**Polimorfism**

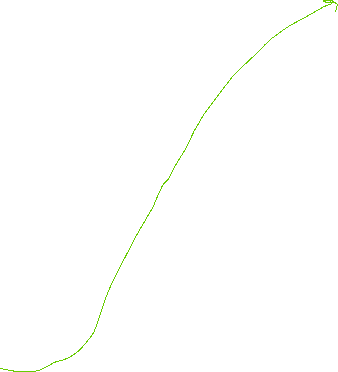
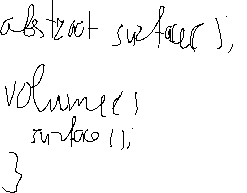
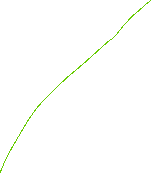
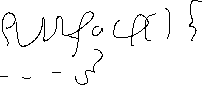
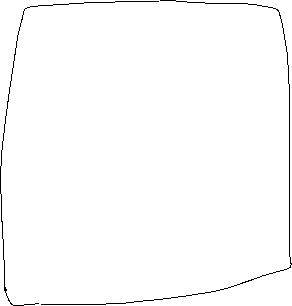
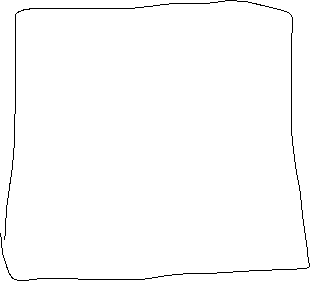
Reprezinta proprietatea subclaselor de a suprascrie metode ce sunt prezente in supraclase, astfel incat orice instanta a subclaselor care ar face apel la acea metoda, sa o execute pe cea din subclasa.

**Metode abstracte**

* Metodele abstracte mereu apartin unei clase abstracte.Le utilizam cand nu putem crea nici macar aproximativ un model de metoda pentru toate subclasele, ci ele le vor crea aparte sau cand e obligatoriu ca toate subclasele sa aiba o asa metoda.
* Orice subclasa ce extinde o superclasa abstracta, trebuie neaparat sa furnizeze un corp pentru toate metodele abstracte. In caz contrar, daca nu declaram si subclasa ca abstracta, vom avea eroare.
* Putem sa utilizam in cadrul metodelor neabstracte din clasele abstracte, metode ce au fost definite ca abstracte. De exemplu:

abstract class class1{  
 int l = 1;  
 int L = 2;  
 int h = 3;  
 abstract double surface();  
 double volume(){  
 return h\*surface();  
 }  
}

chiar daca metoda surface() nu are corp definit, ea poate fi apelata in volume(), caci clasa class1 oricum nu poate avea instantieri,adica obiecte, si deci nu putem oricum chema metoda ei abstracta surface(), direct. In schimb, cand superclasa va fi mostenita de o subclasa, subclasa va trebui sa suprascrie metoda abstracta surface(), si deoarece toate metodele sunt si virtuale automat, cand volume() va chema metoda surface, compilatorul va verifica, ca in cazul a orice clasa care mosteneste ceva, daca metoda surface() nu cumva se gaseste deja in bloc parinte(aici principal), si deoarece se gaseste, se va executa anume metoda cu corp din bloc principal. Deci, metoda surface fara corp nici nu va fi atinsa de compilator.Teoretic, cand se declara clasa abstracta, metoda volume() cand apeleaza clasa surface face referire la metoda abstracta surface, dar stiind ca ea oricum va fi suprascrisa mai apoi.



Deci, de ex:



cl2 obj = new cl2();

obj.volume();

***Nici la upcasting metoda din bloc secundar nu pierde legatura cu coprul ei!***

**Atributele sunt executate din bloc la metoda!**

* Metodele abstracte ar putea fi evitate prin crearea unei metode in clasa parinte ce nu va face nimic, si deci in caz ca o subclasa nu o va suprascrie, se va utiliza metoda ce nu face nimic, dar nu e prea comod asa, si poate avea rezultate neasteptate. In plus, nu obliga nici de cum clasele copii sa o suprascrie.
* **O metoda abstracta nu poate fi private!**
* O clasa ce mosteneste o clasa abstracta si nu suprascrie toate metodele abstracte, trebuie si ea declarata ca abstracta.
* **Nu putem instantia clasele abstracte, dar putem crea variabile de tipul lor si tablouri.Fie clasa Figura abstracta. Putem face asa ceva:**

**Figura[] obj = new Figura[10];**

**Mostenire si poliformism**

* Orice clasa mosteneste automat clasa Object
* O variabila de tip Object poate avea o referinta la absolut orice obiect
* Clasa Object are metodele:
* toString()
* equals()
* clone()
* **Suprascriere–** redefinirea unei metode identice(dupa nr de parametri,tipul lor) din clasa parinte in clasa copil si cu tip de return compatibil in ierarhie.

**Nu putem suprascrie metode ce se diferentiaza doar dupa tipul de return, numai daca metodele returneaza un obiect de tip clasa compatibil in ierarhie.** Nu e valabil si la parametri. De ex:

class class1{  
 class1 func(){  
 return new class1();  
 }  
}  
class class2 extends class1{

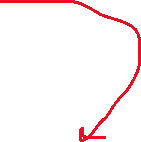
@Override  
 class2 func(){  
 return new class2();  
 }  
}

* **Supraincarcare –** crearea unei noi metode cu acelasi nume deja existenta, dar cu noi parametri sau cu acelasi nr de parametri dar de diferit tip.

**dar nu e posibil:**

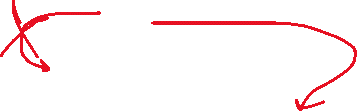
class class1{  
 int func(){  
 return 100;  
 }  
}  
class class2 extends class1{  
 double func(){  
 return 100.5;  
 }  
}

* **Daca supraincarcam metode ce se diferentiaza doar print tipul parametrului formal, si sa zicem ca avem de ex clasa1 si clasa2 si clasa2 mosteneste de la clasa1, daca vom crea 2 metode ce vor diferentia doar prin faptul ca metoda1 va avea un parametru de tip clasa1 si a doua metode de tip clasa2, la executarea metodei cu un param actual de tip clasa2, se va executa metoda cu param de tip clasa2**
* **Chiar daca exista mai multe metode supraincarcate care ar putea sa se ocupe de obiectul ca parametru actual, mereu se va alege cea care are exact tipul lui, sau alta cu un parametru de tip parinte mai apropiat in ierarhie:**
* public class inter {  
   public static void main(String[] args) throws Exception {  
   *func*(new clasa2());  
    
   }  
   public static void func(int a){  
   System.*out*.println("int");  
   }  
   public static void func(clasa1 a){  
   System.*out*.println("double");  
   }  
   public static void func(clasa2 a){  
   System.*out*.println("long");  
   }  
  }  
    
  class clasa1{  
    
  }  
  class clasa2 extends clasa1{  
    
   }



}

public class inter {  
 public static void main(String[] args) throws Exception {  
 *func*(5);  
  
 }  
 public static void func(double a){  
 System.*out*.println("double");  
 }  
 public static void func(int a){  
 System.*out*.println("int");  
 }  
  
}



**Modificatorul Final**

Se aplica pentru:

* **Variabibile** – ca valorile lor, odata definite, sa nu mai poata fi modificate. Nu e nevoie de atribuit o valaore unei variabile finale chiar la crearea ei ca in C++, se poate si dupa, insa odata atribuita, nu poate fi modificata.
* **Clase** – ca sa nu poata fi mostenita
* **Metode** – pentru a impieda suprascrierea
* **Attribute** – ca si variabilele, se poate seta valoarea lor odata, insa ea poate fi setata fie la momentul definirii ei in clasa, fie doar in constructor! Nu putem sa o setam dintr-o metoda! Ea nu e setata automat de constructori ca ceilalti membri nefinali.
* class class1{  
   final int var;  
    
  }

ERROR! Nu exista constructor ce sa seteze valoarea variabilei si ea nu e setata nici la declararea ei.

* Daca dorim sa facem un downcating si sa accesam si metoda, facem asa:

((tip)obj).metoda();

**Constructori**

* In Java, un constructor catre superclasa nu poate trimite membrii nestatici. De exemplu:

class class1{  
 int a = 1;  
 class1(int a){  
 this.a = a;  
 }  
}  
class class2 extends class1{  
 int b = 10;  
  
 class2(int a){  
 ~~super(b);~~  
 }  
}

Acest program va da o eroare, din cauza ca incercam sa trimitem constructorului superclasei un membru care inca nici nu exista.

Constructorul este cel care creaza obiectul in memorie, insa pana constructorul clasei parinte, cel chemat prin super() nu termina executia, nu se va crea inca obiectul propriu zis.

Cand se apeleaza super(b); obiectul de tip class2 inca nu exista! Abia se creaza bloc secundar pentru clasa parinte, si deci pana constructorul clasei parinte nu termina sa se execute, obiectul de tip class2 inca nu va avea setati membrii in blocul principal, caci se lucreaza inca in bloc secundar. Dupa ce se executa blocul clasei parinte, se creaza direct propriu zis obiectul de tip class2, si se trece astfel la bloc principal, unde deodata dupa executia lui super() se creaza toti membrii obiectului, iar mai apoi se executa instructiunile din constructor.

Super(b) incearca sa trimita un membru b inca necreat in obiect, dar nici obiectul defapt nu e creat.

De asemenea, daca se cheama intai un alt constructor prin this(), se intampla cam acelasi lucru. Constructorul original nu va fi cel care se va ocupa de crearea obiectului si a membrilor, ci cel chemat prin this(). Acest constructor apelat din cel principal se va ocupa ca prima instructiune de crearea obiectului si a membrilor. Dupa ce se executa this() si se revine la constructorul principal, obiectul cu tot cu membrii sai e deja creat.

Constructorul ce a fost apelat la crearea obiectului se ocupa de crearea lui si de initializarea tuturor membrilor sai. Acesta e primul lucru pe care il face constructorul, chiar si inainte de a executa vreo instructiune din el. Daca exista vreun super(), se vor executa intai constructorul din clasa parinte, si dupa se va crea obiectul si membrii lui! Daca exista un this(), atunci deja acest al doilea constructor apelat se va ocupa ca prima instrutiune de crearea obiectului cu tot cu membrii lui.

* constructor(int a){

# se creaza obiectul si membrii intai de toate

this.a = a;

}

* constructor(int a){

this()/super()

# se creaza obiectul si membrii

this.a = a;

}

* Fie ca avem clasa1 obj = new clasa1(10);

class clasa1{

int a;

clasa1(int a) {

**this();**

this.a = a;

}

clasa1(){

# se creaza obiectul si membrii

System.out.printn(a); #a = 0

}