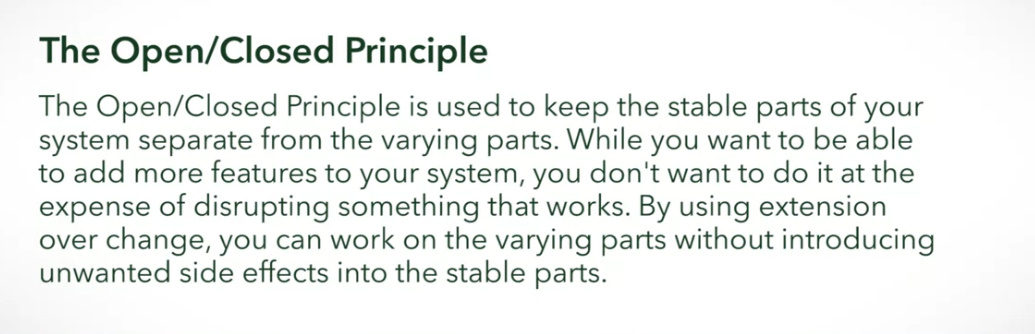
**Liskov Substitution**

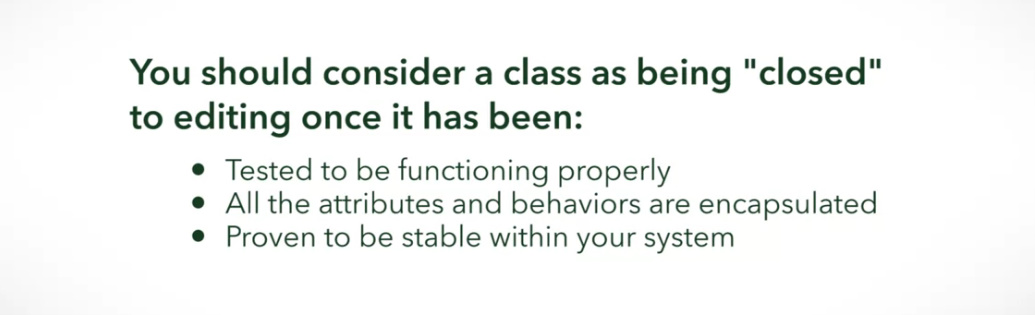
***If a class, S, is a subtype of a class, B, then S can be used to replace all instances of B without changing the behaviors of a program.***

* Deci, cand folosim inheritance, e foarte important ca subclasele sa se comporte identic cu superclasele.
* De ex, fie A o clasa. B o extinde. B trebuie sa faca tot ceea ce face clasa A, doar ca sa mai adauge anumite chestii separate. B nu trebuie sa modifice scopul pentru care A a fost creata
* Daca il vom folosi pe B in locul lui A, asta nu trebuie sa influenteze nicidecum programul

**Open/Closed Principle**

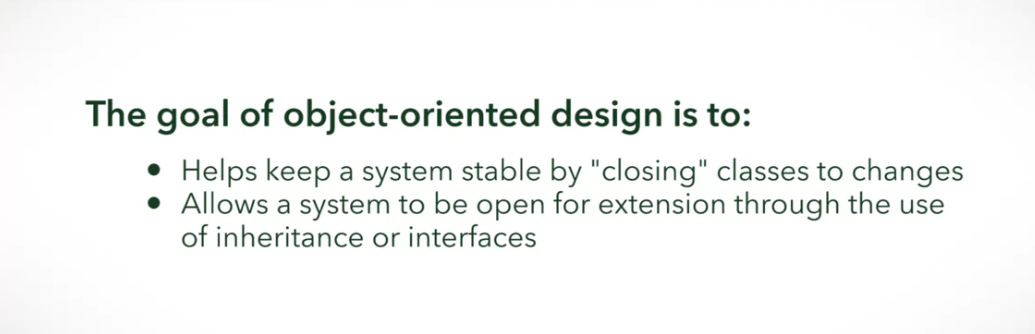


* Clasele trebuie sa fie open pentru extension si closed pentru changes

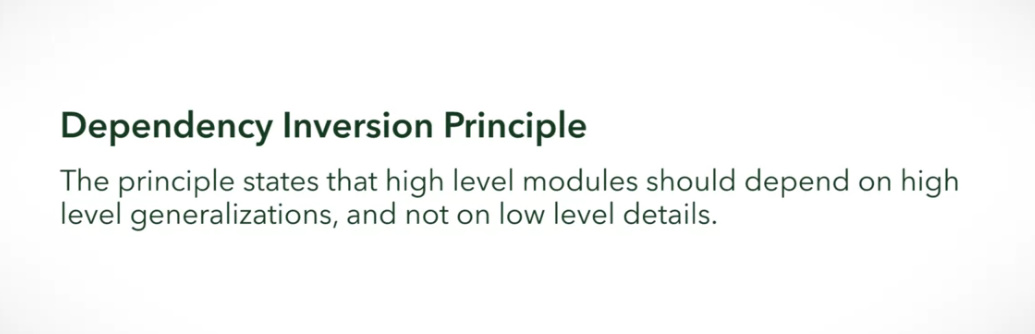


* Nu inseamna ca in timpul dezvoltarii la soft nu putem sa modificam clasele. Putem, dar odata ce deja clasele au fost create, au inceput sa fie folosite in system, nu le mai modificam mai mult. In faza de crearea a designului e normal sa le modificam, dupa deja nu. Asta inseamna **Closed**
* Exista 2 metode e a extinde capacitatile sistemului, deci de a adauga noi comportamente:

1. Cream o sub clasa prin a extinde superclasa si adaugam noi fields si methods
2. Super class este abstracta si forteaza inheritance si polimorfismul



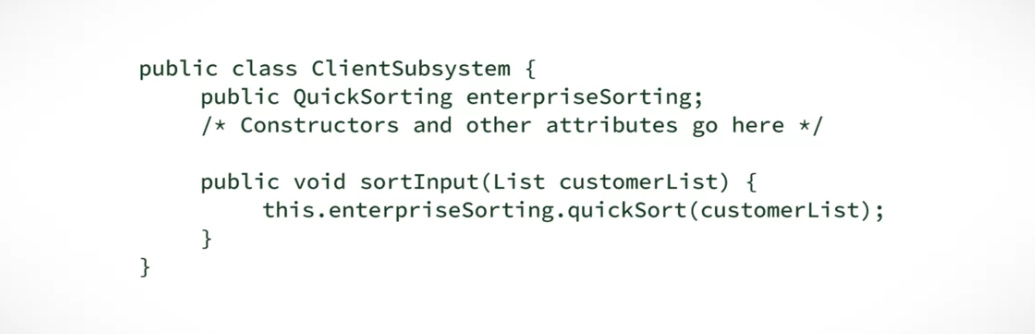
**Dependency Inversion**

* Clasele noastre au adesea colaboratori. E foarte important ca clasele date sa nu fie dependente de colaboratiri, adica de o implementare concreta pentru colaboratori
* 

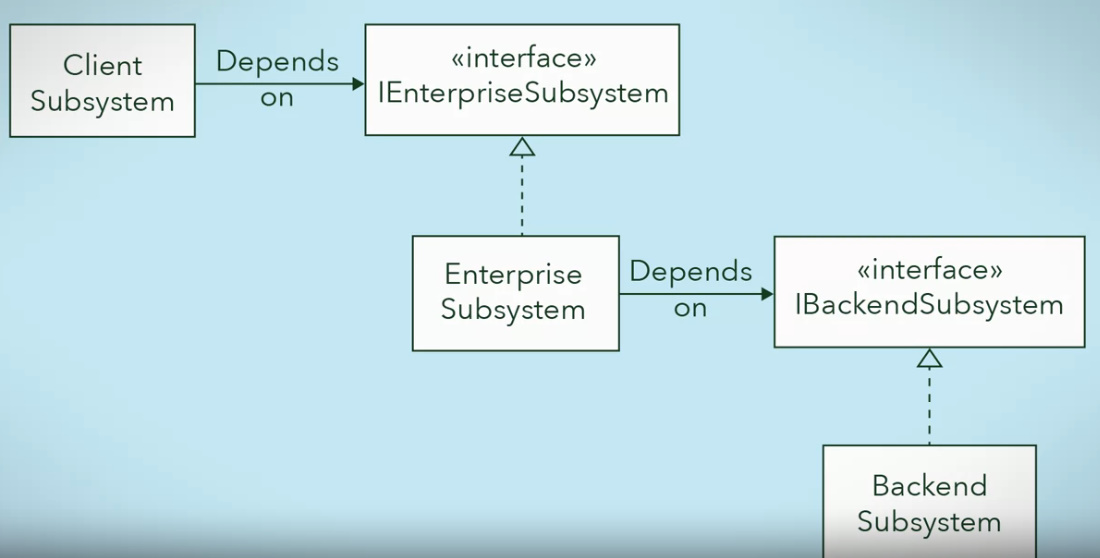
Deci, **Dependency Inversion** inseamna ca client class trebuie sa depinda de interfete si clase abstracte(sa le aiba ca fields), nu implementari concrete. Totul trebuie generalizat.

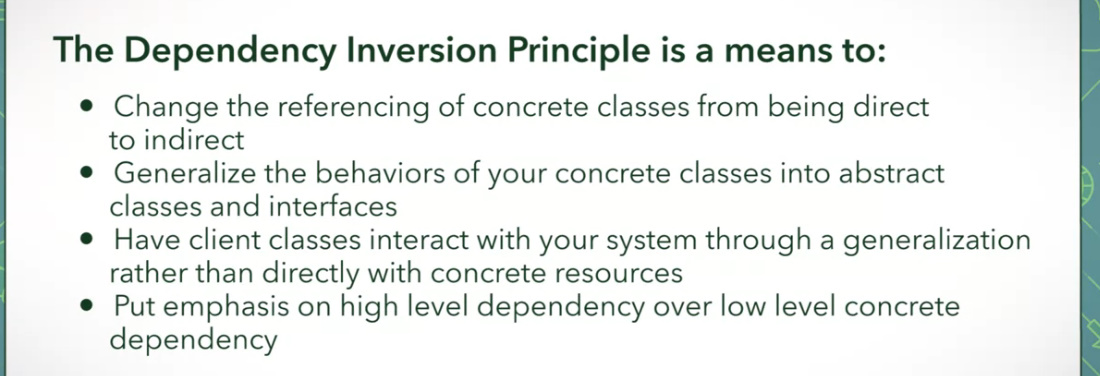
**high level resources** – interface si abstract class

**low level resources** – clase concrete, implemntari

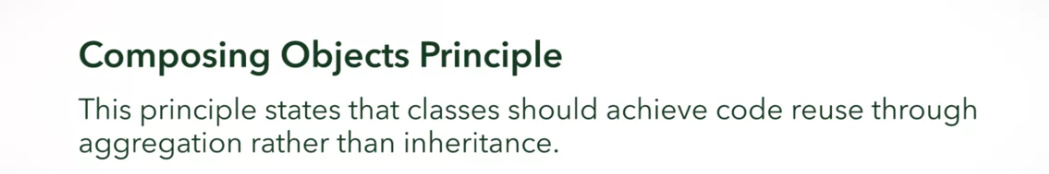
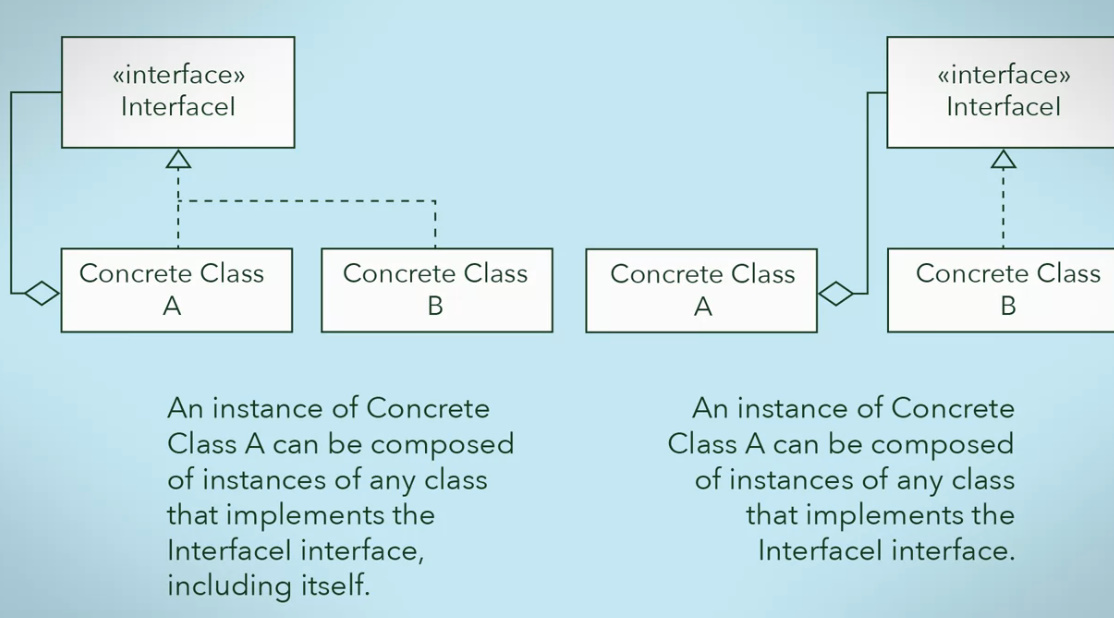
* 

QuickSorting este o clasa concreta. Dar,daca vom decide ca vrem deja sa folosim MergeSort, va trebui sa facem multe modificar in ClientSubsystem

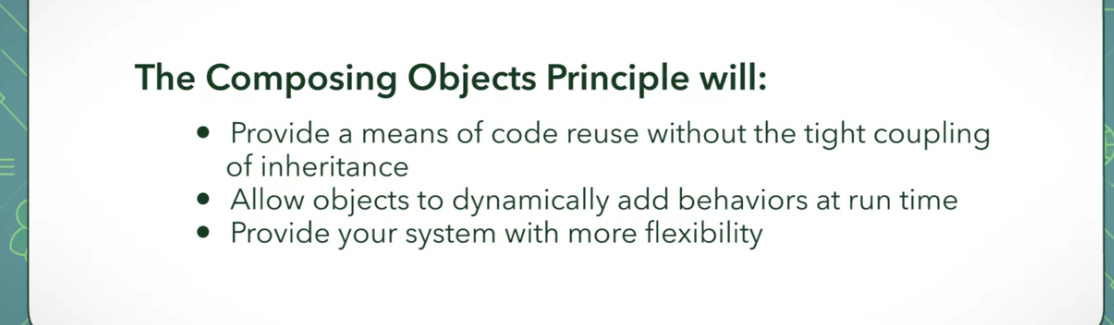
* 



**Composing Objects**

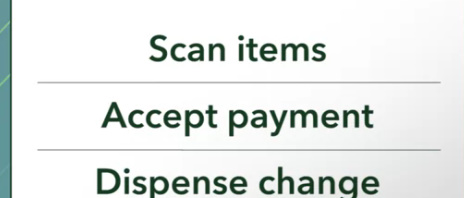
* 
* Composing Objects descurajeaza inheritance, si pune mai mult accent pe aggregation, adica pe folosirea la fields pentru obiectele de care el depinde, de cat de a folosi inheritance
* 
* Composing Objects prevede deci sa pastram clasele separate, si sa le facem sa colaboreze prin Aggregation
* Asa, sistemul e mai flexibil la schimbari
* Cu composing objects putem modifica comportamentele la obecte la runtime, caci fields pot fi modificate oricand la runtime
* La Inheritance, comportamntele se definesc doar la compiletime
* Totusi, composing objects are si dezavantaje





**Interface Segregation**

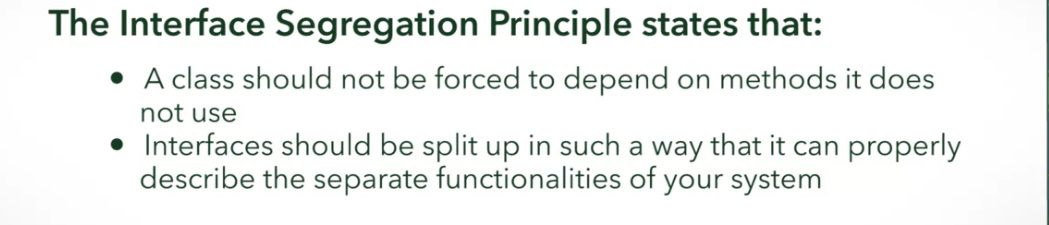
* In OOD, Interfetele au un rol foarte important, de aceea trebuie sa ne straduim sa le folosim cat mai mult
* Totusi, daca o interfata are mult prea multe resposnabilitati, adica metode, asta deja poate cauza probleme
* De ex, la un magazin putem scana produsele in 2 feluri: La casa cu casier sau la casa cu auodeservire
* O interfata comuna ar putea fi creata si ea ar avea asa responabilitati:

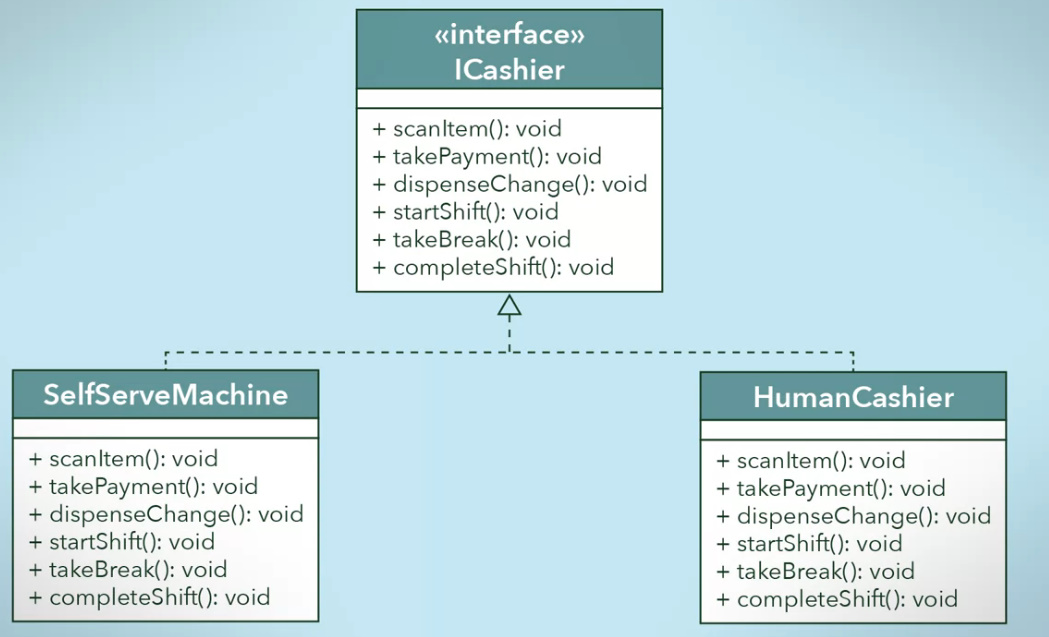


Astea sunt comportamente comune

Totusi, sunt si comportamente pe care ele nu le au in comun

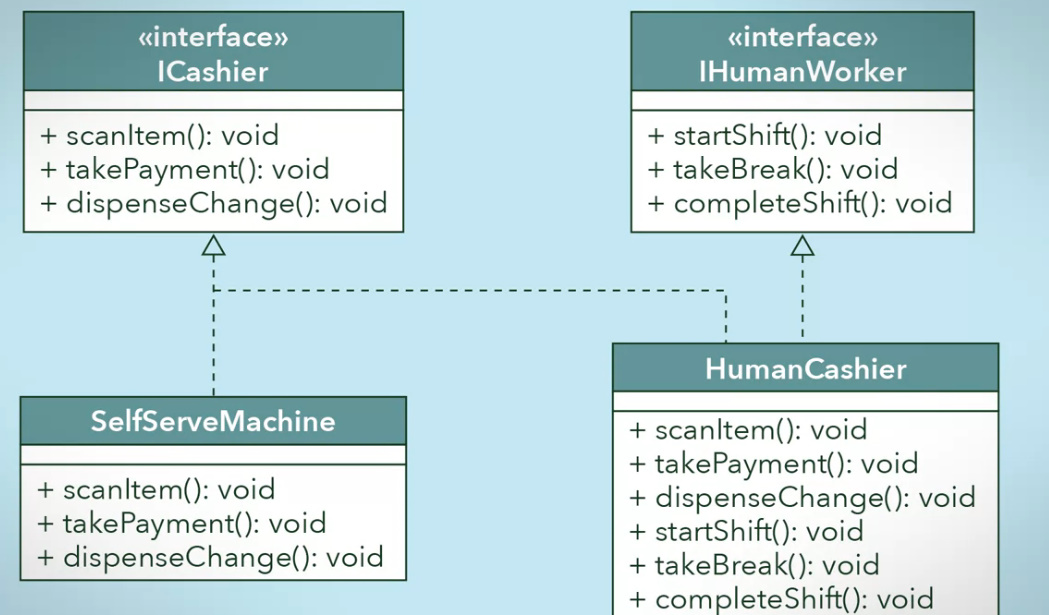
Gen, o casa cu casier poate fi in repaus, daca casierul a luat pauza

* Aceste comportamente legate de om pot fi puse in CasierCache clasa,dar aici e o problema. Client nu mai poate apela aceste metode umane prin interfete, insa daca punem implementari concrete, el devine dependent de 2 clase concrete si nici nu are nevoie de ambele deodata, ci de una.
* Totusi, pentru a rezolva problema data putem folosi Interface Segregation
* **Interface Segregation** se bazeaza pe aceea ca interfetele mari trebuie generalizare in interfete mai mici. O clasa nu trebuie sa fie fortata sa implementeze niste metode pe care nu le foloseste. Asta inseamna ca clasele ce implementeaza interfetele nu trebuie sa aiba implementari doar de frumusete a metodelor din interfete.
* ****
* De ex, daca am face asa pentru casele din magazin:



Avem problema. takeBreak() nu are ce cauta in SelfServeMachine, caci ea nu are cum lua pauza, dar interfata o obliga sa o suprascrie, si o va face, dar cel mai probabil doar prin a pune { } de frumusete si gata. Asta e rau de tot.

* Aplicam Interface Segregation si despartim Icashier interface in mai multe



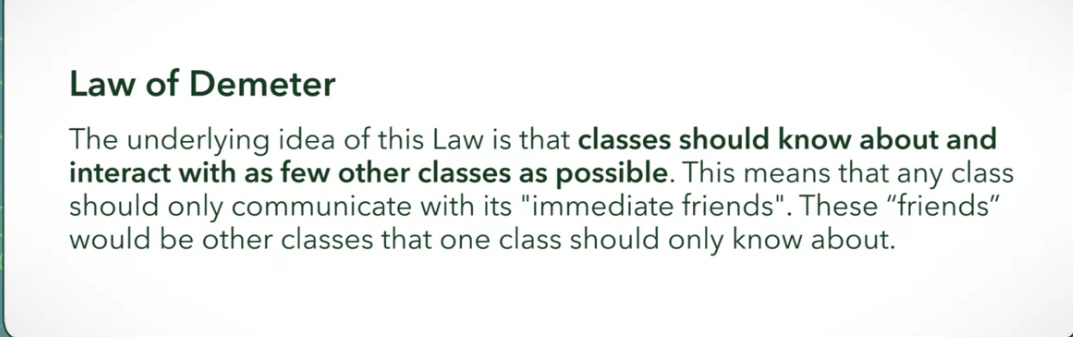
In loc de o intefata generala, acum avem 2 interfete particulare ssi vedem ca SelfServeMachine implementeaza doar Icashier, dar Humasncashier implementeaza si Icashier si IhumanWorker

Icashier are metodele generale pentru ambele si IhumanWorker doar pentru HumanCashier

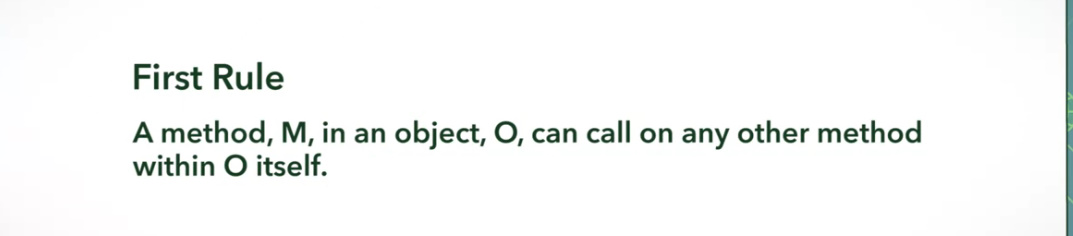
* Teoretic, parca am putea sa facem ca IhumanWorker sa implementeze Icashier, dar daca de exemplu Icashier, la un moment dat, va avea metode ce nu sunt pentru HumanCashier, nu vom fi capabili sa despartim cumva interfetele

**Principle of Least knowledge**

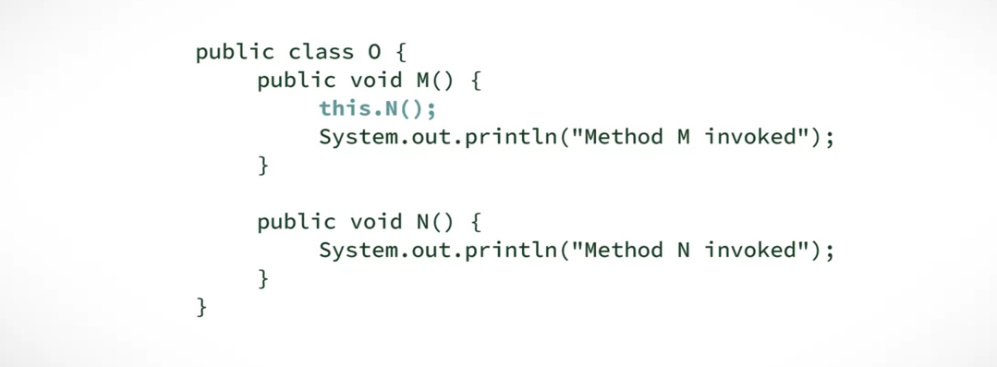
* O clasa trebuie facuta in asa fel incat nu trebuie sa depinda sau sa stie de orice clasa din system
* Limitand care clasa comunica cu altele putem implementa principiul Least knowledge
* **Least Knowlege** este realizata printr-o lege numita **Law of demeter**

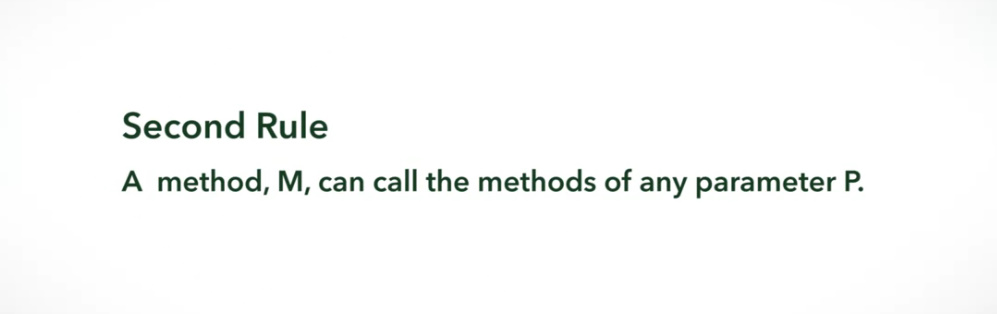


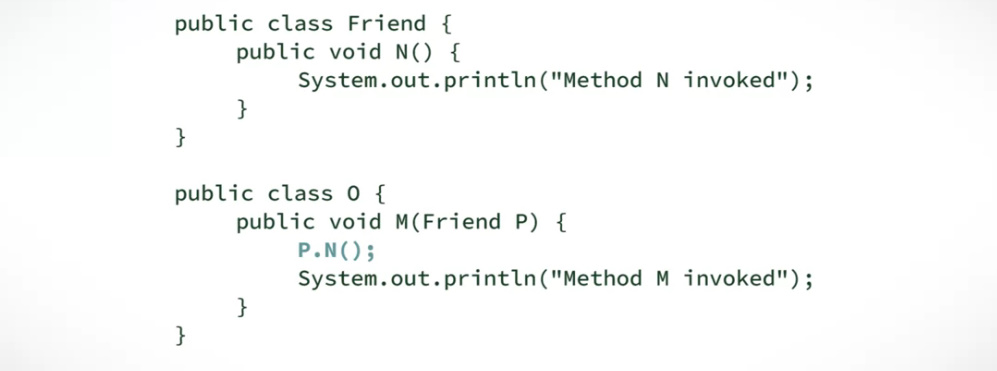
* Law of Demeter are urmatoarele rules:

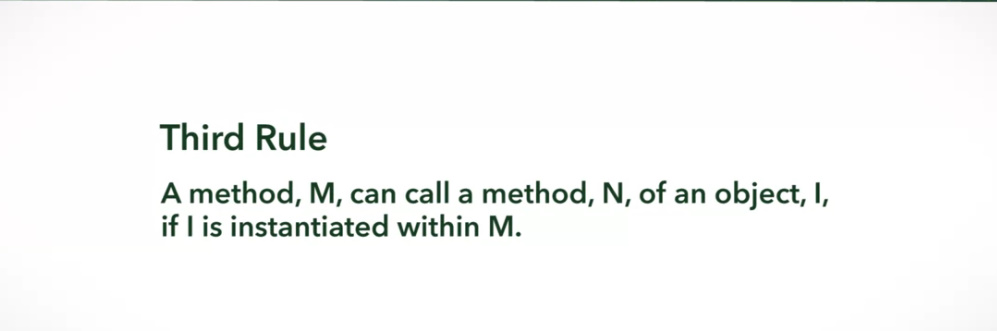


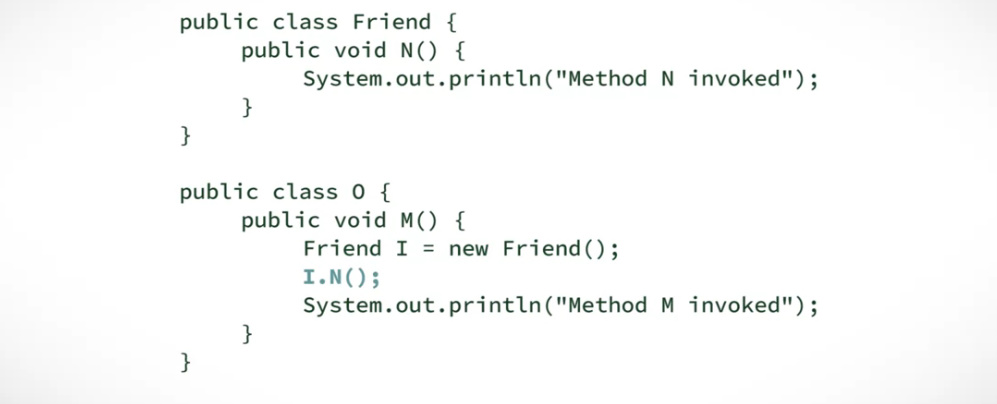
Deci, o metoda dintr-o clasa poate apela orice alta metoda tot din aceeasi clasa

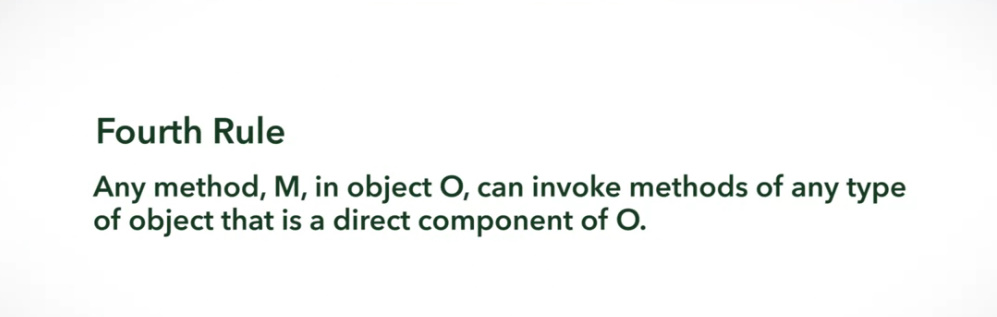


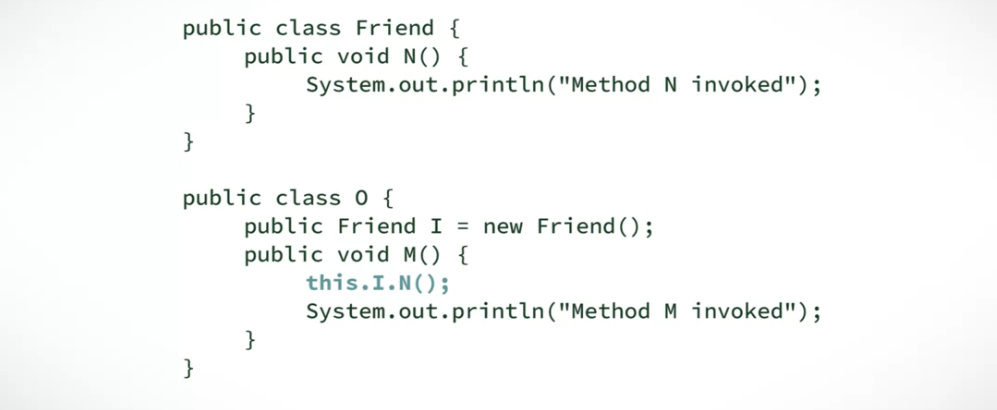
1. 

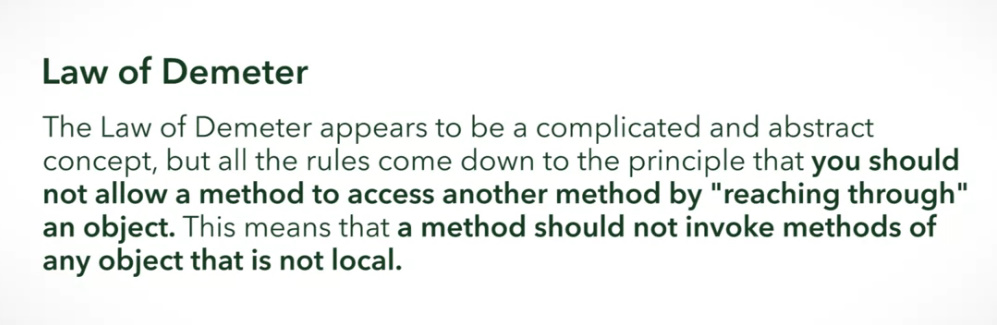


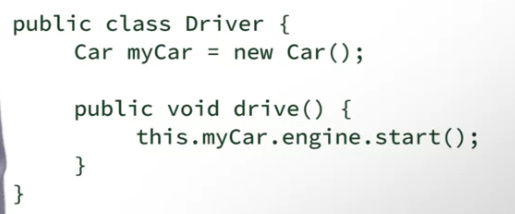
1. 



1. 



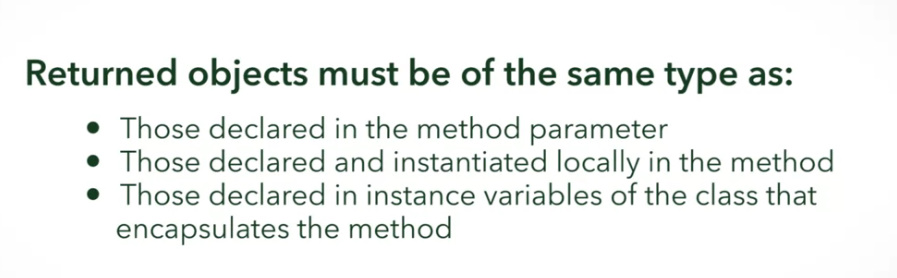
* 
* **Reaching Through** – inseamna ca accesam obiecte ce deja ies din clasa si nu trebuie sa ne bagam la ele

****

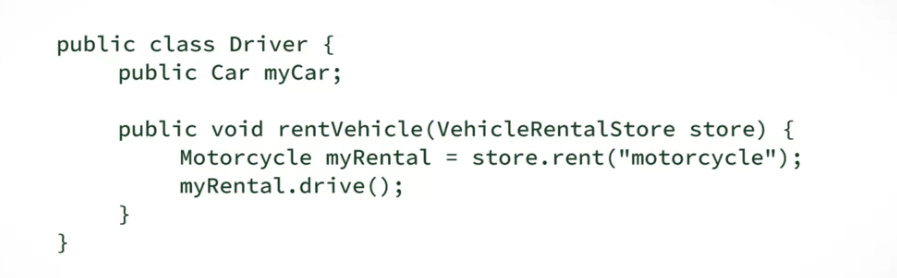
Nu trebuie sa accesam obiectele ce nu se gasesc direct in clasa noastra, gen cum facem cu metoda engine din myCar

De exemplu, cand conducem o masina, nu ne bagam la motor sau la roti, ci folosim ce ne ofera masina, ca volanul, pedalele etc.

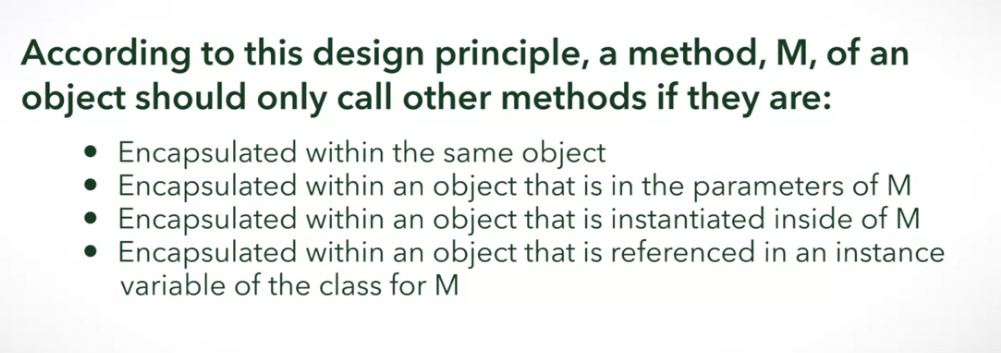
* Problema e ca nici nu stim ce fel de obiect e engine



* Driver stie doar de Car si VehicleRentalStore, dar nu si de Motorcycle



* Clasele nu trebuie sa se bage prea adanc in altele



Regula data nu poate fi aplicata mereu, dar trebuie sa ne straduim