**Cerinte la soft ce trebuie obtinute cu un bun design in OOD:**

- **Flexible** – putem adauga in soft noi capabilitati

- **Reusable** – codul actual poate fi folosit pentru a adauga noi chestii

- **Maintainable** – code usor de inteles si modificat

**Terms**

* **Models** – sunt modele ale obiectelor ce urmeaza a fi create si construirea la Models si Tehnical Design e golul la Object Oriented Design
* **UML** – unified modeling language

**UML Diagrams**

* UML diagrams sunt produsul la Tehnical Design.
* Ele ne ofera posibilitatea de a crea componente mult mai detaliate ca CRC diagrams
* UML Diagrams se mai numesc Class Diagrams
* In UML specificam fields(numele,tipul) si methods(numele, return tip, parameters) si nivelul lor de accsebilitate

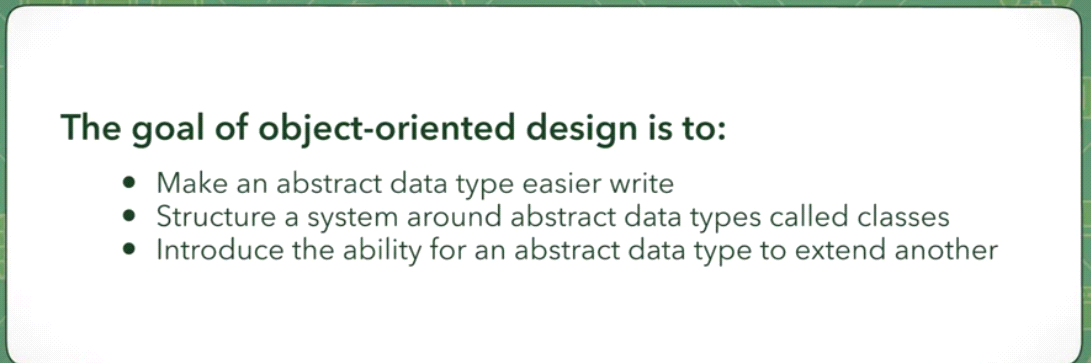
**OOD Role**

* Object Oriented Design are rolurile de a:

1. face un tip de date abstract mai usor de scris

2. structura un sistem in jurul la tipuri de date abstracte, numite clase

3. introduce posibilitatea ca un abstract data type sa mosteneasca de la altul

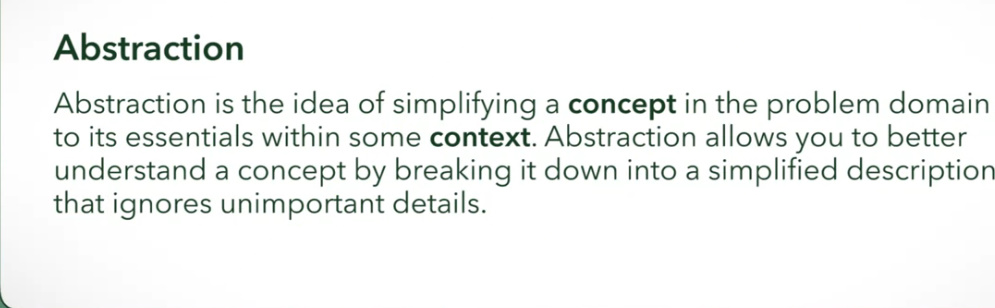


**Exista 4 principii de design**

* Abstraction
* Incapsulation
* Decomposition
* Generalization

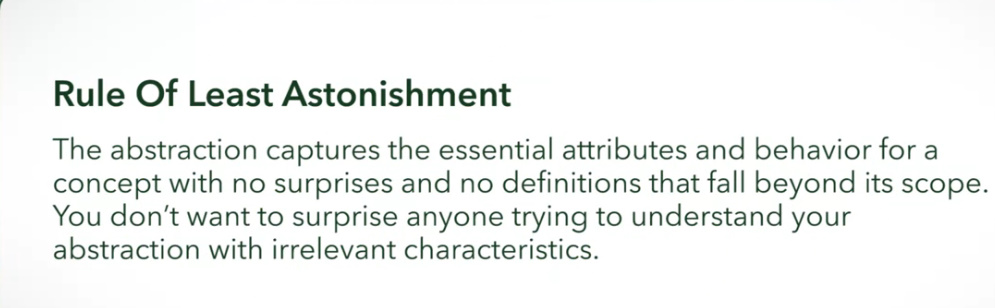
**Abstraction**

* Abstractizarea inseamna sa ascundem detaliile de implementare de user si sa prezentam doar functionalitatile, deci doar detaliile strict necesare.
* In JAVA, o obtinem prin clase abstracte si interfete. Clasele abstracte pot avea si atribute.



Abstractizarea elimina detaliile neimportante.

* Abstractizarea implica regula **Rule of Least Astonishment**

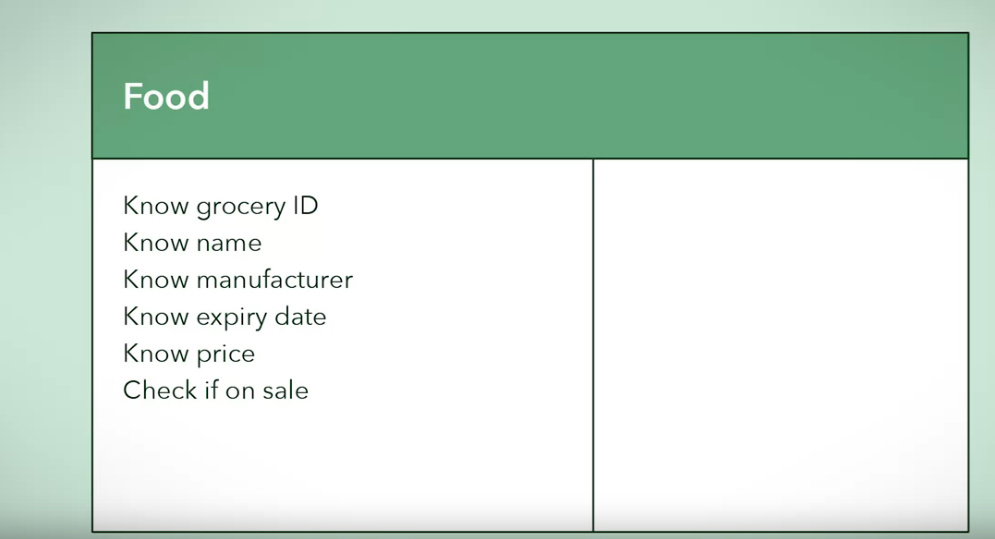


* In Object Oriented Modeling, Abstraction se refera la o clasa.
* Abstraction este foarte mult determinata de context. De ex, daca ne referim la person din soferi, nu o sa ne intereseze doar caractersticile umane, ci si cele ale unui sofer
* De ex, fie un Student:

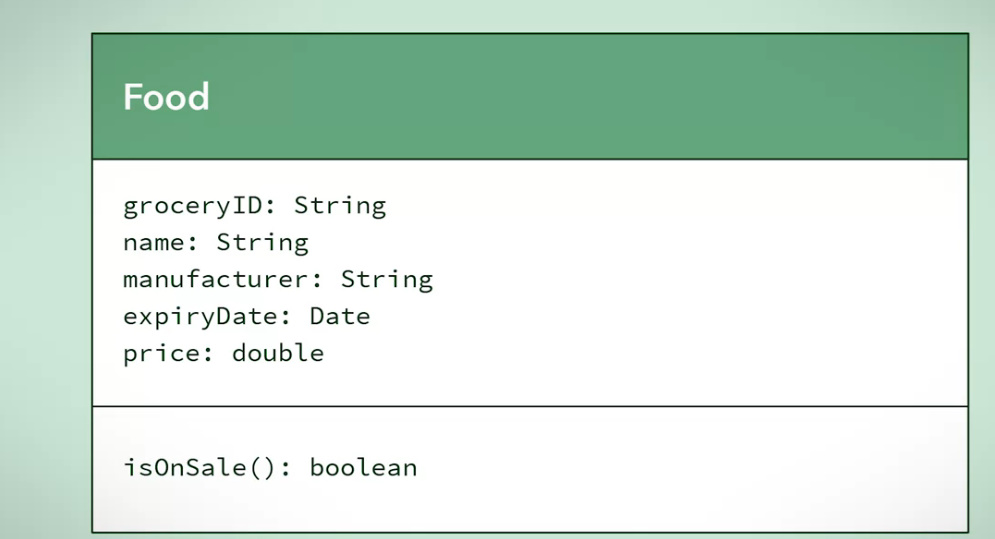


Nu includem ceea ce nu intra in Student

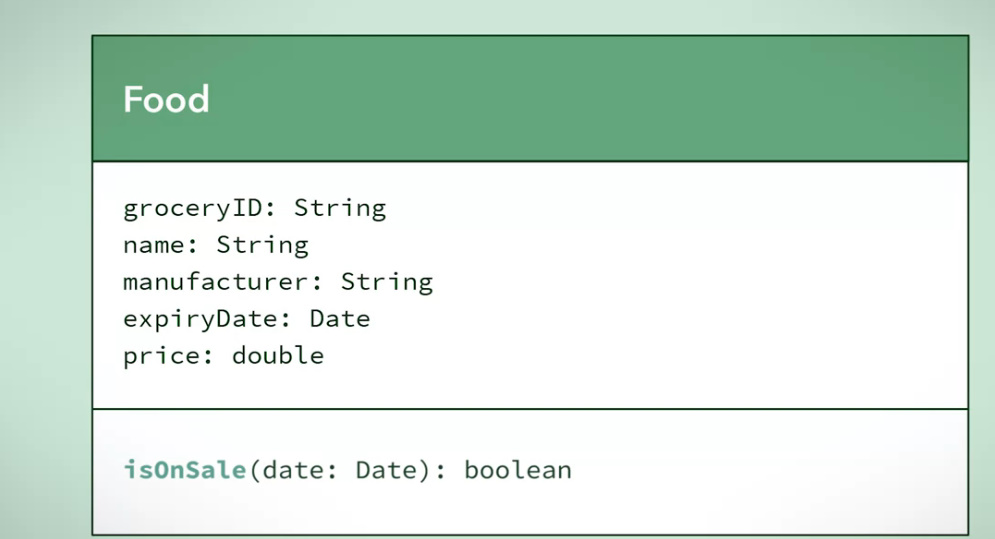
* Abstraction trebuie sa descrie **atributele** si **behaviors**
* **Atribute** – fields
* **Behaviors** – responsabilities sau le zicem metode in Java
* Fie clasa Food. CRC ar arata asa:



si UML

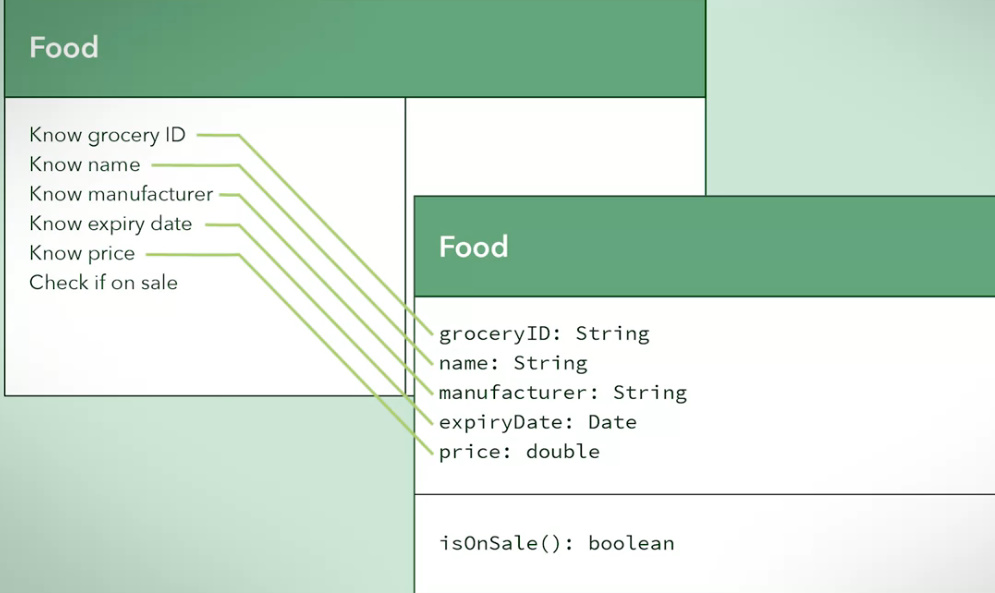


sau asa



**Daca facem comparatie intre CRC si UML, vedem ca responsabilities din CRC devin fields sau methdods in UML**

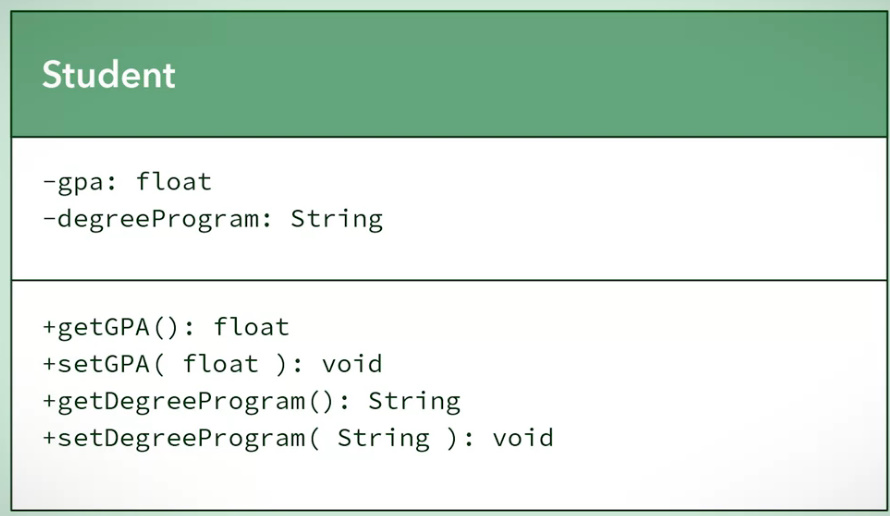
* **CRC sunt pentru a ne focusa pe problema, nu pe implementation! URL e pentru a ne ajuta sa scriem codul.**

****

**Encapsulation**

* **Encpsulation** – maparea datelor(fields,methods) intr-o singura unitate(clasa), care le face sa lucreze ca un tot intreg, si restrictionam accesul la anumite detalii(private) direct.
* **Encapsulation** prevede ca obiectele create de la o clasa au deja propriile valori la atribute.
* Encapsulation prevede si data hiding, adica sa putem accesa fieldurile doar prin metode.
* Encapsultion e ca un fel de implemenatare pentru abstraction
* 
* Clasa este ca un black box, care ne ofera date de iesire, dar nu ne arata cum, caci restrictioneaza accesul la ceea ce nu trebuie sa vedem.

De ex, fie clasa Student



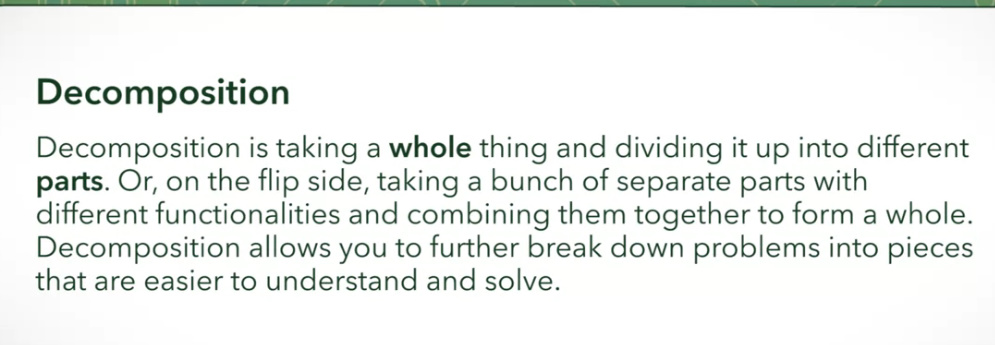
- inseamna private

+ inseamna public

# - protected

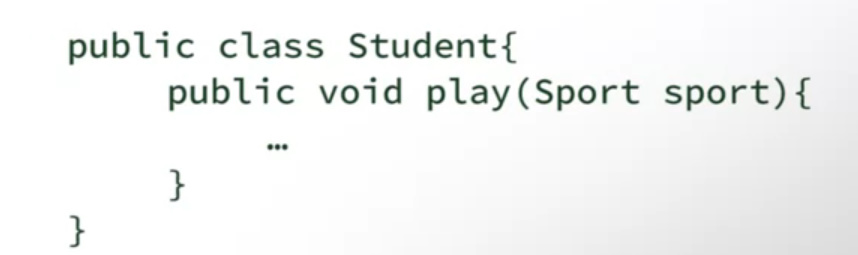
Folosim setteri si getteri pentru Incapsulation ca sa limitam acces

**Decomposition**



* Cea mai buna metoda este de a lua un component si de a identifica reponsabilitatile sale, si pentru fiecare responsabilitate sa cream un component separat.
* De exmplu, masina are volan, roti, scaune, motor etc.
* Numarul de elemente care se descompun poate fi fixed sau variabil. De ex, o masina are mereu 4 roti, dar poate avea mai multi pasageri
* **Sharing** - Unele componente pot impartasi bucati comune
* **Lifetime** – componentul si bucatile pot avea acelasi lifetime sau diferite.
* Sunt 3 tipuri de relationship in decomposition:

- **Association** – un **fel de** conexiune intre 2 clase in care clasele comunica una cu alta, dar nici una nu o detine pe cealalta si nu e dependenta. Ele pot exista si total independent.Ea doar arata ca clasele interactioneaza intre ele, nu comunica direct, si nu arata ce fel de asociere exista intre ele propriu zisa.Asocierea poate fi de tip **1 to many, many to one, many to many, one to one.**  Clasele comunica adesea prin metode, de ex fie Studen si Sport

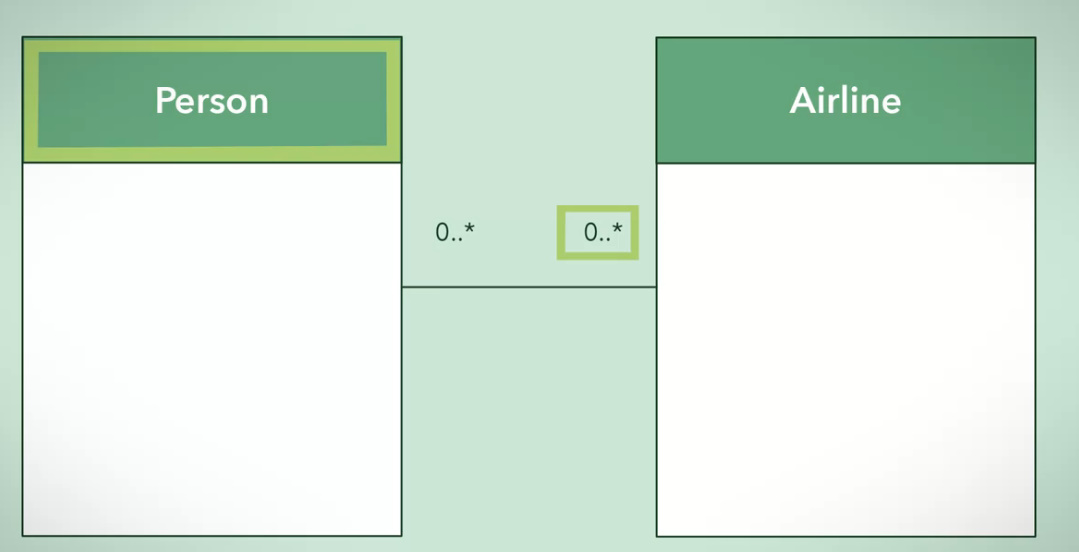


Un Student poate juca un Sport, dar ele nu stiu de exista lor. Nici una nu are field object catre alta. Putea fi si un List<Sport> ca sa fie One to Many

**Aggregation** si **Composition** sunt forme de Association.

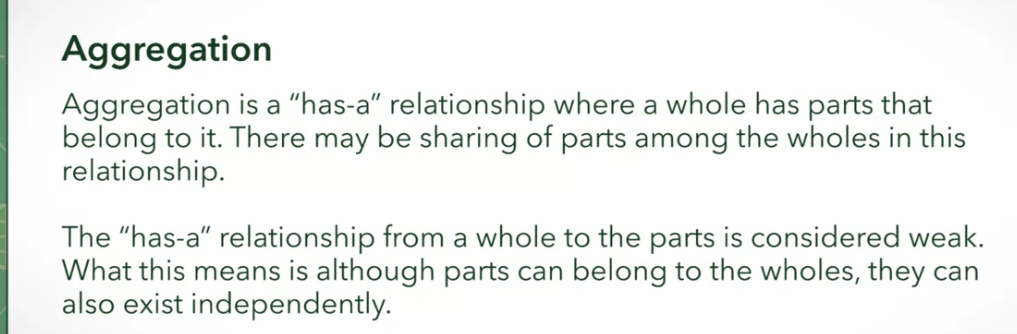


Grafic, in ULM, asta se face cam ca si in Entity Relationship Diagram

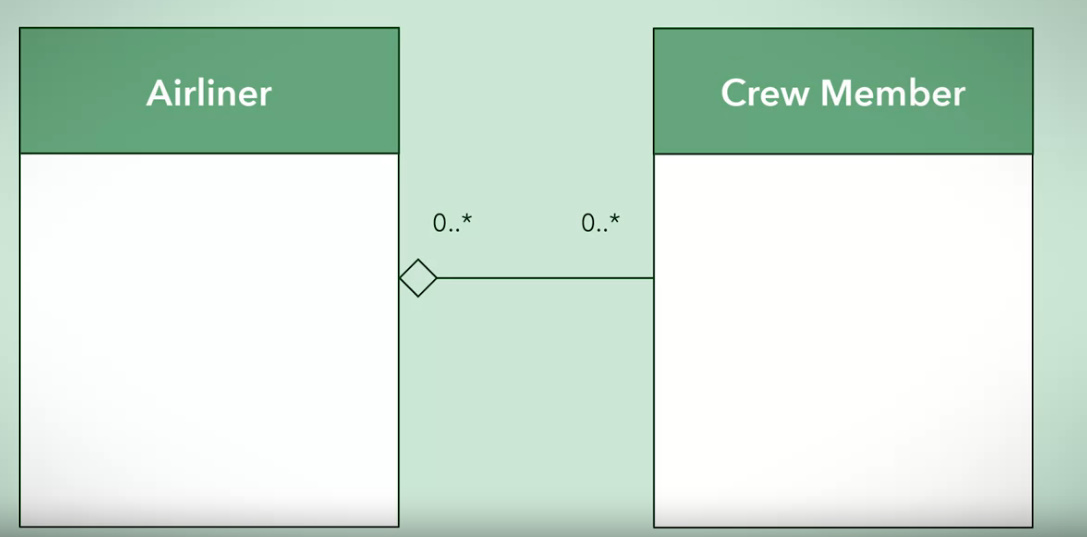


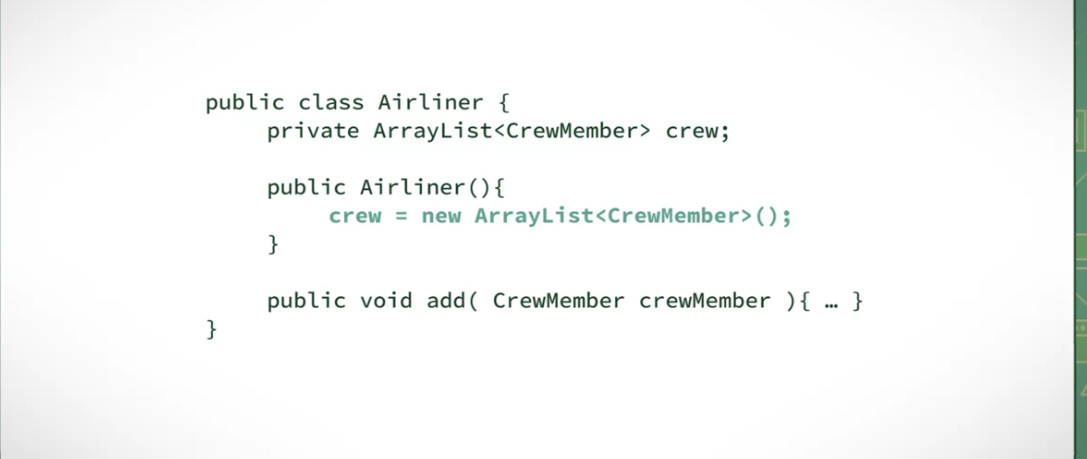
O persoana poate avea mai multe airlines. Cate airline poate avea o persoana e pus langa Airline

- **Aggregation** – Este o forma speciala de Association, bazata pe **Has A**. Adica, o clasa comunica cu alta clasa prin obiect field. Totusi, in cazul la aggregation, relatia dintre clase este **unidirectionala**. Ambele clase pot exista **independent**, una fara alta, adica obiectele nu depind una de alta, de aceea Aggregation este weak(distrugerea la un obiect nu duce si la distrugerea celuilalt). De ex, avion si Echipaj. Avionul nu inceteaza sa existe fara Echipaj, si echipajul nici atat fara avion,dar Avion are totusi in el Echipaj ca field, desi nu e dependent de el si nici invers.

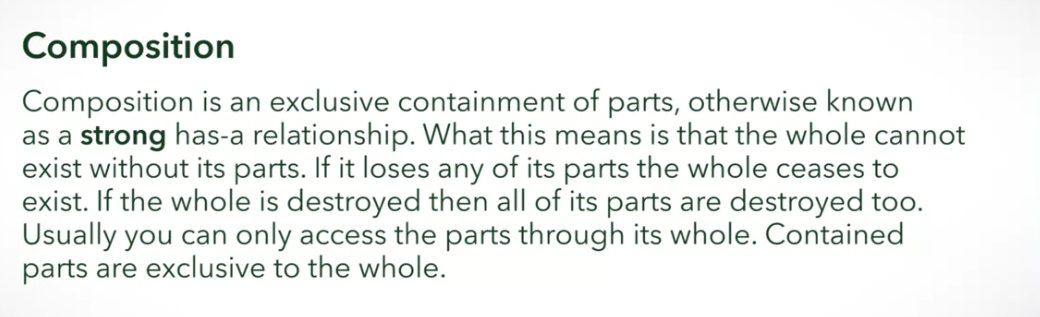


In UML, mai adaugam un Diamond operator(romb) pentru a arata in care clasa cealalta se gaseste. Deci, care clasa are un field object.

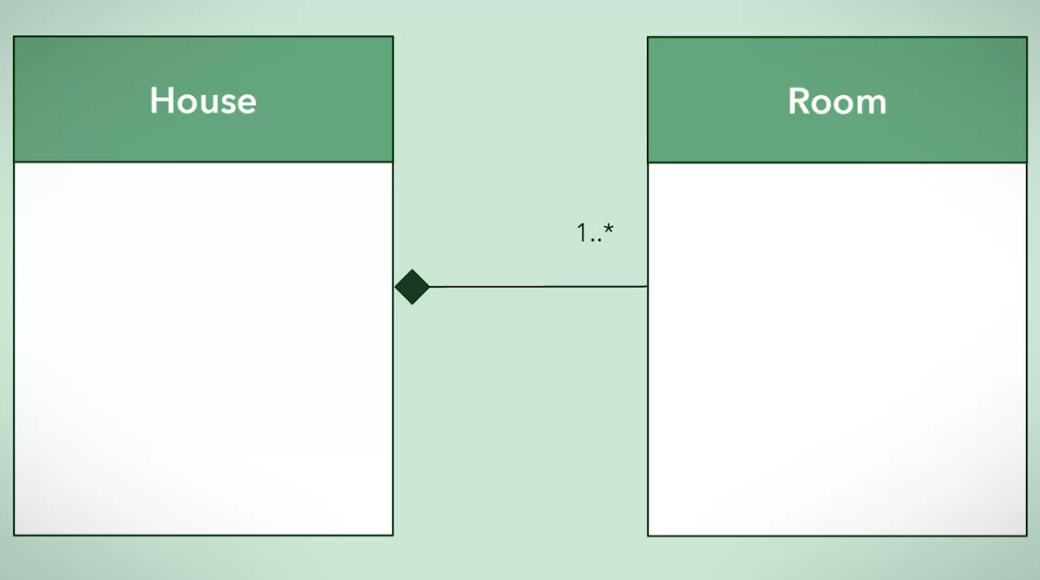


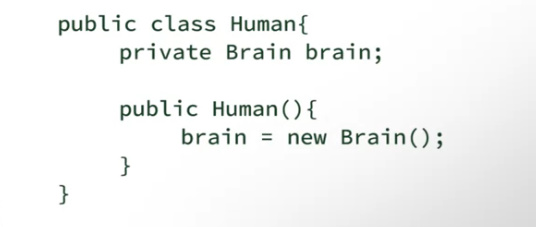


- **Composition** – Este o forma stricta de Aggregation, in care clasele **depind** una de alta, si relatia este **Part Of**.O clasa depinde de cealalta puternic, si daca acea clasa e distrusa, se distruge si cealalta, dar cealalta, desi nu are un field la prima, oricum nu ar putea exista fara ea, caci e in interiorul ei. Deci, un obiect e format in bucati de alte obiecte si nu poate exista fara ele. Legatura este **tot unidirectionala**.



In UML putem un diamond operator colorat la obiectul care este ca un intreg, care va avea cealalta clasa ca field, adica care o contine pe cealalta clasa ca field. House va avea List<Room>, dar Room nu va avea un House field, ca si asa e deja in el.





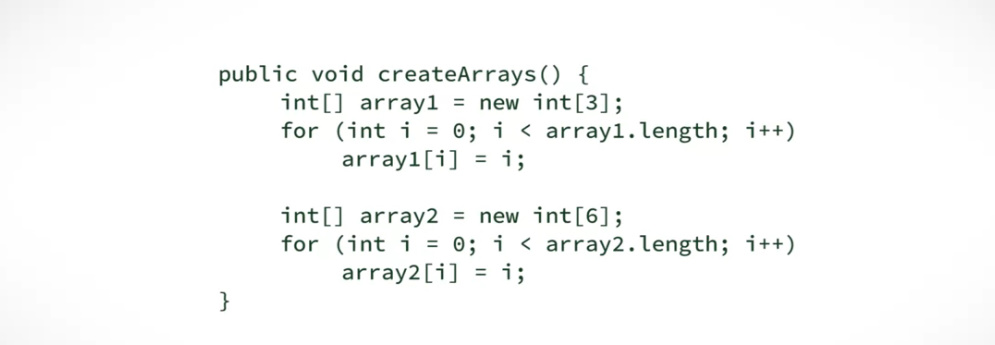
Brain nu va avea in el un Human ca field, caci el deja e in Human, nu poate exista fara Human, si Human nu poate exista fara Brain in el

**Composition vs Aggregation**

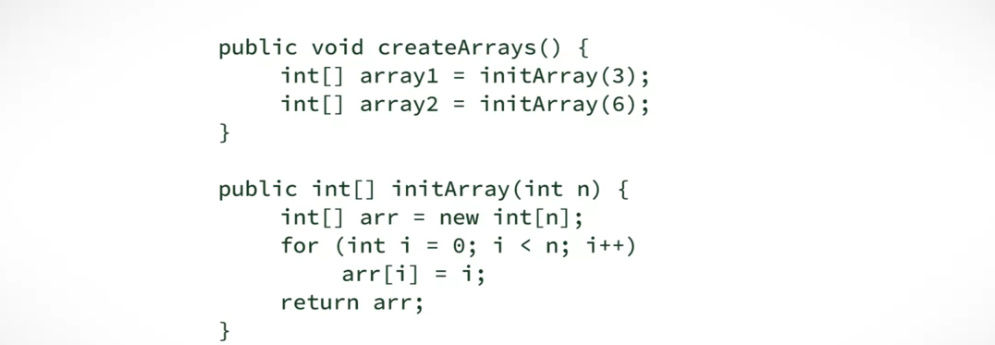
* Composition are un Part-Of, Aggregation Has A
* In Composition, obiectele nu pot exista una fara alta, in Aggregation sunt independente si pot
* Aggregation e weak relation, composition e strong

**Generalization**

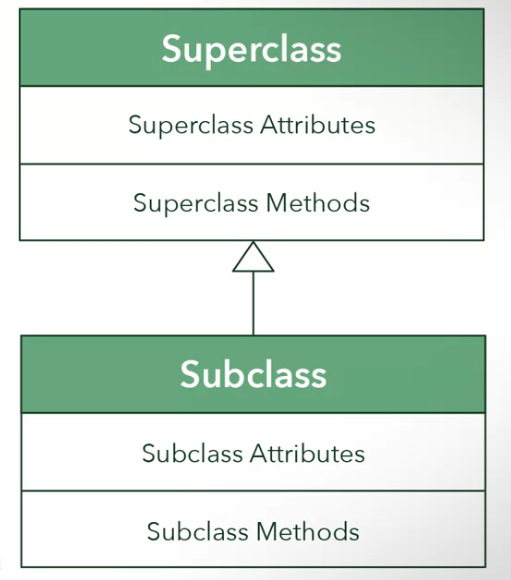
* Ne ajuta sa reducem nivelul de redutanta cand rezolvam probleme.
* De exemplu, metodele ne permit sa generalizam un code ce ar trebui repetat de mai multe ori in diferite parti. Asa, il generalizam in metoda pentru diferite inputuri si nu il repetam mereu.
* DE ex:

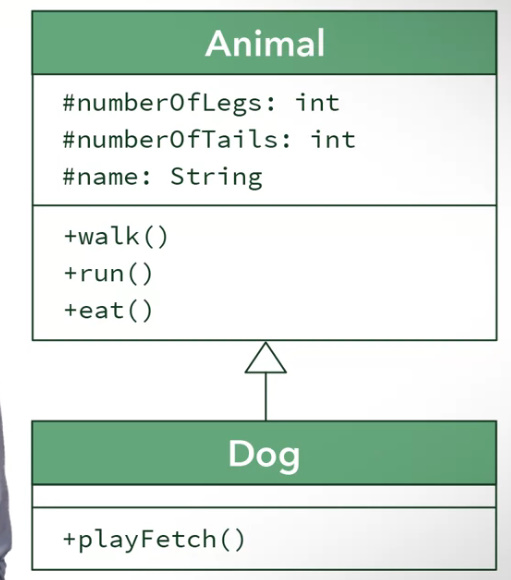


crearea acestui array poate fi generalizata intr-o metoda si initializarea tot



* Generalization poate fi facuta si la nivel de clasa, prin inheritance, si anume prin clase abstracte si interfete.
* Generalization deci se refera si la a crea clase generale. De ex, clasa Animal poate avea subclasele Dog, Cat etc.
* Generalization ne ajuta sa respectam regula DRY(don’t repeat yourself)
* In UML punem o sageata:

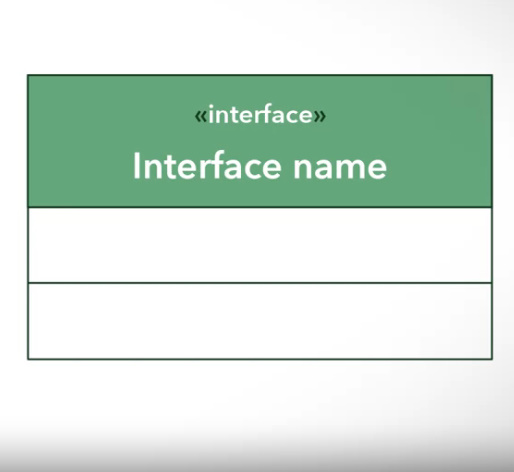




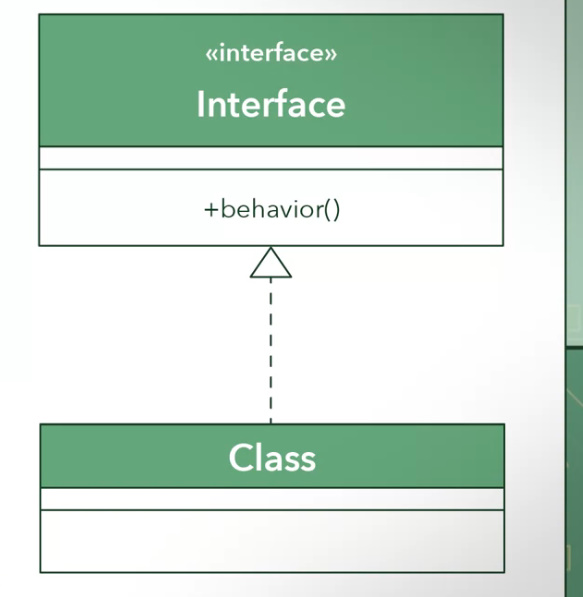
# - protected

**Interface in UML**

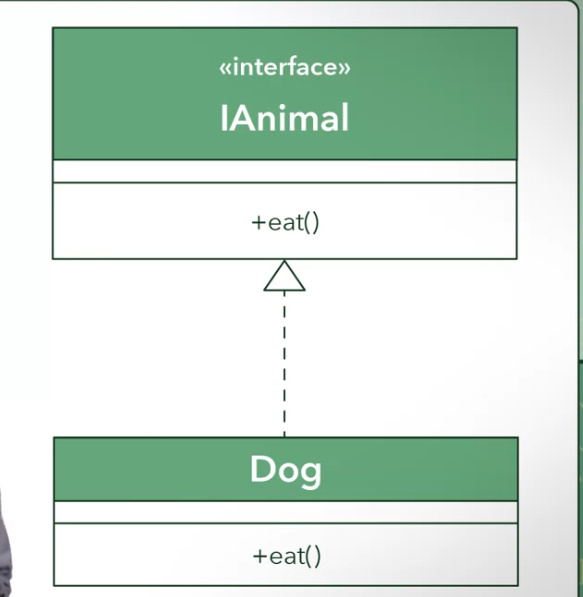
* Interface sunt create exact ca clasele, doar ca deasupra la nume se pune <<interface>>



Iar clasele care implementeaza interfetele se unesc cu ele prin sageata intrerupta



Cand o metoda e suprascrisa, tot o scriem in clasa, chiar daca ea deja a fost scrisa in interfata



**De ce Java nu suporta multiple inheritance**

* Poate crea data ambiguity atunci cand extindem mai multe clase, si ele pot avea fields sau metode identice, si Java nu va sti pe care sa o aleaga