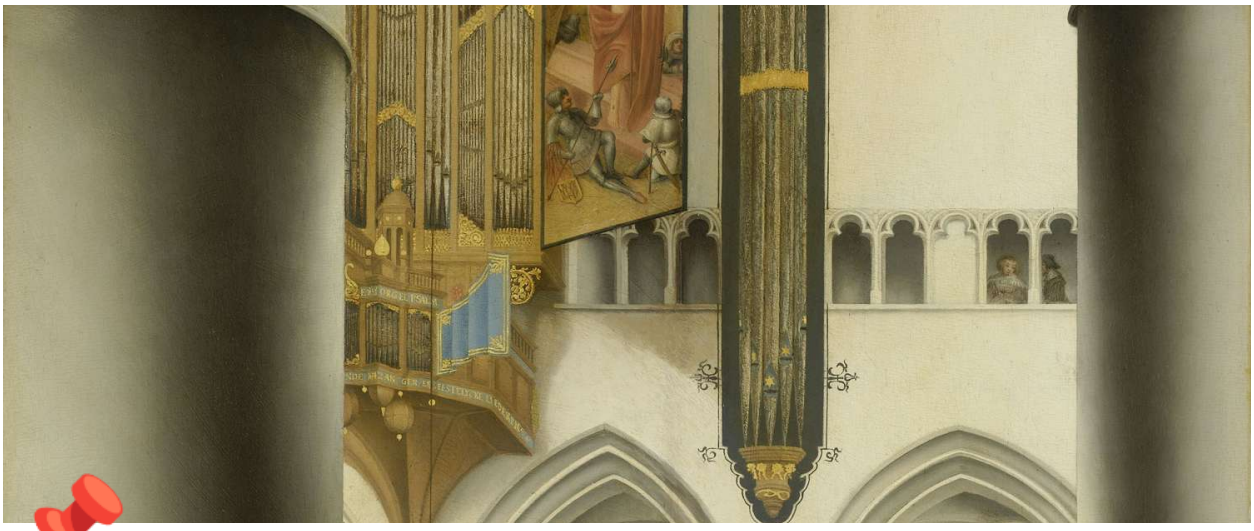


object oriented Design

💡 notes from Alberta university on Coursera

1st topic (video)

we can measure the performance of a design of software by asking a few questions ➖

1) did a small code change produce a ripple-effect for changes elsewhere in the code?

2) was the code hard to reuse ?

3) was that software difficult to maintain after a release ?

2nd topic (video)

the most important thing in design is to be simple in other way easy to read by other programmers without comments

there is a quote we could say here “if it’s simple you probably will get it right”

3rd topic (article)

- **abstract** - in object-oriented design, a class that cannot be instantiated directly, but must be subclassed. It can also apply to a method or attribute of a class.
- **abstract data types** - data types that are defined by the their behaviour as opposed to their structure. Defined by the developer rather than the programming language
- **access modifiers** - keywords that control which other classes can access a variable or method in a class. These include public, protected, private, a nd no keyword, sometimes called default
- **abstraction** - the act of simplifying a concept in context. In object-oriented design, it is the simplification of the real world entity into its most important attributes and behaviours for the purpose of the software.
- **attribute** - a property that object of a class must have, even though their values may be different. For example, two student objects each have a grade attribute, even though one has an 'A' and the other has a 'B.'
- **behaviours** - the actions that an object can take
- **boundary object** - an object whose role is to interface with an external component, such as a user or an adjacent system
- **class diagram** - a UML diagram for showing the behaviours, attributes, inheritance, and connections of classes
- **code review** - systematic reviews of written code done by the development team. Not only to find mistakes, but to get developers using the same conventions, constructs, design principles, etc.
- **cohesion** - describing the complexity within a module, e.g. a class or a method. high cohesion describes a module that has a clear purpose and is no more complex than it needs to be. low
  -cohesion describes a module which has an unclear purpose or which is overly complex.
- **component** - a discrete part that has a particular role or function, called a responsibility. From a design perspective, a component will eventually be turned into an object, function, or group of subcomponents

- **conceptual integrity** - the consistency of software. Software with conceptual integrity will seem like it is programmed by one developer, even if it was programmed by a team
- **concern** - a general term, referring to some action or role that is part of the solution to the problem.
- **control object** - an object whose role is to manage other objects or control their interactions
- **counterexamples** - during model checking, counterexamples are instances wherein the system did not behave as expected
- **coupling** - describing of the complexity of connections between modules. tightly coupled modules are highly dependent on each other and difficult to reuse in other contexts. loosely coupled modules are less dependent and easier to reuse.
- **CRC** - stands for class responsibility collaborator: a technique for summarizing and mapping
  objects in an object-oriented design
- **deadlock** - a situation in which the system can never continue because subprocesses need
  other subprocesses to act before they can continue
- decomposition - breaking an entity into parts that can be implemented separately
- **degree** - when talking about coupling, degree is the number of connections between the two
  modules of interest. This is one dimension of how coupled these modules are.
- **design** - the process of planning a software solution, taking the requirements and constraints
  into account. Divided into higher-level conceptual design and more specific technical design.
- **design patterns** - established solutions to common coding problems. Characterized by their
  general form and function rather than by specific code
- **ease** - when talking about coupling, ease is how obvious connections are between modules.

This is one dimension of how coupled these modules are.

- **encapsulation** - bundling attributes and behaviours into an object, exposing features of that
  object to other objects as necessary, and restricting the remaining features
- **entity** - the role or behaviour that is being represented by a software object or process
- **flexibility** - when talking about coupling, flexibility is how easily a module can be swapped for a
  different module. This is one dimension of coupling.
- **getter** - a method for getting the value of a class variable which is not directly accessible
- **generalization** - factoring out common features of classes or functions that can be reused in
  other places. Allows for more code reuse
- **global variable** - a variable that is accessible by any subroutine or subcomponent
- **flexibility** - the ability of a design to adapt to changes or be adapted to different purposes
- **implementation** - the process of creating a working program from the design
- **information hiding** - designing classes so that the information that other classes do not need is
  hidden away from them.
- **inheritance** - attributes or behaviours that subclasses inherit from a superclass or implement
  through an interface
- **instantiate** - to create an object of a class
- **interface inheritance** - a method of inheritance wherein if a class implements an interface, it
  must define all the methods specified in the interface
- **Liskov substitution principl**e - a principle stating that a superclass should be able to be

replaced by its subclass without changing behaviour significantly

- **local variable** - a variable accessible only to one class or subroutine
- **maintenance** - modifying software after delivery to fix, improve, or change features
- **maintainable** - the ability of code to be changed
- **model** - the abstract representation of the key concepts and relationships that make up a
  software solution
- **model checking** - a systematic check of all of the system's states. Consists of several steps:
  modeling phase, running phase, and analysis phase
- **module** - general term to refer to a programming unit, like a class or a method.
- **namespace** - an abstract container for a group of related modules, given a unique identifier
- **object-oriented modelling** - modelling a software solution using concepts from object-oriented languages such as Java. Characterized by representing key concepts with software objects.
- **override** - a subclass may have a method that is already in the superclass, in which case the
  subclass' method will be used instead.
- **package** - a means of organizing related classes into the same namespace
- **polymorphism** - the ability to interact with objects of different types in the same way. Usually
  achieved through inheritance or through interfaces in Java
- **programming paradigm** - the style or way in which programming achieves its objectives, which
  can vary by language and toolset
- **quality attributes** - properties of a software system that indicate its quality
- **requirements** - the requirements that the software will be designed to fulfill. These could be
  functional requirements such as providing some result, or business requirements

such as being
user-friendly, or meeting budgetary restrictions

- **responsibility** - the purpose or function of a component of the software
- **reusable** - the ability of code to be reused in different contexts
- **rule of least astonishment** - a design principle dictating that a component should behave as
one would expect it to behave
- **separation of concerns** - a principle dictating that different concerns should be in different
modules
- **service-oriented architecture** - a type of architecture characterized by providing services to
external clients. The clients do not know how these services are provided.
- **sequence diagram** - a UML diagram that shows the sequence of actions that form one process
- **setter** - a method for setting the value of a class variable which is not directly accessible. Allows
for gatekeeping e.g. restricting the values to which the variable can be set
- **software architecture** - the higher-level structure of a software system; how various
components are arranged into a coherent and functional whole
- **state diagram** - a UML diagram that shows the different states of a system
- **tradeoff** - a decision between alternatives that each provide benefits and downsides
- **Unified Modelling Language** - a visual design language encompassing many different
diagrams that depict software in different ways
- **verification** - confirming that the software solution meets the requirements

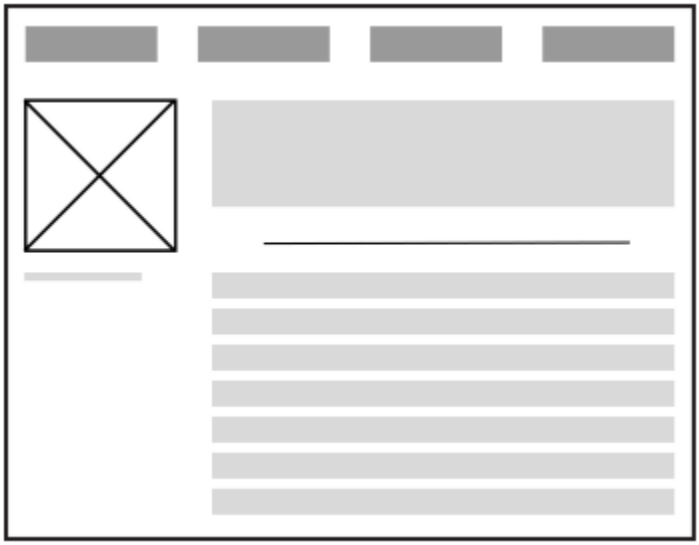
Topic 4 (video)

- **Object oriented moduling** :- is a representation of a entity as an object

Topic 5 (video)

There are types of modul the software like :-

- **conceptual Design -** are created with an initial set of requirements as a basis
- This is what it means of conceptual mock-ups



- **Technical Design -** Technical designs build on conceptual designs and requirements to define the technical details of the solution

Topic 6 (article)

- Categories of objects -

1. **Entity objects - a**re the most familier ,because they correspond to some real-world entity in the problem space (for exmaple if you have an object representing a building or a customer then that entity is an object by themselves) ,they will also be able to modify themselves .
2. **Boundary objects** - are objects which sit at the boundary between systems. This could be an object that deals with another software system

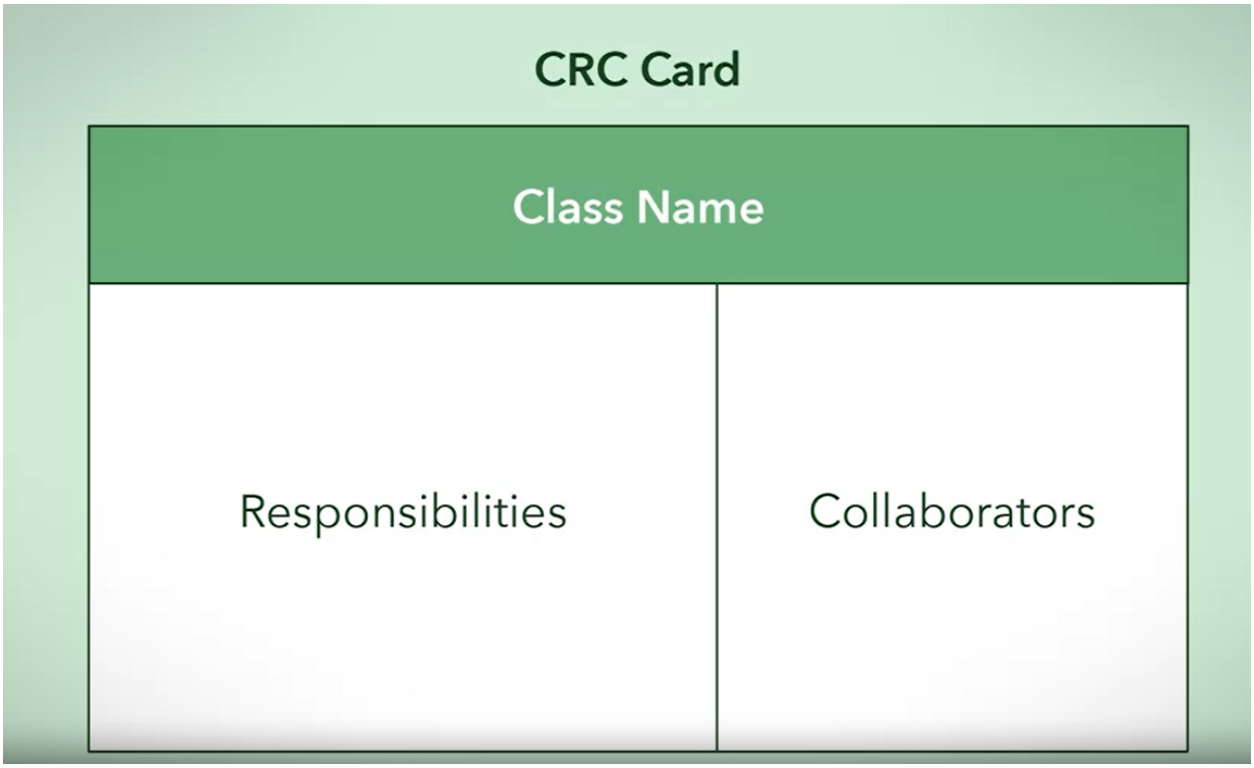
3. **Control objects** -are objects which are responsible for coordination. You will discover control objects when you attempt to break down a large object, and find that it would be useful to have an object that controls the other objects.

Topic 7 (video)

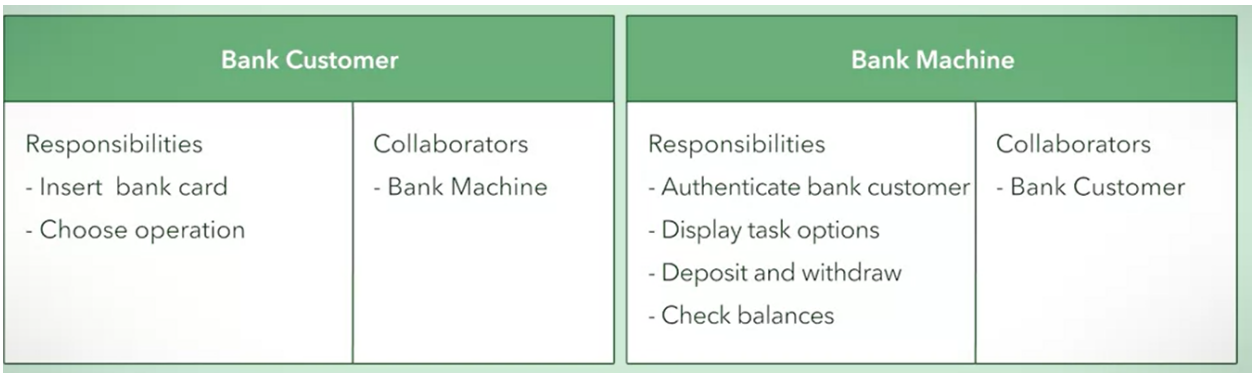
the summary of that topic that we have to make a balance between (performance ,secuirty,convenience) you can search about it for more details

Topic 8 (video)

- **CRC** (class-Reponsibilty-Collaborator)
- CRC cards used to : - record ,organize ,Refine Design



- as it’s shown in this figure when we will put the components name at class name,the responsibilty in responsibility section and finally the connections in collaborators section



when we put the CRC cards together we will get an image of how that system suppose to work

Topic 9 (video)

- The four design principles -

1. Abstraction
2. Encapsulation
3. Decomposition
4. Generalization

- **Abstraction** - is the idea of simplifying a concept in the problem and allows you to better understand a concept by breaking it down into a simplified description that ignores unimportant details
- from the top view of abstraction it means that we should define (attributes / behaviour) for that entity
- **Encapsulation** - encapsulation forms a self-contained object by bundling the data and functions it requires to work (means it makes sure that attributes and behaviours work together in the class)
- **Decomposition** - is taking a whole thing and dividing it up into different parts
- **Generalization** - it means to take the common between two or more classes or methods and make them general through something like - inheritance
- Some Rules for Inheritance -

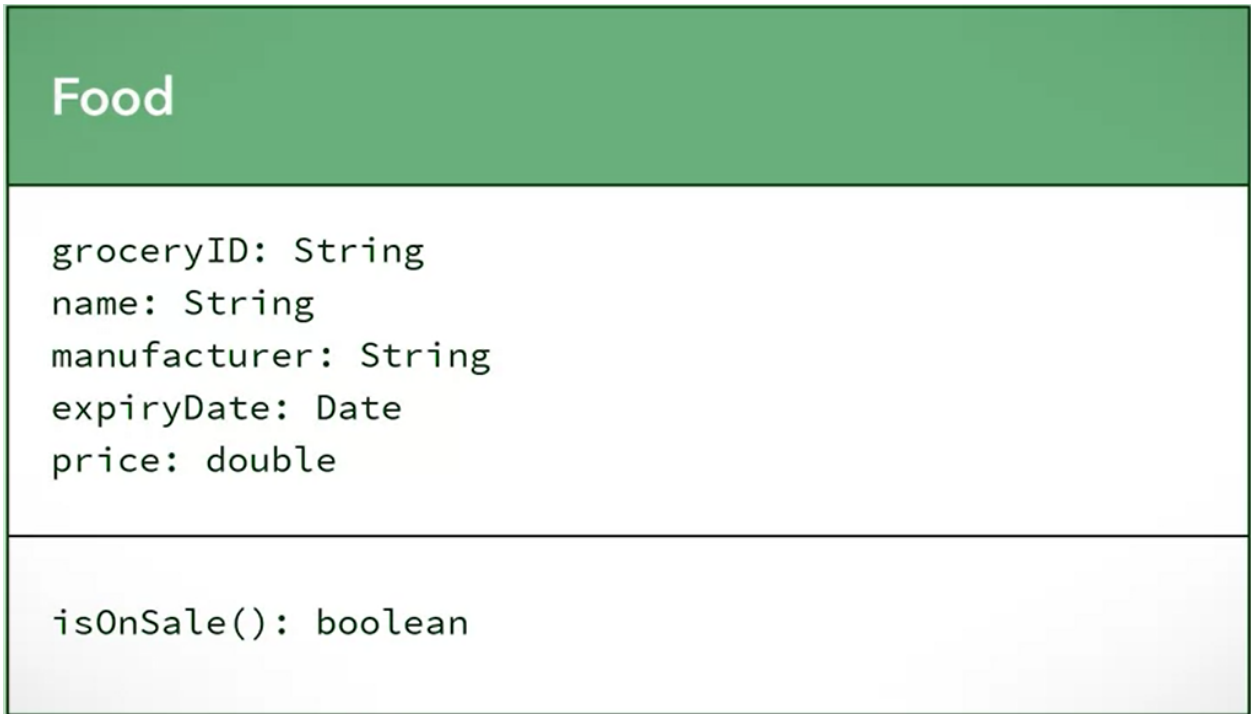
1.The parent class is called also (super class) meanwhile the child class is called (sub classes)

- Inheritance support a quote that says (D.R.Y) Don’t repeat your self .

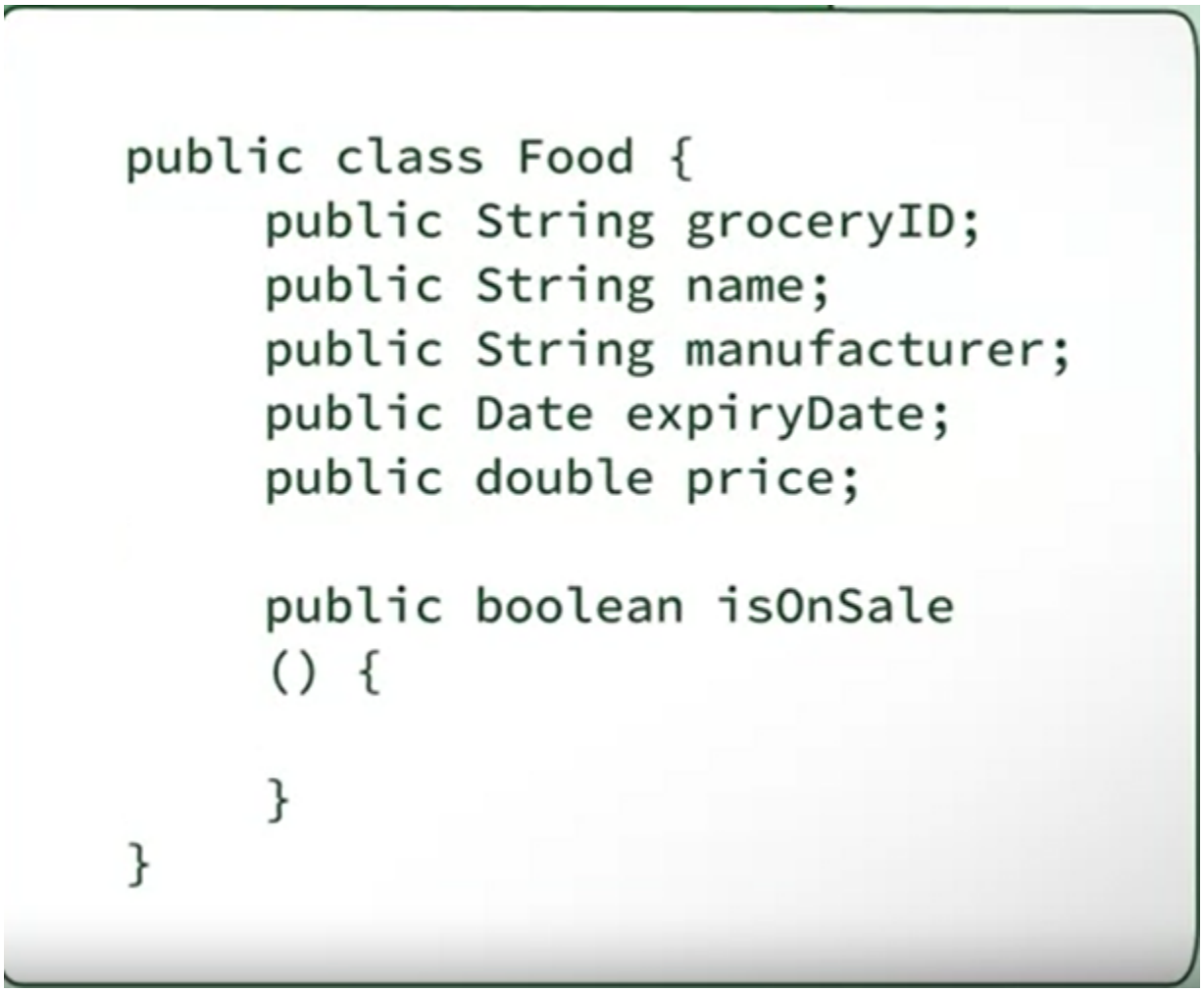
Topic 10 (video)

- **Abstraction in UML -**
- UML diagram allows you to represent the code better than CRC cards but also it is just visualisation

- this is how we use UML diagram to represent a class at the first section there are the attributes and in the last section there are the behaviour or methods and ofcourse at the top is the name of class

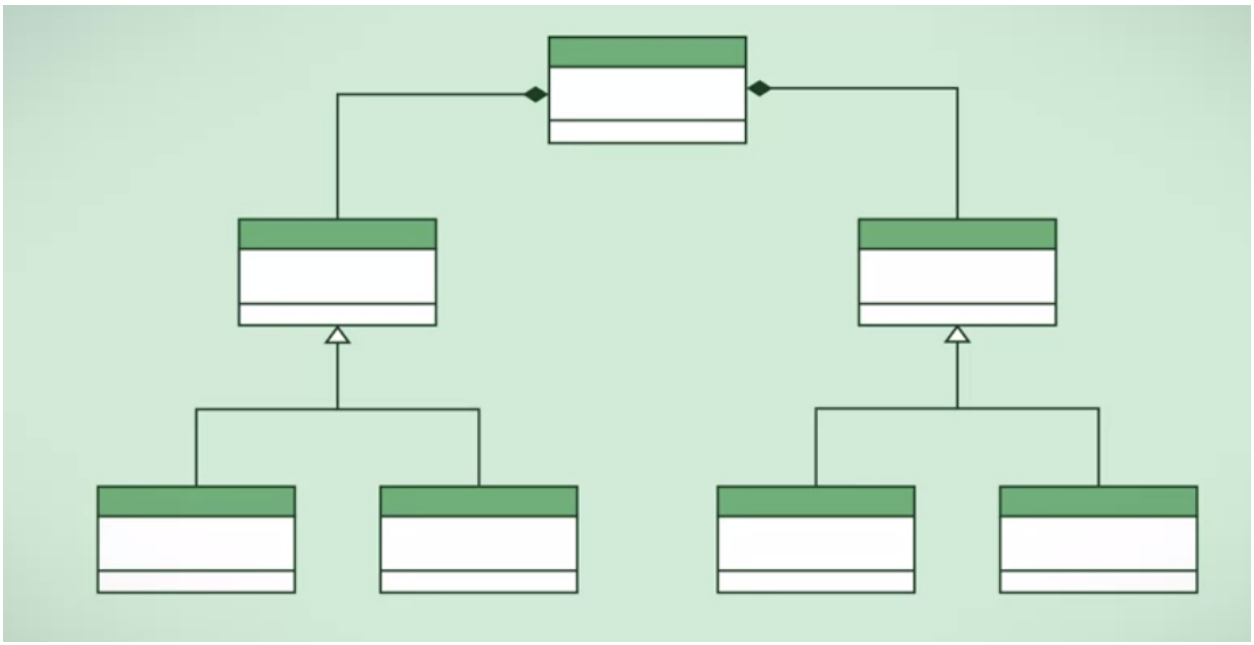


- the code for this UML in Java -

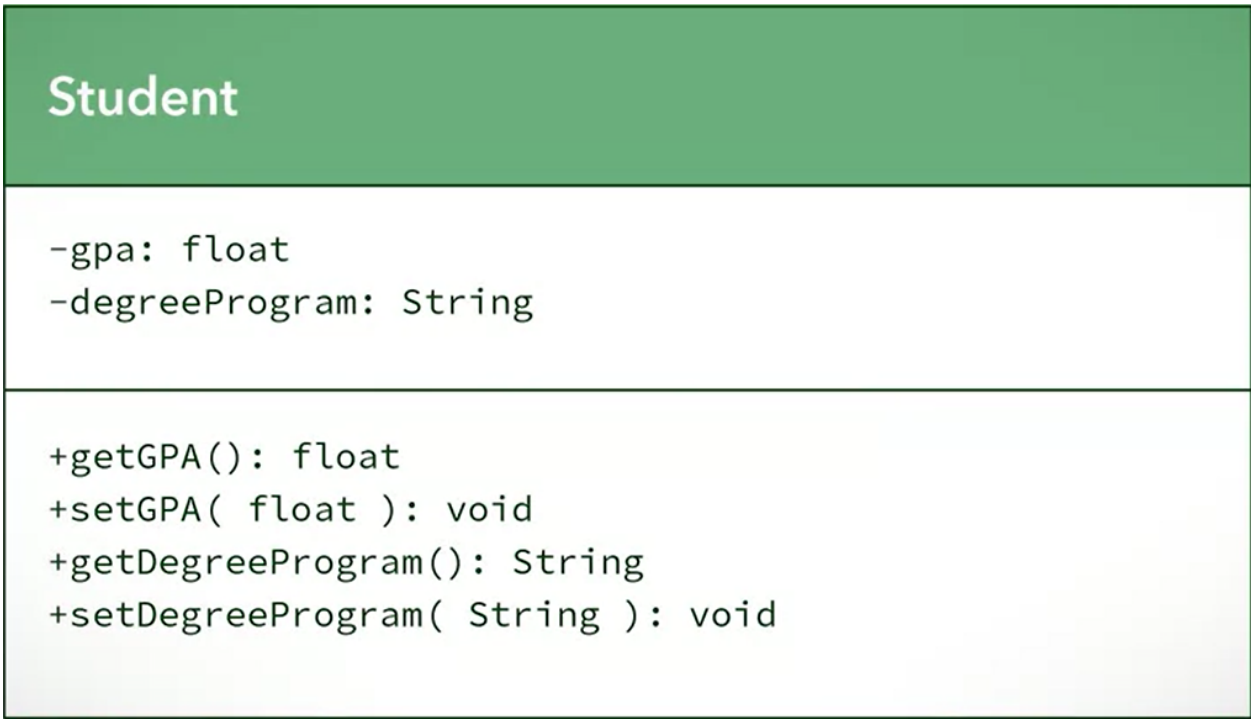


```
public class Food {
    public String groceryID;
    public String name;
    public String manufacturer;
    public Date expiryDate;
    public double price;

    public boolean isOnSale
    () {

    }
}
```

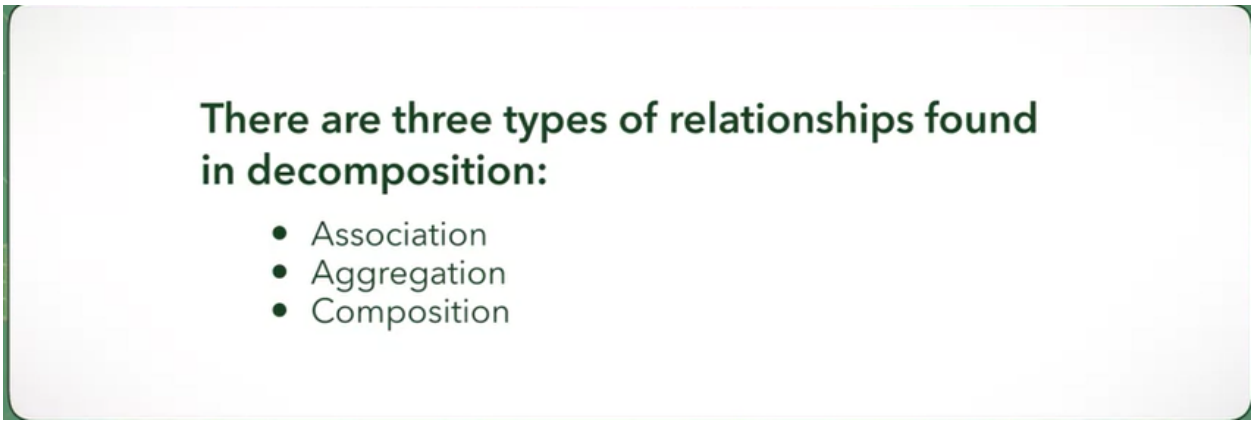
- when we put the UML classes together we get the final image of system



- **Encapsulation in UML -**



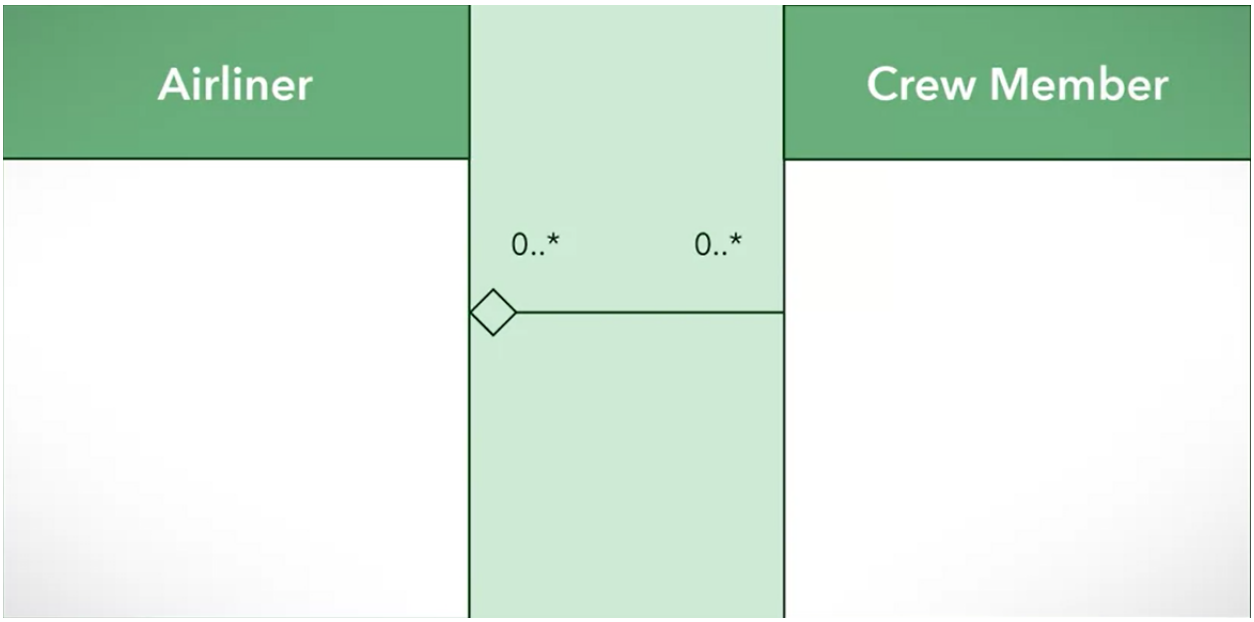
- for this UML diagram we would notice that the (-) means that attribute or behaviour has an access modifer with a value “private” and the (+) means it’s “public”
- **Decomposition in UML -**



- **the association** is “some” relationship this neabns that there is a loose relationship between 2 objects ,these objects may interact together some times
- **Aggregation** - is a “has-a” relationship where a whole has parts that belong to it there may be sharing of parts among the whole in this relationship

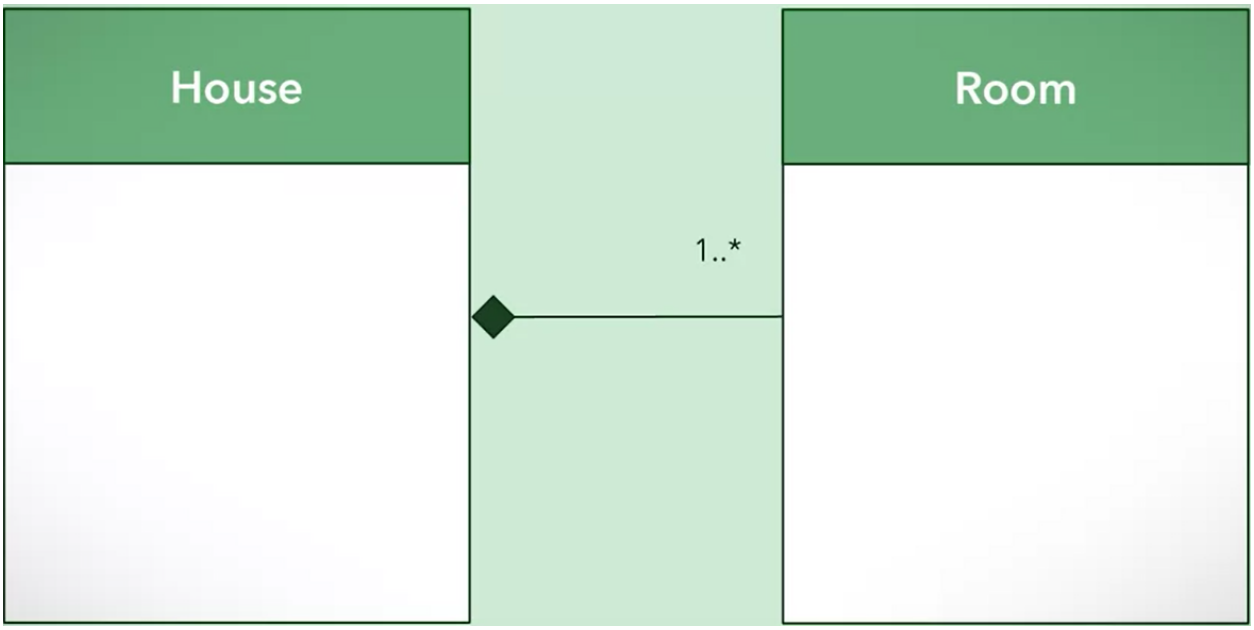
-the “has-a” relationship from a whole to the parts is considered weak because they can also exist independetly

- the retation of a “has-a” represnted with this diamond

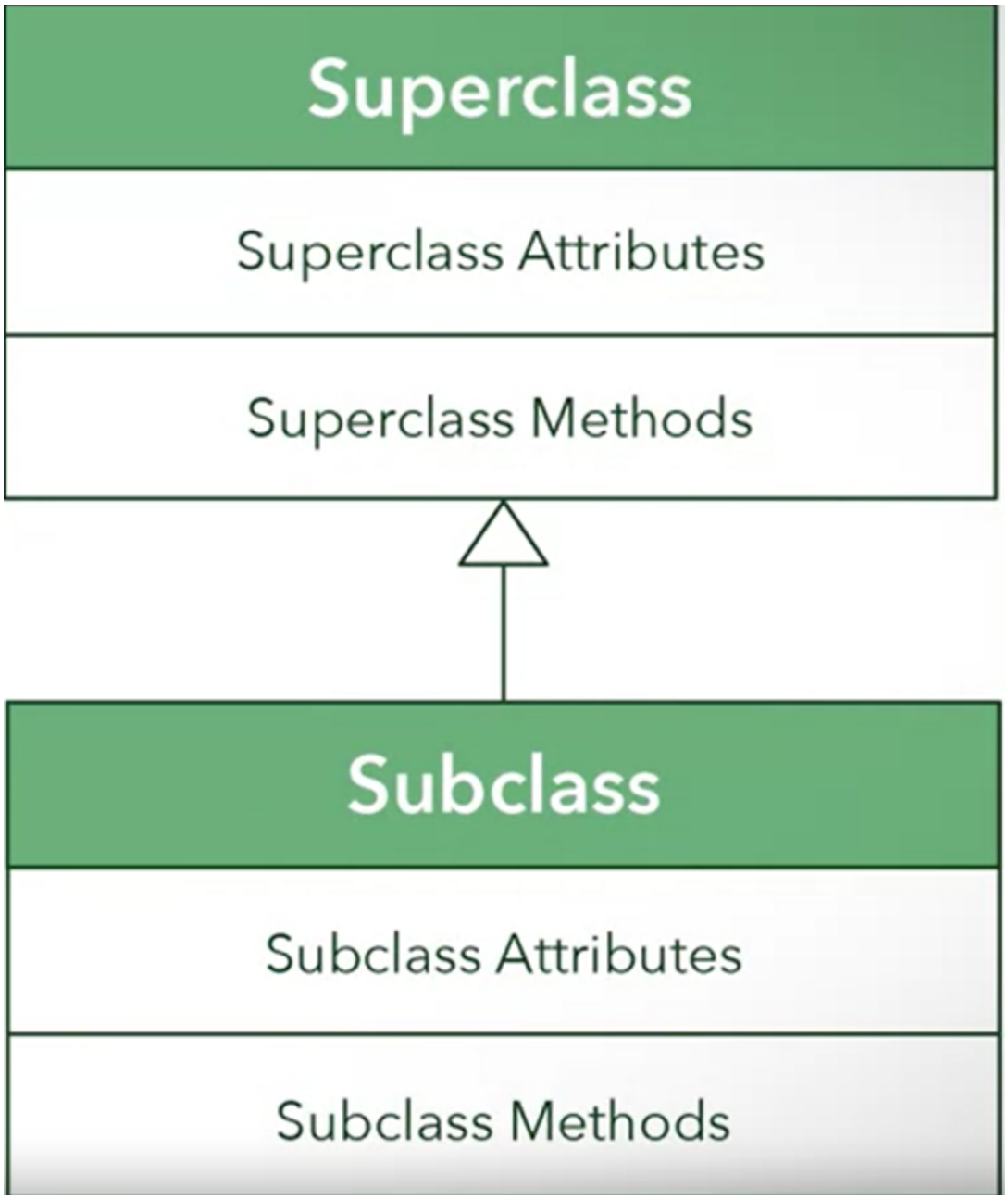


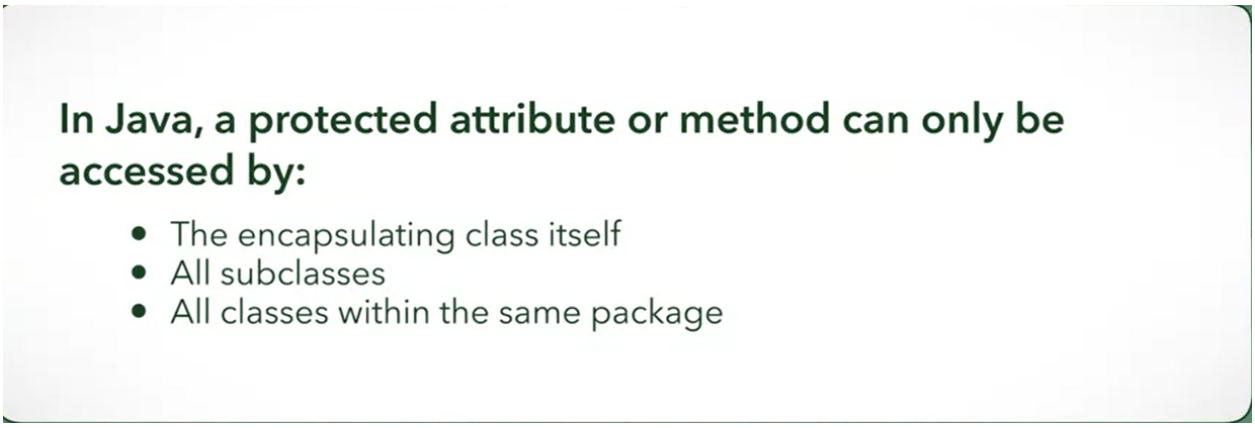
- **Composition**-is an exclusive containment of parts or we can say a strong has-a relationship ,this means is that the whole cannot exist without its parts

- the realtion of composition -

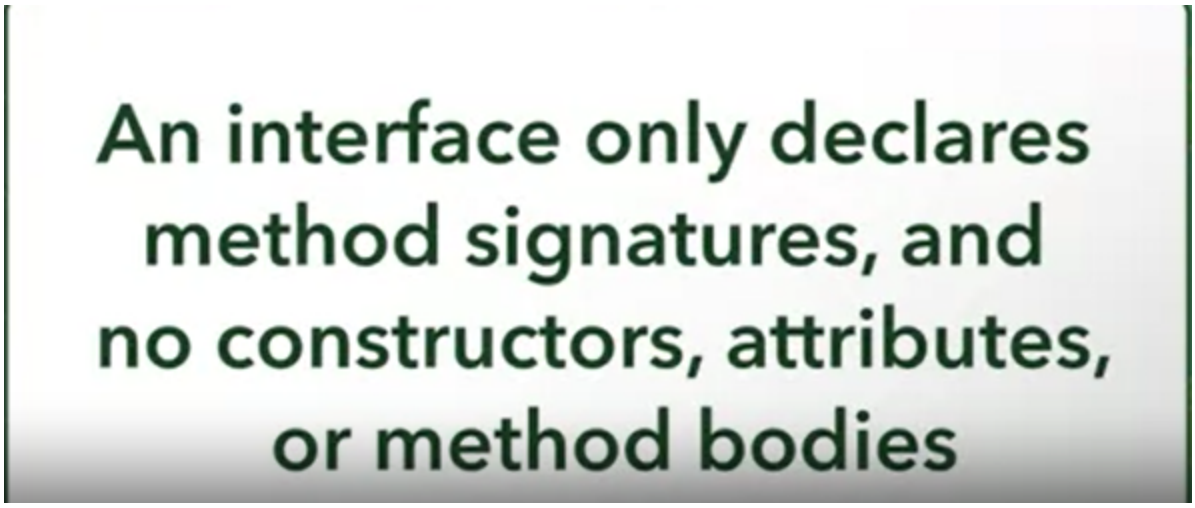


- **Generalization -(Inheritance)**

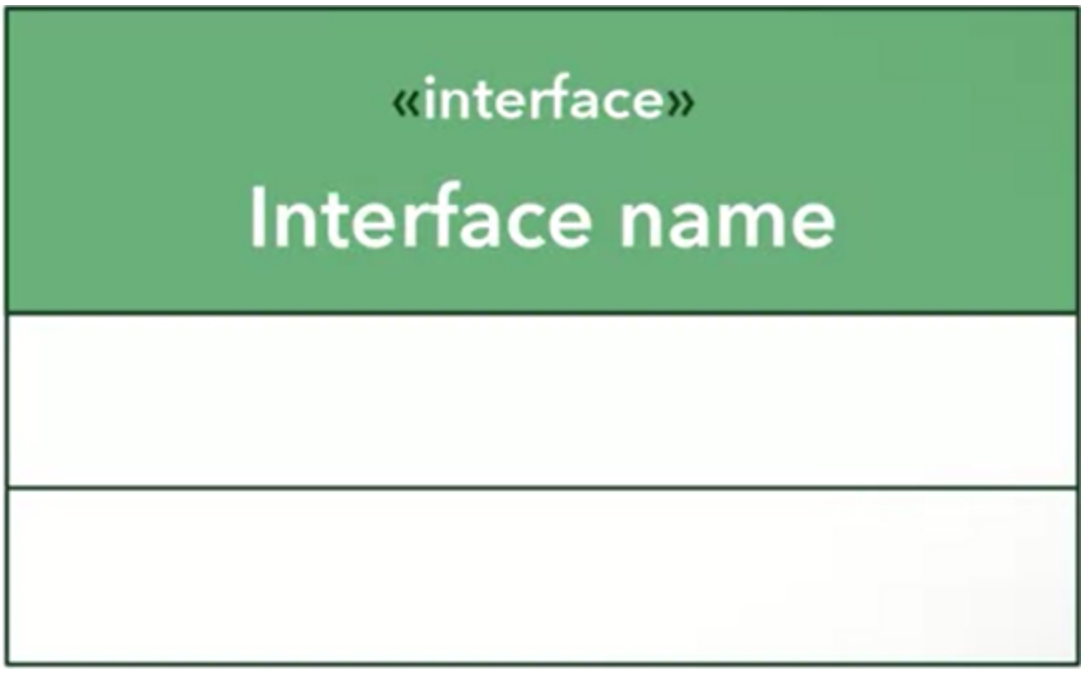

Superclass
Superclass Attributes
Superclass Methods
Subclass
Subclass Attributes
Subclass Methods



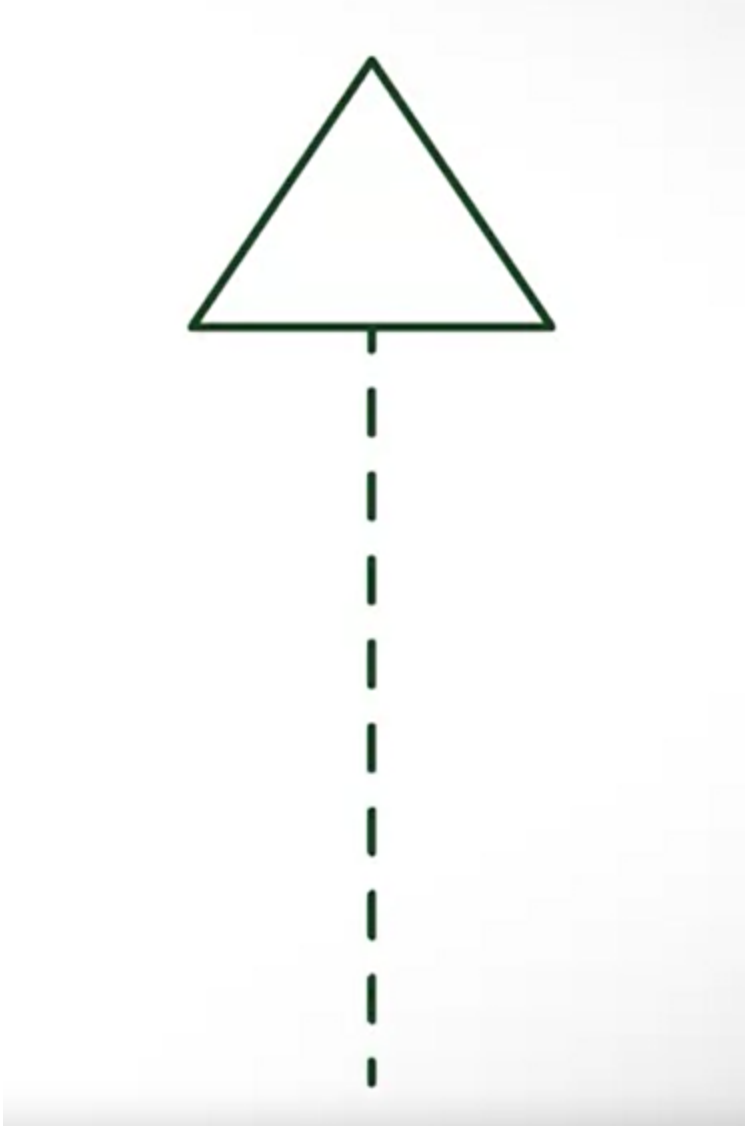
- **interface -**



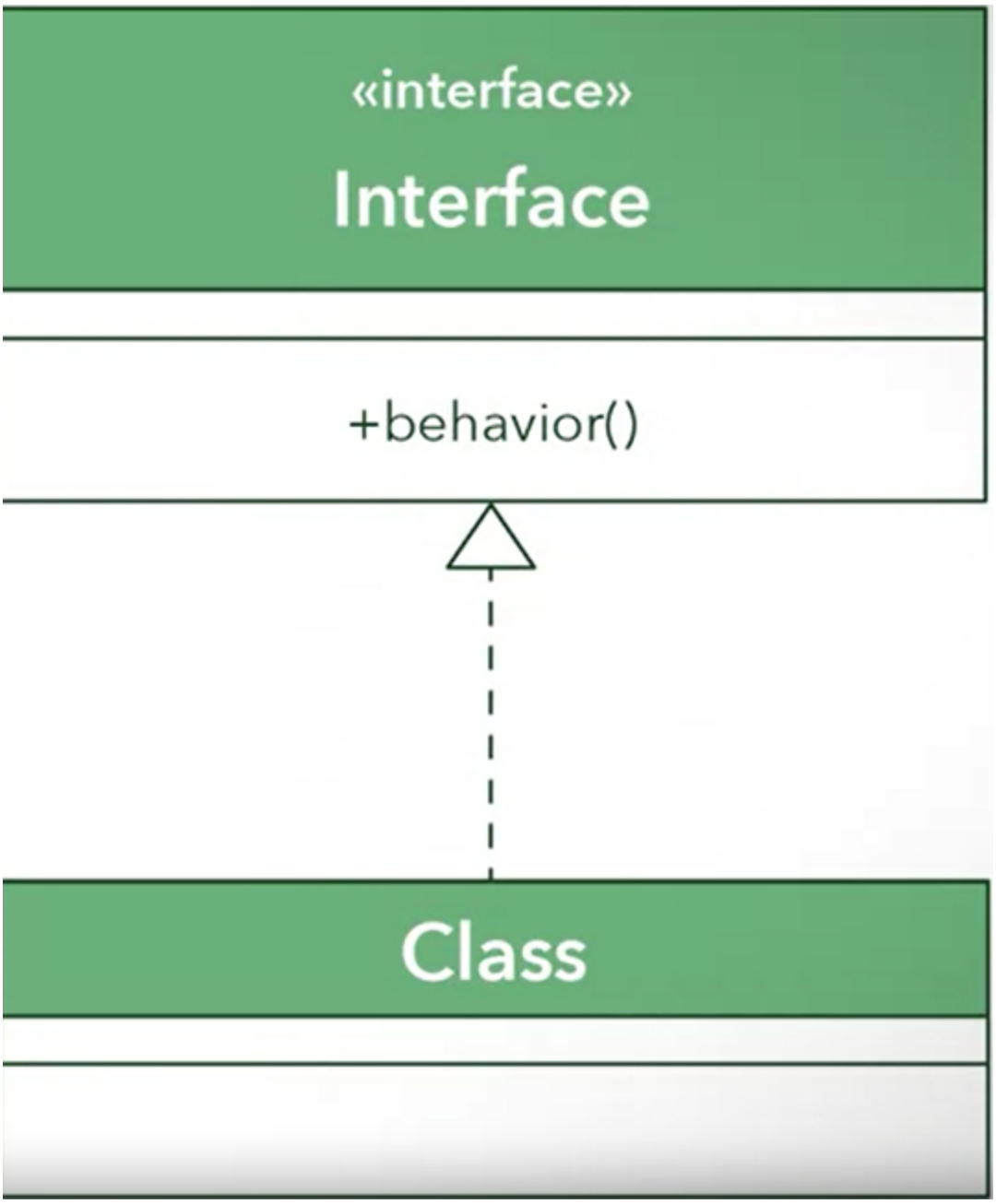
- TO represent interface in UML diagram we use this shape -



- and to represnet the relatoion between interface and classes we use -



- so we will get the final result -



Topic 11 (video)

- TO evaluate the Design compexity you will use : -

1**.Coupling** - is the complexity between module and another module

2**.Cohesion** -focuses on complexity within a module, and represents the clarity of the responsibilities of a module

- **Coupling -**focus on complexity between a module and other modules. Coupling can be balanced between two extremes: tight coupling and loose coupling. If a module is too reliant on other modules, then it is “tightly coupled” to others. This is a bad design. However, if a module finds it easy to connect to other modules through well-defined interfaces, it is “loosely coupled” to others. This is good design.

-In order to evaluate the coupling of a module ,the metrics to consider are➖

- **Degree-**is the number of connections between the module and others. The degree should be small for coupling(the good design should have only few parameters)
- **Ease-**is how obvious are the connections between the module and others.(it means if you impleneted a system ,if other person come to check this system he must know every thing in system without any comments through clean code and other things)
- **Flexibility-**indicates how interchangeable the other modules are for this module. Other modules should be easily replaceable for something better in the future, for coupling purposes.
- **Cohesion-** focuses on complexity within a module, and represents the clarity of the responsibilities of a module. Like complexity, cohesion can work between two extremes: high cohesion and low cohesion.
- **High Cohesion -**if the design has a clear one task (the meaning of design or module here is like method) in other word the good design is a high coshesion
- **Low Cohesion-**if the design has more then one task and the purpose isn’t clear that means it is a low cohesion and so ,a bad design

Topic 12(video)

- **Seperation of concerns-**is a key idea that underlies object-oriented modelling and programming. When addressing concerns separately, more cohesive classes are created and the design

principles of abstraction, encapsulation, decomposition, and generalization are enforced

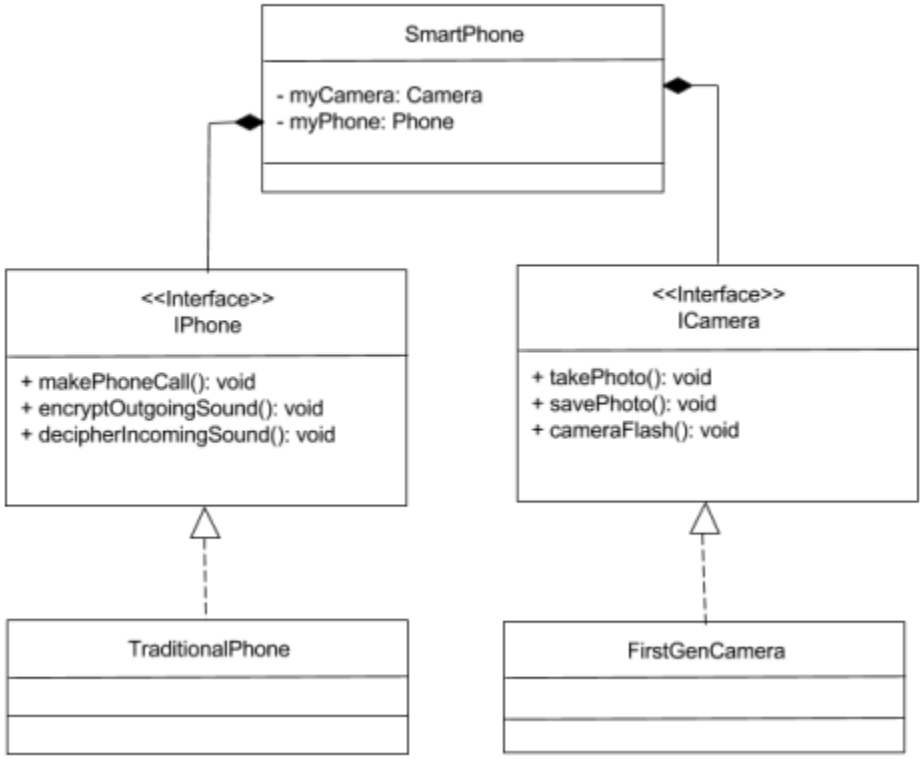
- **Example**-consider the following code

```
public class SmartPhone {
private byte camera;
private byte phone;
public SmartPhone() { … }
public void takePhoto() { … }
public void savePhoto() { … }
public void cameraFlash() { … }
public void makePhoneCall() { … }
public void encryptOutgoingSound() { … }
public void decipherIncomingSound() { … }
}
```

if we looked at this code we will find that is a module for OS that evaluate two tasks

capture a photo and make calls but the module is very large and have a lot of behaviours , and if we applied the princpiles we have learned we will find that it has low cohesion and the behaviours not related to each other so we will divide the class and make composition

- **look at this UML diagram -**



- **and the implemintation of the code will be -**

```
public interface ICamera {
public void takePhoto();
public void savePhoto();
public void cameraFlash();
}
public interface IPhone {
public void makePhoneCall();
public void encryptOutgoingSound();
public void deciphereIncomingSound();
}
public class FirstGenCamera implements ICamera {
/* Abstracted camera attributes /
public class TraditionalPhone implements IPhone {
/ Abstracted phone attributes */
}
```

- **and the other class will be -**

```
public class SmartPhone {
private ICamera myCamera;
private IPhone myPhone;
public SmartPhone( ICamera aCamera, IPhone
aPhone ) {
this.myCamera = aCamera;
this.myPhone = aPhone;
}
public void useCamera() {
return this.myCamera.takePhoto();
}
public void usePhone() {
return this.myPhone.makePhoneCall();
}
}
```

Topic 13 (video)

- **Information Design** - usually we use it to hide information from user or other developer to do that we use Acsses modifiers
- There are 4 access modifiers in Java -

-Public :- is accessable from anywhere in code

-Private :- not accesseble within it’s class

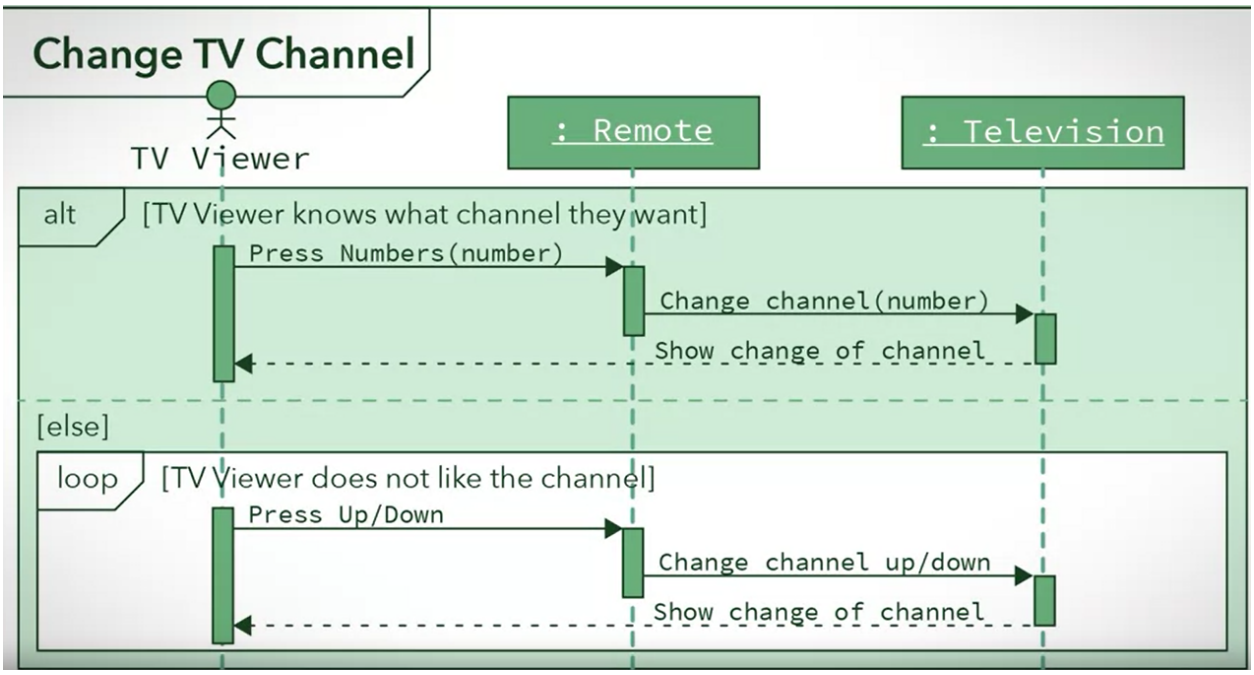
-Protected:- accessable in it’s class and the inherited classes (superclass - subclasses)

-Default :- only allows access to attributes and methods to subclasses or classes that are part of the same package or encapsulation

Topic 14 (video)

- **Liskov Substitution Principle** -asubclass can replace a superclass, if and only if, the subclass does not change the functionality of the superclass

- **UML Sequence Diagram** - are another important technique in software design. They are a type of UML diagram, commonly used as a planning tool before the development team starts programming. Sequence diagrams are used to show a design team how objects in a program interact with each other to complete tasks. In simple terms, a sequence diagram is like a map of conversations between different people, with the messages sent from person to person outline
- For more look at this figure ➖

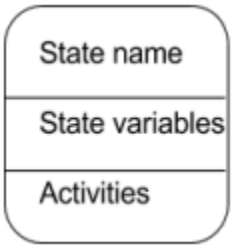


Topic 15 (video)

- **State Diagrams** :-They are a technique used to describe how systems behave and respond
- **The Explanation of State diagrams** -

- A filled circle indicates the starting state of the object. Every state diagram begins with a filled circle.

- Rounded rectangles indicate other states. These rectangles have three sections: a state name, state variables, and activities.



- **Activities** :-are actions that are performed when in a certain state. There are three types of activities
- There are 3 types of activities : -

-Entry activity

-Do activity

-Exit activity

- The arrow indicates transitions from one state to another

$$TheEND$$

Course link :- Object-Oriented Design - Object-Oriented Analysis and Design - Week 1 | Coursera