**Mostenirea**

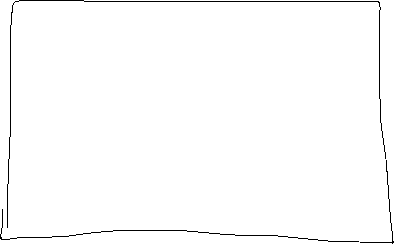
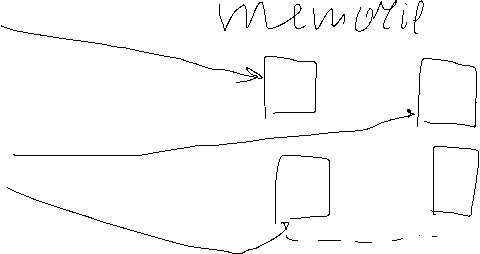
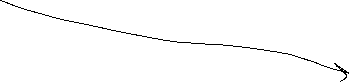
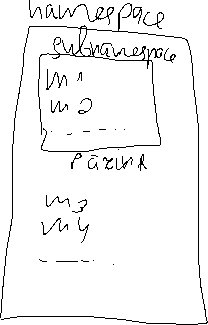
* Mostenirea reprezinta o relatie **“este un”** sau **“is a”**
* **O clasa derivata copie si tipul superclasei**. De ex, daca avem clasa Parinte si clasa Copil, care mosteneste de la Printe, **clasa Copil este si de tip Parinte,** de aceea putem face asta:

**Parinte obj1 = new Parinte();**

**Copil obj2 = new Copil();**

**obj1 = obj;**

Acesta este upcasting. Cand o clasa mosteneste de la o superclasa, la crearea obiectului se creaza intai un bloc secundar, unde vor fi membrii mosteniti legati de zonele lor de memorie, si apoi in bloc al obiectului, cel principal adica, se vor adauga membrii sai proprii. Deci, o astfel de egalare pur si simplu permite lui obj1 sa acceseze blocul secundar din obj2 care i-ar corespunde. **Iata de ce, un obiect poate fi de mai multe tipuri!**



* Relatia **„has-a”** inseamna ca clasa are drept membru un obiect de un tip, nu mosteneste clasa acelui obiect.

class copil{

Parinte obj;

}

* O ierarhie este o structura arborescenta, de exemplu:

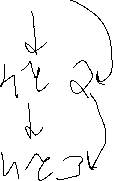
class nr1{}

class nr2 extends nr1{}

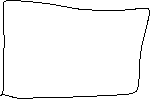
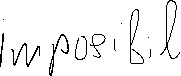
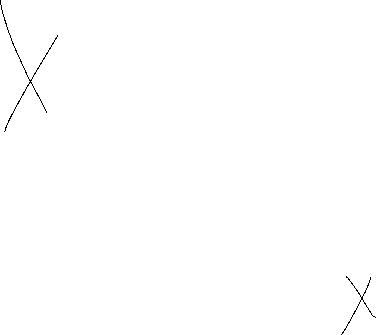
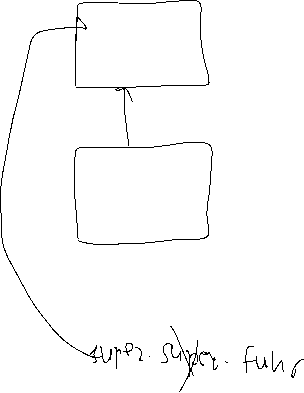
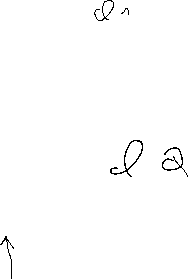
class nr3 extends nr2{}

....................................

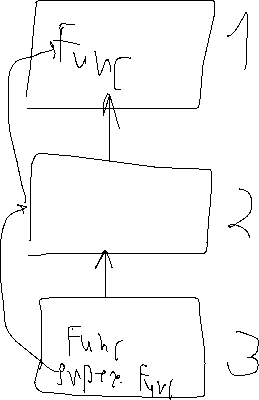
Adica, o ierarhie prevede ca exista o clasa pe care o mostenesc toate clasele ce sunt mostenite de cea de la ultimul nivel.



* **Override** – pentru metode
* **Shadowing** – pentru variabile. Daca o clasa copil mosteneste un atribut de la parinte si totusi ea mai defineste odata acel atribut, ea va avea deja 2 atribute identice. Unul mostenit si altul propriu. Ea il va putea utiliza pe cel propriu, dar va putea utiliza metode mostenite ce vor folosi deja atributul clasei de baza. **Atributele nu sunt virtuale!**
* Daca clasa copil mosteneste anumiti membri si dorim sa accesam anume membrii din clasa parinte anume, pur si simplu utilizam **super.metoda().**
* **super** este o referinta la blocul-ul clasei parinte.
* **~~Super.super.metoda()~~ – nepermis!**

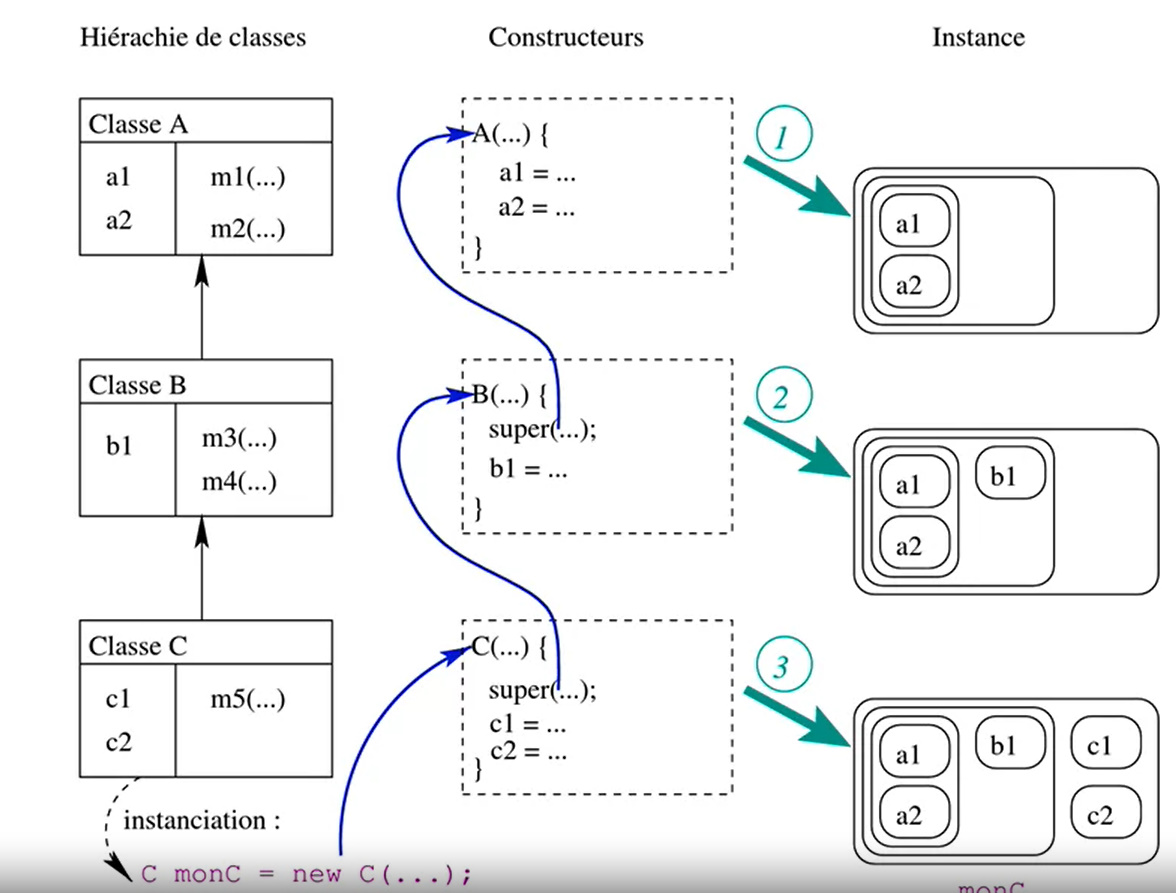


Dar daca, de ex, avem o asa ierarhie de clase, unde clasa3 mosteneste de la clasa2 si clasa2 de la clasa1, si clasa3 ar suprascrie o metodata din clasa1 si clasa2 nu, super.metoda() ar face trimitire la la metoda din clasa2 mostenita de la clasa1, adica oarecum super ar face trimitire la clasa1, teoretic doar.



**Constructori**

* Initializarea membrilor se face in mod normal in constructori. Fiecare clasa are un construtor responsabil de initializarea membrilor sai. Daca avem o clasa ce mosteneste de la alta clasa, e logic ca clasa copil sa isi initializeze proprii membrii si clasa parinte pe ai ei. Iata de ce, in constructorul clasei copil trebuie sa apelam constructorul clasei parinte.
* Constructorul default din clasa copil apeleaza automat constructorul default din clasa parinte, fara a mai scrie super()(spre deosebire de python), insa daca nu exista constructorul default, va trebui sa alegem noi unul.
* **super() trebuie sa fie mereu prima instructiune din constructor**
* Daca clasa parinte nu are un constructor default, clasa copil trebuie neaparat sa aiba cel putin un constructor care sa apeleze constructorul clasei parinte. Daca nu ar avea niciun constructor, acel costructor al clasei parinte nu ar putea fi apelat in cazul dat.
* super() nu poate fi apelat de vreo metoda!
* Cel mai bine e mereu sa facem apel la constructorul din clasa parinte, chiar si daca e vorba de un constructor default, prin super(). Asa e mai clar.
* Ordinea si modul in care se cheama constuctori



**Package**

* Un package este o multime de fisiere .java, in care se contin diferite clase. Fiecare package trebuie sa aiba clase ce au ceva comun intre ele. Daca nu specificam la inceputul fiecarui fisier .java carui package apartin clasele sale, atunci ele vor fi setate automat ca apartinand unui package default(nu in toate compilatoarele). Pentru a specifica carui package aparin toate clasele fisierului, se scrie la inceput de fisier:

**package nume;**

* Chiar daca clasele se afla in fisiere, ele defapt nu apartin acelor fisiere, ci package carui aparnit fisierele. De ex, fie ca am avem

**Pack1**

**File1 cu class File1, class File1\_2, class File1\_3**

**File2 cu class File2, class File2\_2**

Toate aceste clase sunt automat setate in Pack1, de aceea daca in File2 am incerca sa cream o clasa deja existenta in File1, nu am putea, caci clasa deja ar exista in package. Toate fisierele comunica intre ele prin package. Ele defapt doar permit sa grupam clasele, si atat, nu le detin.

**Package=Namespace**

* Cand importam o clasa dintr-un fisier din package, utilizam:

**import package.numeclasa;**

Nu trebuie sa specificam numele fisierului din care vine clasa!Doar numele la package.De exemplu fie:

**Package1**

**File1 cu class File1, File1\_1**

**File2 cu class File2, File2\_2**

Daca in File1, unde lucram, vom dori sa importam vreo clasa din File2, nu vom putea scrie:

**~~import Package1.File2.File2;~~**

ci

**import Package1.File2;**

Nu ar avea sens sa scriem si numele fisierului, asa cum toate clasele din toate fisierele se gasesc automat in acelasi package, de aia si se pune **package nume;** la inceput de fisier. Trebuie pus anume numele la package unde se gaseste fisierul!

* Un package poate contine mai multe package-uri in el, numite subpackage.

Pentru a defini clasele unui fisier intr-un subpackage folosim:

**package numepack.numesubpack;**

Pentru a importa o clasa dintr-u subpackage, folosim:

**import numepack.numesubpack.clasa;**

* O clasa publica poate fi importata din orice package! De exemplu, daca avem:

**Pack1**

**File1 cu public class File1, class File1\_1**

**Pack2**

**File2 cu public class File2, class File2\_2**

**File3 cu public class File3, class File3\_3**

**Pack4**

**File4 cu public class File4, class File4\_4**

* Chiar daca Pack2 face parte din Pack1, asta nu inseamna ca clasele din Pack2 fac parte si din Pack1. Pack1 are clasele sale, si Pack2 ale sale, separate.
* Clasele publice permit sa fie citite din orice package. De altfel, clasele nepublice nu pot fi citite din alt package! De exemplu, clasele File2\_2 si File3\_3 pot fi importate de catre orice fisier din Pack2, dar nu si din alte Packageuri, ceea ce inseamna ca Pack4 nu poate importa clasele File2\_2 si nici File3\_3. Nici File1 nu le poate importa. Clasele nepublice au nivelul de acces default.
* Nici File2,File3 sau File4 nu pot importa clasa File1\_1.
* In schimb, clasele publice File2, File1, File3 si File4 pot fi importate de orice package. **Poate exista o singura clasa publica per fisier si trebuie sa aiba numele identic cu al fisierului!** Asta din cauza ca atunci cand am fi intr-un fisier din alt package, si vom vrea sa importam o clasa din fisier, o vom putea importa doar pe cea publica, si respectiv cea ce are nume identic cu fisierul.Asa se restrictioneaza dreptul de a importa anumite clase si daca e necesar sa cautam fisierul unei clasel, vom sti ca numele lor sunt identice, si nu vom cauta prin toat fisierele.
* Daca in alt package am importa Pack1.\*; steluta ar importa strict clasele lui Pack1, nu si a subpack-urilor din el! Fiecare subpack trebuie importat separat!

**Poliformism**

* In Java, toate metodele sunt automat virtuale. De accea, daca avem de exemplu o clasa parinte si copil, ambele cu metoda test(), dar care are corp diferit, si daca avem:

**parinte var = new copil();**

**var.test();**

se va executa metoda test a clasei copil, caci in Java tote metodele sunt virtuale, si deci variabila var e de tip parinte, dar ea are o referinta catre un obiect de tip copil, deci anume cotinutul variabilei, adica obiectul de tip copil, va fi cel care va executa metoda, si nu se va lua dupa obiectul de tip parinte. **Metodele virtuale, cand sunt apelate, ele mereu vor cauta in fiecare bloc parinte, pana in bloc principal, daca nu cumva mai exista o metoda suprascrisa, si o vor executa anume pe ea**.In C++, daca nu se pune virtual in fata metodei, se ia metoda dupa tipul variabilei, nu a obiectului, adica metoda apelata nu mai cauta prin blocurile principale daca nu e cumva suprascrisa, ci se executa direct, dar daca se pune virtual, e ca in Java.

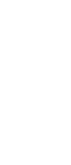
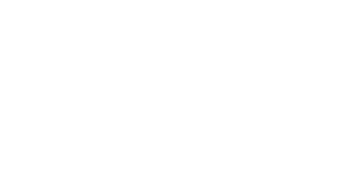
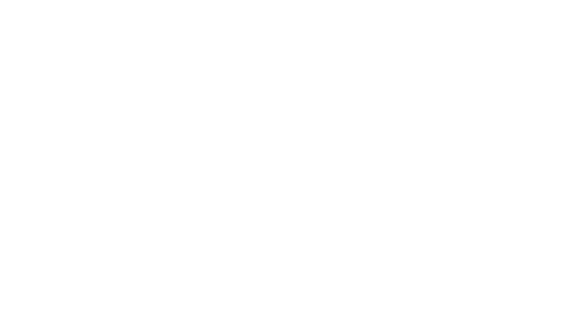
* Daca metoda test nu ar exista in clasa parinte, ci doar in cea copil, nu am putea sa o executam, caci metoda obiectului copil se leaga de corpul ei din bloc principal, astfel ca nu ar exista metoda din clasa parinte, ci copil. Daca clasa parinte nu ar avea asa metoda, nu ar avea cum sa fie gasita in blocul secundar.

**parinte var = new copil();**

**var.test();**







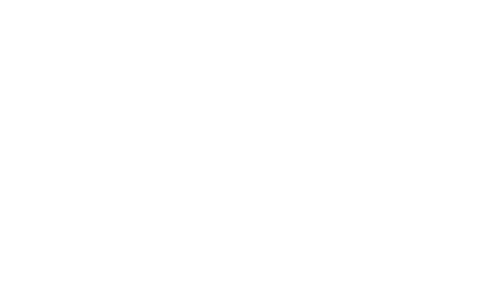
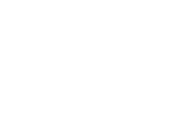
Metoda test din blocul secndar Parinte

nu pierde legatura cu

corpul! Doar lasa executia

celei din blocul principal.

Daca metoda test() nu ar exista in clasa parinte:



Nu exista deci nici metoda test in bloc secundar la care are acces, si deci nu are cum sa se execute.

* Daca de exemplu avem:

**class clasa1{**

void func1(){

**func2();**

}

void func2() {

}}

**class clasa2{**

void func1(){

super.func1();

}

**void func2() {**

}}

**class clasa3{**

void func1(){

**super.func1();**

}

void func2() {

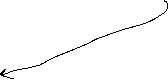
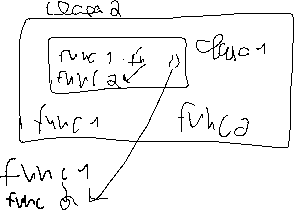
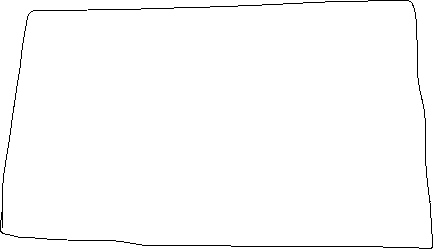
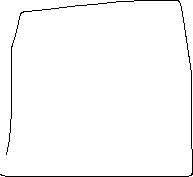
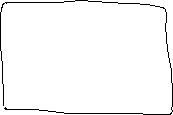
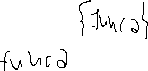
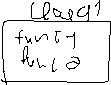
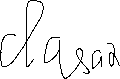
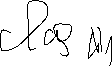
}}

Daca avem:

**clasa3 obj = new clasa3();**

**obj.func1();**

In acest caz, cand se va ajunge la func1 din class clasa1, se va executa func2 anume din clasa3, nu din clasa1!



Deci, orice metodata nu ar fi executata, ea mereu intai va fi cautata in blocul parinte al blocului din care e metoda, apoi la urmatorul bloc parinte si tot asa pana se ajunge la cel principal al obiectului. Asa cum metoda e virtuala automat, daca ea e executata din



bloc secundar, ea intai va cauta daca nu e suprascrisa in bloc parinte, si daca e, se executa din acel bloc, daca nu se cauta pana ajunge la cel principal, si daca nici acolo nu e, se executa ultima gasita. Daca nu e suprascrisa in bloc principal, atunci se executa din cel secundar.Doar super poate face ca metoda sa nu fie cautata in bloc principal! Asta e din cauza ca nu putem accesa o metoda din clasa parinte prin alta metoda tot din clasa parinte atat timp cat acea metoda ce vrem sa o accesam e suprascrisa deja in clasa copil! Metodele sunt doar suprascrise in clasa copil, si e logic ca orice apel la ele le-ar accesa anume din clasa copil, caci nu degeaba sunt suprascrise, exceptie facand cele chemate cu super. Anume asta si e rolul suprascrierei metodei, ca daca obiectul incearca sa acceseze o metoda suprascrisa,chiar si

din orice alta metoda din clasele parinte, mereu sa se acceze metoda din clasa sa, adica copil!

Asta nu inseamna ca metodele din bloc destinat clasei parinte pierd legatura cu obiectul metoda al lor. Ele pastreaza mereu legatura cu corpul lor original, doar ca metodat va verifica intai daca ea nu e suprascrisa in clasa copil, chiar si daca metoda e chemata din bloc secundar. Acest lucru se datoreaza faptului ca metodele sunt virtuale automat.

***La upcasting metoda din bloc secundar nu pierde legatura cu coprul ei! Ea tot e virtuala, si variabila de tip parinte, daca e egalata cu un obiect de tip copil, ea va avea o referinta anume la bloc secundar din obiect, si nu va putea nicidecum accesa bloc principal, dar metodele din bloc secundar mereu vor putea accesa bloc principal, si deci cand variabila de tip parinte va accesa metoda din namespace secundar, metoda oricum intai va cauta in bloc principal daca nu cumva e suprascrisa.***

Acest lucru nu este valabil si in cadrul atributelor!Atributele se acceseaza din bloc curent, nu principal, caci Java considera ca fiecare metoda lucreaza doar cu atributele din bloc unde se gaseste ea, de altfel nu am mai apela metode din clasele parinte.

**In C++, nu se face asa ceva. Acolo se executa direct metoda din blocul current, fie el si secundar. Doar daca metoda e declarata ca virtual, se va verifica daca nu e deja suprascrisa in clasa copil, si daca da, se va executa de acolo, daca nu, se va executa din clasa parinte.**