**Atributul static**

* Nu pot fi create in interiorul unei metode, caci trebuie sa fie create doar ca membrii de clasa, spre deosebire de C++
* Orice membru static poate fi accesat direct utilizand numele clasei, fara a mai crea obiecte.
* Orice clasa in Java are mereu un obiect propriu, doar pentru clasa, deci si un bloc.Orice membru static apartine clasei, si deci se gaseste nu in blocul fiecarui obiect creat,ci anume in blocul obiectului al clasei si fiecare obiect nou creat de acel tip de clasa, va avea o referinta la obiectu al clasei. Obiectele nu creaza copii ale membrilor statici, ca si in cazul celor nestatici! Blocul unei clase se creaza chiar la definirea ei,la runtime, si daca nu are membri statici, oricum se creaza, doar ca e gol si fiecare obiect are referinta la el.
* Metodele nestatice pot utiliza membri statici, dar metodele statice nu pot utiliza membri nestatici, caci ei nu apartin bloc-ului clasei, si metodele nu ar sti carui obiect apartin acei membri nestatici.
* Deobicei, atributele statice sunt si finale, desi nu mereu.

* ***Metodele statice nu sunt virtuale!!!***
* O clasa mosteneste **oarecum** metodele statice ale unei alte clase daca ea nu furnizeaza o metoda identica cu alt corp. Cand un obiect va accesa metoda statica, el o va cauta prin bloc, si de nu o gaseste, cauta in bloc la care are referinta, adica bloc la Classa.
* O clasa copil nu poate suprascrie o metoda statica a unei clase parinte!

Acest lucru are loc din cauza ca clasa copil are un bloc propriu cu referinta la blocul clasei parinte.**Blocul** **Clasei copil mereu va avea referinta la blocul clasei parinte**. Deci, metoda statica a clasei parinte se gaseste in blocul-ul ei. Cand clasa copil va crea o metoda statica identica cu cea din clasa parinte, ea va fi pusa in blocul clasei copil, ce va avea o referinta catre cel al clasei parinte! Cand se va incerca scrierea lui @Override, vom avea eroare, caci metoda statica,desi e indentica ca prototip, nu poate fi suprascrisa, asa cum in bloc al clasei copil nu se gaseste vreun bloc secundar cu acea metoda, ci doar o referinta catre alt bloc! A suprascrie inseamna a redefini comportamentul unei metode deja existente in blocul secundar, dar metoda statica a clasei parinte nu exista propriu zis in blocul secundar al clasei copil,nici nu exista bloc secundar pentru clasa parinte, ci exista doar o referinta catre blocul clasei parinte, dar referinta nu are nimic de a face cu metoda. Referinta se foloseste daca e necesar de a cauta o metoda sa fie executata, nu suprscrisa!



De asta metodele statice nici nu pot accesa membri nestatici. Orice obiect de tip copil va avea in bloc principal o referinta la bloc al clasei copil si in bloc secundar va avea o referinta la bloc al clasei parinte, si deci nu va **mosteni** propriu zis membrii statici, doar va avea referinta la acel bloc ce ii contine!

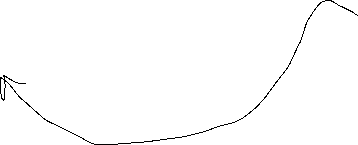
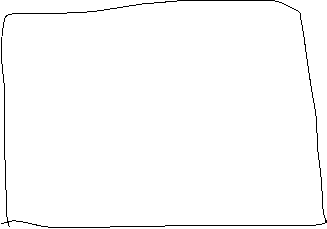
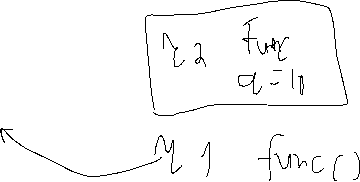
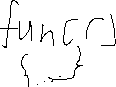
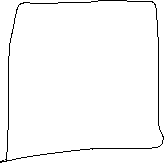
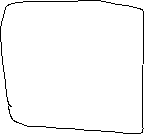
Fie:

class A{  
 static int *a* = 10;  
 public static void func(){  
 System.*out*.println("A");  
 }  
}  
class B extends A{  
@~~Override~~

public static void func(){  
 System.*out*.println("B");  
 }  
}



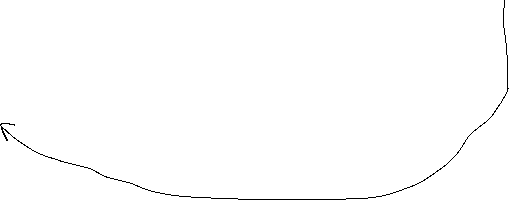
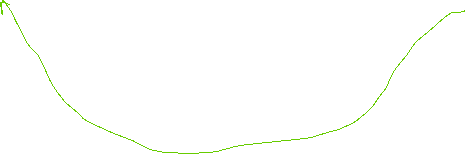
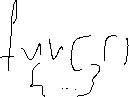
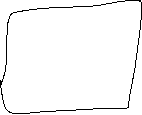
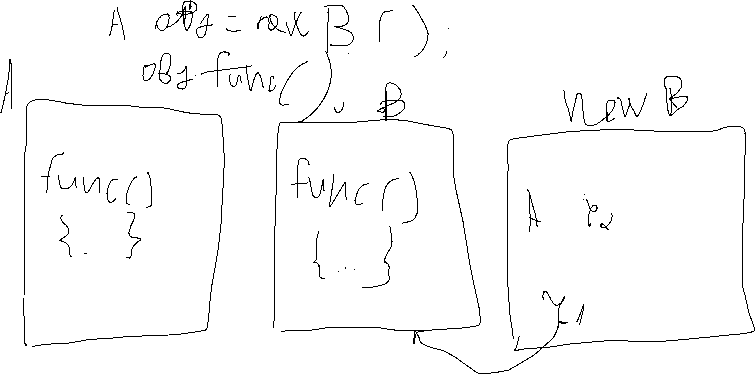
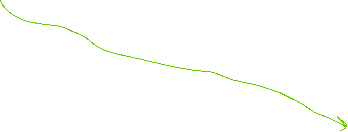
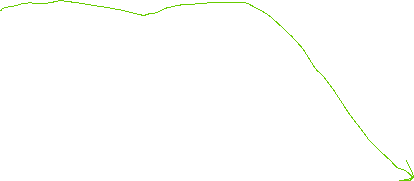
Presupunem ca vom avea un obiect de tip B



* **Anume din cauza ca clasa copil cu membrii statici are un bloc separat pentru ei, ce nu are legatura cu bloc ale obiectelor create si are doar o simpla referinta la clasa parinte,nu un bloc secundar, super nu functioneaza in metodele statice!.**
* Din cauza ca metodele statice se gasesc in blocuri separate neuinite intre ele, adica de clasa, e logic ca ele nu sunt virtuale, si cand se va accesa o metoda statica printr-o variabila de tip parinte ce are o referinta la un obiect de tip copil, se va executa metoda statica din clasa parinte! Acest lucru are loc din cauza ca variabila de tip parinte e legata de bloc secundar al obiectului, unde se gaseste si o referinta catre bloc al clasei parinte, si deci deoarece metoda nu se gaseste acolo,referinta va fi accesata ca sa se caute acolo metoda, si deoarece ea nu e virtuala si nici nu are sens sa fie, se va executa direct din bloc al clasei parinte. Ea nici nu ar avea cum sa caute daca nu cumva e suprascrisa, caci acel bloc al clasei nu are nici-o legatura cu bloc al obiectului.



public class Course{  
 public static void main(String[] args){  
 A obj = new B();  
 obj.*func*();  
 }  
}  
class A{  
 public static void func(){  
 System.*out*.println("A");  
 }  
}  
class B extends A{  
 public static void func(){  
 System.*out*.println("B");  
 }  
}

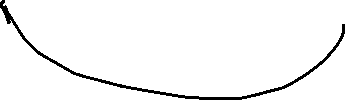
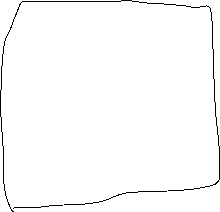
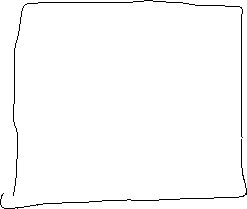
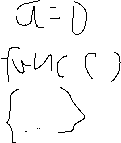


* **Datorita faptului ca bloc al clasei copil are o referinta la bloc al clasei parinte, putem accesa metoda statica a clasei parinte direct prin clasa copil. Daca metoda nu se gaseste in bloc al clasei copil, se cauta in bloc al clasei parinte, la care are referinta.**



public class Course{  
 public static void main(String[] args){  
 ++A.*a*;  
 A.*func*();  
 B.*func*();  
 }}  
class A{  
 static int *a* = 0;  
 static void func(){  
 System.*out*.println(*a*);  
 }  
}

class B extends A { }



**Deci, cand o clasa mosteneste o alta clasa, va avea in blocul ei o referinta la bloc al clasei parinte.**

**Interface**

* Interfatele nu au fost create pentru a permite mostenirea multipla.
* Interfatele sunt utilizate pentru a adauga functiuni speficie anumitor subclase ale unei superclase abstracte, pe care nu orice subclasa ar trebui sa le aiba, si deci nu pot fi definite in superclasa abstracta. Clasele abstracte contin functiuni pe care orice subclasa trebuie neaparat sa le aiba, dar interfetele au functiuni ce doar unele subclase trebuie sa le aiba.
* Atributele sunt automat **public,static,final**
* Metodele sunt automat ***public,abstract****.*
* Mod de mostenire:

**class clasa1 : extends parinte implements inter1, inter2….**

* O interfata poate si ea mosteni de la alte interfate, dar acest lucru nu se fae prin **implements**, ci **extends**!

**Interface inter1 extends inter2;**

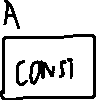
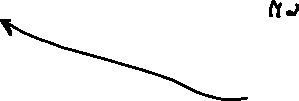
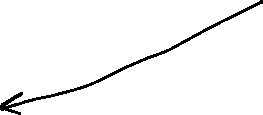
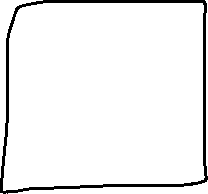
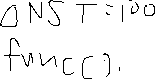
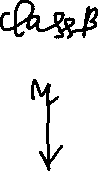
* **In cazul claselor, o metoda poate fi suprascrisa si poate sa-I fie modificat dreptul de acces numai cu conditia ca dreptul de acces e extins, nu restrans.**

class a{  
 ***public*** void func(){ }  
}  
class b extends a{  
 ~~protected~~ void func(){ }  
}

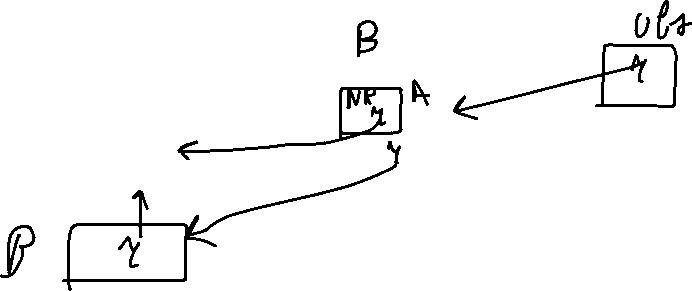
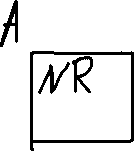
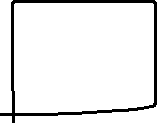
class a{  
 ***protected*** void func(){ }  
}  
class b extends a{  
 public void func(){ }  
}

**Dar interfatele au mereu totul public, deci putem declara o metoda redefinitia din interfata doar ca public!**

* **Daca o clasa implementeaza o interfata dar nu suprascrie toate metodele, ea trebuie declarata abstracta!**
* Interfatele nu pot fi instantiate! Totusi, ca si in cazul claselor abstracte,care tot nu pot fi instantiate, putem crea variabile de tip interfata care sa pastreze o referinta la un obiect a carui clasa implementeaza interfata data. E acelasi principiu ca si in cazul variabilelor de tip parinte ce sunt egalate cu obiecte de tip copil si deci pastreaza referinta catre bloc secundar ce se gaseste in bloc principal al obiectului de tip copil si-I corespunde, si deci variabila poate accesa doar ceea ce se stocheaza in bloc secundar, caci ea va avea referinta anume catre bloc secundar, nu principal, adica ce mosteneste, si desi metodele nu au corp in interfate, sunt si care pot avea(default) si ele oricum, ca si la clasele asbtracte, si orice clase, sunt virtuale automat si vor verifica intai daca nu cumva in bloc secundar sunt suprascrise ca sa execute metoda suprascrisa acolo, si ea va fi suprascrisa neaparat, fie ca e interfata sau clasa abstracta.
* Interfatele tot au obiect si deci bloc pentru atributele si metodele statice!!!**Deci cand o Interfata e definita, se creaza automat un obiectm deci bloc pentru ea si cand o clasa mosteneste aceasta ineterfata, in blocul ei va avea referinta catre bloc al interfetei.Ca si in cazul claselor, orice obiect de tip copil va avea in bloc principal o ref catre bloc al clasei sale si in bloc secundar o ref catre bloc al interfete.**
* class Course{  
   public static void main(String[] args){  
   A obj = new B();  
   obj.func();  
   System.*out*.println(obj.*CONST*);  
   }  
  }  
    
  interface A{  
   int *CONST* = 100;  
   void func();  
  }  
  class B implements A{  
   int CONST = 200;  
   public void func(){  
   System.*out*.println("hello");  
   }  
  }



public class test {  
 public static void main(String[] args){  
 System.*out*.println(B.*NR*);  
 B obj = new B();  
 System.*out*.println(obj.*NR*);  
 }  
}  
  
interface A{  
 int *NR* = 100;  
}  
class B implements A{  
}



* Relatia dintre o clasa si interfata pe care o implementeaza este **“se comporta ca un”**
* **Interfatele au si ele obiecte si deci blocuri ca si clasele cu membri statici, dar ideea e ca in acest bloc se gasesc doar metodele statice si atributele statice. Metodele abstracte e logic ca nu se gasesc, si nici cele default, caci ele doar vor fi copiate in bloc secundar al obiectului de tip clasa ce o va implementa.**

**Ambiguitati**

* Daca o clasa mosteneste o clasa abstracta si o interfata ce au un atribut identic sau mai multe interfete ce au la fel un atribut identic, si nu il va suprascrie(shadowing) va aparea o eroare daca vom incerca sa il accesam, caci compilatorul nu va sti pe care sa il acceseze.
* In schimb, chiar daca o clasa mosteneste o clasa abstracta si o interfata ce au aceeasi metoda abstracta sau mai multe interfate cu o metoda abstracta identica, nu va aparea nici-o ambiguitate, caci ele oricum vor fi suprascrise, si asa cum sunt virtuale, mereu se va cauta metoda suprascrisa in bloc principal al obiectului, deci la ele nici nu s-ar ajaunge.
* Referinta din blocul dedicat interfetei, din clasa, catre blocul interfetei nu e ascunsa! **Java pur si simplu nu permite accesarea metodelor statice din blocul interfetei direct din clasa, si atat.**

**Interface pana la Java 7**

* Contin doar constante, ce sunt automat:
* public
* static
* final

si respectiv, trebuie initializate chiar la creare, caci nu exista constructor.

* Contin doar metode abstracte, ce sunt automat:
* Abstract
* public

Toate aceste lucruri automate nu trebuie scrise, caci Java le adauga automat, dar putem sa le si scriem.

**Interface dupa java 8**

* Contin si metode statice
* Contin si metode default
* Metodele default se mostenesc si pot fi suprascrise, ele se comporta ca cele obisnuite, pur si simplu cuvantul default indica ca ea nu va fi automat abstracta, dar oricum mereu e automat publica! Metodele default nu fac pate din bloc al Interfatei!Ele sunt chiar copiate in bloc secundar, dedicat interfatei in clasa ce mosteneste, si deci nu pot fi accesate din Interfata.
* Daca o interfata mosteneste o alta interfata, ea poate suprascrie metodele abstracte din ea prin redefinirea metodei ca default, si deci nici clasa ce le implementeaza nu trebuie sa o mai suprascrie

interface A{  
 void func();  
}  
interface A1 extends A{  
 @Override  
 default void func(){  
 System.*out*.println("Test");  
 }  
}  
class B implements A, A1{ }

* Daca o interfata are o metoda default si o clasa ce implementeaza interfata suprascrie metoda default, e evident ca orice obiect va accesa metoda suprascrisa in clasa, dar daca dorim sa accesam metoda default din interfata, putem sa utilizam **Interfata.super.metoda()**

Nu putem utiliza doar super, ca in cazul mostenirii unei clasei, ci trebuie sa specificam si numele Interfetei, caci am putea mosteni mai multe interfete si nu ar fi clar la care din ele se refera super. In plus, super ar face trimitire la clasa Object pe care o mosteneste orice clasa create de noi. Mai mult ca atat, daca nu am utiliza super, ci doar Interfata.metoda(), s-ar crea impresia ca metoda() e statica, nu default.

interface A1 {  
 default void func(){  
 System.*out*.println("Test");  
 }  
 static void func2(){  
 System.*out*.println("Statica");  
 }  
}  
class B implements A1{  
 public void test(){  
 A1.super.func();  
 A1.*func2*();  
 }  
}

* Daca o clasa mosteneste doua interfete sau mai multe ce au aceeasi metoda default, adica identica dupa nume, **ea trebuie neaparat** sa o suprascrie,ca o foloseste sau nu, ca sa evite ambiguitatea.
* interface a{  
   default void func(){  
   }  
  }  
  interface b{  
   default void func(){  
   }  
  }  
  class c implements a,b{ }
* Daca gen s-r mosteni si o clasa ce sa aiba tot metoda func(), nu ar mai fi

ambiguitate, asa cum orice referire la func() ar fi aleasa din clasa parinte

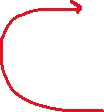
interface i1{  
 default void func() { }  
}  
interface i2{   
 default void func() {}  
}  
class clasa1{  
 public void func() {}  
}  
class clasa2 extends clasa1 implements i1,i2{  
   
}



**Same methods no conflict**

* Daca o clasa mosteneste o clasa si o interfata ce au aceeasi metoda, si interfata deci are metoda data ca default, desi ar parea ca ar trebui sa fie un conflict cand clasa copil va incerca sa acceseze metoda care e implementata si in clasa parinte si interfata, nu apare nici-un conflict.

interface int1{  
 int *a* = 100;  
 default void func(){  
 System.*out*.println("Interfata");  
 }  
}  
class clasa1{  
 int a = 100;  
 public void func(){  
 System.*out*.println("Clasa");  
 }  
}  
  
class clasa2 extends clasa1 implements int1{  
 public void met(){  
 func();  
 }  
}



metodele nu se comporta asa ca atributele. Daca am fi incercat gen sa accesam acelasi atribut, ca aici:

interface int1{  
 int *a* = 100;  
  
}  
class clasa1{  
 int a = 100;  
  
}  
  
class clasa2 extends clasa1 implements int1{  
 public void met(){  
 System.*out*.println(a);  
 }  
}



atunci e ambiguitate, dar la metode chestiile stau putin altfel

* ***Metoda clasei parinte mereu are prioritate asupra metodei default din interfata, mereu!!!*De aceea, daca compilatorul are de ales intre o metoda din clasa parinte si una default din interfata(intefete), mereu o va alege pe cea din clasa.**



**Metode Statice in Interface**

* Metodele statice din interface nu sunt mostenite. Interfatele au un obiect si deci bloc propriu, si orice clasa ce le implementeaza are o referinta la bloc a lor. Totusi, ideea e ca daca 2 interfete au aceeasi metoda default, si o clasa le mosteneste, ambiguitatea poate fi rezolvata prin suprascrierea metodei, insa daca ele ar avea aceeasi metodata statica, sau daca ar mosteni o clasa si o interfata cu aceeasi metodata statica, iar ar fi ambiguitate caci metoda va putea fi accesata prin referinta catre bloc din clasa, dar s-ar putea accesa si prin bloc si din interfata. Si nici nu o putem rezolva, caci nu putem suprascrie o metoda statica, iata de ce, Java are un system special ce nu permite metodelor statice din Interface sa fie accesate prin referinta ce o au clasele catre Interfata, dar doar celor din clase. Asta nu se aplica si asupra la atribute,
* import ne ofera in cazul metodelor statice din interfate, clase simple sau abstracte, posibilitatea de a importa direct doar cate o metoda statica. Acest lucru se face astfel:

**import static pack.clasa/interf.metoda;**