Curs 2
Obiecte si clase in Java

Ciclul de viata al unui obiect

Crearea obiectelor

In Java obiectele sunt create prin instantierea unei clase, cu alte cuvinte prin crearea unei instante a unei clase.Crearea unui obiect presupune trei lucruri:

1. **Declararea obiectului**
2. `NumeClasa numeObiect;`
3. `Ex: Rectangle patrat;`
4. **Instantierea**
   Se realizeaza prin intermediul operatorului **`new`** si are ca efect crearea efectiva a obiectului cu alocarea spatiului de memorie corespunzator.
5. `patrat = new Rectangle();`
6. **Initializarea**
   Se realizeaza prin intermediul constructorilor clasei respective. `Rectangle()` este un apel catre constructorul clasei `Rectangle` care este responsabil cu initializarea obiectului. Initializarea se poate face si cu anumiti parametri, cu conditia sa existe un constructor al clasei respective care sa accepte parametrii respectivi;
7. `patrat = new Rectangle(0, 0, 100, 200);`

   Fiecare clasa are un set de constructori care se ocupa cu initializare obiectelor nou create. De exemplu clasa `Rectangle` are urmatorii constructori:

```
public Rectangle(Point p)
public Rectangle(int w, int h)
public Rectangle(Point p, int w, int h)
public Rectangle()
```

(vezi "Constructorii unei clase")

Declararea, instantierea si initializarea obiectului pot aparea pe aceesi linie (cazul cel mai uzual):

```
Rectangle patrat = new Rectangle(0, 0, 100, 200);
```

Este posibila si crearea unor obiecte anonime care servesc doar pentru initializarea altor obiecte, caz în care etapa de declarare a obiectului nu mai este prezenta:

```
patrat.origin = new Point(0, 0);
//se creeaza un obiect anonim de tip Point care,
//dupa executarea atribuirii este automat distrus de catre sistem
```

Atentie

Spatiul de memorie nu este pre-alocat

Declararea unui obiect nu implica alocarea de spatiu de memorie pentru acel obiect.

```
Rectangle patrat;
patrat.x = 10; //EROARE!
```

Alocarea memoriei se face doar la apelul instructiunii new !

Folosirea obiectelor

Odata un obiect creat, el poate fi folosit în urmatoarele sensuri: aflarea unor informatii despre obiect, schimbarea starii sale sau executarea unor actiuni. Aceste lucruri se realizeaza prin:

- aflarea sau schimbarea valorilor variabilelor sale
- apelarea metodelor sale.

Referirea valorii unei variabile se face prin **`obiect.variabila`**
De exemplu clasa `Rectangle` are variabilele publice `x, y, width, height, origin`. Aflarea valorilor acestor variabile sau schimbarea lor se face prin constructii de genul:

```
Rectangle patrat = new Rectangle(0, 0, 100, 200);
System.out.println(patrat.width);     //afiseaza 100
patrat.x = 10;
patrat.y = 20;                        //schimba originea
patrat.origin = new Point(10, 20);    //schimba originea
```

Obs: Accesul la variabilele unui obiect se face în conformitate cu drepturile de acces pe care le ofera variabilele respective celorlalte clase. (vezi "Specificatori de acces pentru membrii unei clase")

Apelul unei metode se face prin **`obiect.metoda([parametri])`**

```
Rectangle patrat = new Rectangle(0, 0, 100, 200);
patrat.setLocation(10, 20);    //schimba originea
patrat.setSize(200, 300);      //schimba dimensiunea
```

Se observa ca valorile variabilelor pot fi modificate indirect prin intermediul metodelor.
Programarea orientata obiect descurajeaza folosirea directa a variabilelor unui obiect deoarece acesta poate fi adus în stari inconsistente (ireale). In schimb, pentru fiecare variabila a obiectului trebui sa existe metode care sa permita aflarea/schimbarea valorilor variabilelor sale.

```
patrat.width = -100;   //stare inconsistenta
patrat.setSize(-100, -200)
//metoda setSize poate sa testeze daca valorile sunt corecte si sa
//valideze sau nu schimbarea lor
```

Distrugerea obiectelor

Multe limbaje de programare impun ca programatorul sa tina evidenta obiectelor create si sa le distruga în mod explicit atunci când nu mai este nevoie de ele, cu alte cuvinte sa administreze singur memoria ocupata de obiectele sale. Practica a demonstart ca aceasta tehnica este una din principala furnizoare de erori ce duc la functionarea defectuoasa a programelor.
In Java programatorul nu mai are responsabilitatea distrugerii obiectelor sale întrucât, în momentul

rularii unui program, simultan cu interpretorul Java ruleaza si un proces care se ocupa cu distrugerea obiectelor care nu mai sunt folosite. Acest proces pus la dispozitie de platforma Java de lucru se numeste **garbage collector** (colector de gunoi), prescurtat **gc**.
Un obiect este elimnat din memorie de procesul de colectare atunci când nu mai exista nici o referinta la acesta. Referintele (care sun de fapt variabile) sunt distruse în mod :

- *natural*, atunci când variabila respectiva iese din domeniul sau, de exemplu la terminarea unei metode
- *explicit*, daca atribuim variabilei respective valoare `null`.

Cum functioneaza colectorul de gunoaie ?

GC este un proces de prioritate scazuta care se executa periodic si care scaneaza dinamic memoria ocupata de programul Java aflat în executie si marcheaza acele obiecte care au referinte directe sau indirecte. Dupa ce toate obiectele au fost parcurse cele care au ramas nemarcate sunt eliminate din memorie.
Procesul gc poate fi fortat sa se execute prin metoda `gc` a clasei `System`.

Finalizare

Inainte ca un obiect sa fie eliminat din memorie, procesul gc da acelui obiect posibilitatea "sa curete dupa el", apelând metoda de finalizare a obiectului respectiv. Uzual, în timpul finalizarii un obiect îsi închide fisierele si socket-urile folosite, distruge referintele catre alte obiecte (pentru a usura sarcina colectorului de gunoaie), etc. Codul pentru finalizare unui obiect trebuie scris într-o metoda speciala numita `finalize` a clasei ce descrie obiectul respectiv.
De exemplu, clasa `Rectangle` are urmatorul cod pentru finalizarea obiectelor sale:

```
class Rectangle {
...
        protected void finalize() throws Throwable {
            origin = null;
            super.finalize();
        }
}
```

Crearea claselor

Declararea claselor

Declararea unei clase

[public][abstract][final] **class NumeClasa**
[extends NumeSuperclasa]
[implements Interfata1 [, Interfata2 ...]]
{
//corpul clasei
}

Asadar, prima parte a declaratie o ocupa modificatorii clasei. Acestia sunt:

Modificatorii unei clase	
public	Implicit, o clasa poate fi folosita doar de clasele aflate în acelasi pachet cu clasa respectiva (daca nu se specifica un anume pachet, toate clasele din directorul curent sunt

	considerate a fi în acelasi pachet). O class declarata cu `public` poate fi folosita de orice clasa, indiferent de pachetul în care se gaseste.
`abstract`	Declara o clasa abstracta (sablon). O clasa abstracta nu poate fi instantiata, fiind folosita doar pentru a crea un model comun pentru o serie de subclase; (vezi "Clase si metode abstracte")
`final`	Declara ca respectiva clasa nu poate avea subclase. Declarare claselor finale are doua scopuri: • *securitate* : unele metode pot astepta ca parametru un obiect al unei anumite clase si nu al unei subclasei, dar tipul exact al unui obiect nu poate fi aflat cu exactitate decât în momentul executiei; în felul acesta nu s-ar mai putea realiza obiectivul limbajului Java ca un program care a trecut compilarea nu mai este susceptibil de nici o eroare (nu "crapa sistemul"). • *programare în spririt orientat-obiect* : "O clasa perfecta nu trebuie sa mai aiba subclase"

Dupa numele clasei putem specifica, daca este cazul, faptul ca respectiva clasa este subclasa a unei alte clase cu numele *NumeSuperclasa* sau/si ca implementeaza una sau mai multe interfete, ale caror nume trebuie separate prin virgula.
Se observa ca, spre deosebire de C++, Java permite doar **mostenirea simpla**, asadar o clasa poate avea un singur un singur parinte (superclasa). Evident o clasa poate avea oricâti mostenitori (subclase). Extinderea unei clase se realizeaza deci astfel:

```
class B extends A {...}        //A este superclasa clasei B
```

Corpul unei clase

Urmeaza declararea clasei si este cuprins între acolade. Contine:

- declararea variabilelor de instanta si de clasa (cunoscute împreuna ca variabile membre)
- declararea si implementarea metodelor de instanta si de clasa (cunoscute împreuna ca metode membre)

Spre deosebire de C++, nu este permisa doar declararea metodei în corpul clasei, urmând ca implementare sa fie facuta în afara ei. Implementarea metodelor unei clase trebuie sa se faca "inline" in corpul clasei.

In C++

```
class A {
        void metoda1();
        int  metoda2() {
                //implementare
        }
}
A::metoda1() {
//implementare
}
```

In Java

```
class A {
        void metoda(){
                //implementare
        }
}
```

Obs: variabilele unei clase pot avea acelasi nume cu metodele clasei.

Constructorii unei clase

Constructorii unei clase sunt metode speciale care au acelasi nume cu cel al clasei, nu returneaza nici o valoare si sunt folositi pentru initializarea obiectelor acelei clase în momentul instantierii lor.

```
class Dreptunghi {
        Dreptunghi() {
                //constructor
        }
}
```

O clasa poate avea unul sau mai multi constructori care trebuie însa sa difere prin lista de argumente primite. In felul acesta sunt permise diverse tipuri de initializari ale obiectului la crearea sa, în functie de numarul parametrilor cu care este apelat constructorul.

```
class Dreptunghi {
        double x, y, w, h;
        Dreptunghi(double x0, double y0, double wo, double h0) {
                x=x0; y=y0; w=w0; h=h0;
        }
        Dreptunghi(double x0, double y0) {
                this(x0, y0, 0, 0);
        }
        Dreptunghi() {
                //initializare implicita
                this(0, 0, 100, 100);
        }
}
```

Constructorii sunt apelati automat la instantierea unui obiect. In cazul în care dorim sa apelam explicit constructorul unei clase folosim metoda **`this( argumente )`**, care apeleaza constructorul corespunzator (ca argumente) al clasei respective. Aceasta metoda este folosita atunci când sunt implementati mai multi constructori pentru o clasa pentru a nu repeta secventele de cod scrise la constructorii cu mai putine argumente.

Dintr-o subclasa putem apela si constructorii superclasei cu metoda **`super( argumente )`**.

```
class Patrat extends Dreptunghi {
        double size;
        Patrat(double x0, double y0, double s0) {
                super(x0, y0, s0, s0); //se apeleaza constructorul
superclasei
                size = s0;
        }
}
```

Constructorii sunt apelati automat la instantierea unui obiect.

In cazul în care scrieti o clasa care nu are declarat nici un constructor, sistemul îi creeaza automat un constructor implicit care nu primeste nici un argument si care nu face nimic. Deci prezenta constructorilor în corpul unei clase nu este obligatorie. Daca însa ati scris un constructor pentru o clasa care are mai mult de un argument, atunci constructorul implicit (fara nici un argument) nu va mai fi furnizat implicit de catre sistem.

```
class Dreptunghi {
        //nici un constructor
}
...
Dreptunghi d;
d=new Dreptunghi(); -> OK (a fost generat constructorul implicit)
```

```
class Dreptunghi {
        double x, y, w, h;
        Dreptunghi(double x0, double y0, double wo, double h0) {
                x=x0; y=y0; w=w0; h=h0;
        }
```

```
}
...
Dreptunghi d;
d=new Dreptunghi(); -> Eroare la compilare
```

Constructorii unei clase pot avea urmatorii specificatori de acces:

Specificatorii de acces ai constructorilor	
`private`	Nici o alta clasa nu poate instantia obiecte ale acestei clase. O astfel de clasa poate contine metode publice (numite "factory methods") care sa-si creeze propriile obiecte si sa le returneze altor clase, controlând în felul acesta diversi parametri legati de creare obiectelor sale.
`protected`	Doar subclasele pot crea obiecte de tipul clasei respective.
`public`	Orice clasa poate crea instante ale clasei respective
`implicit`	Doar clasele din acelasi pachet pot crea instante ale clasei respective

Declararea variabilelor membre

Variabilele se delara de obicei înaintea metodelor, desi acest lucru nu este impus de compilator.

```
class NumeClasa {
        //declararea variabilelor
        //declararea metodelor
}
```

Variabilele unei clase se declara în corpul clasei dar nu în corpul unei metode. Variabilele declarate în cadrul unei metode sunt locale metodei respective.
Declararea unei variabile presupune specificarea urmatoarelor lucruri:

- numele variabilei
- tipul de date
- nivelul de acces la acea variabila de catre alte clase
- daca este constanta sau nu
- daca este variabila de instanta sau de clasa

Generic, o variabila se declara astfel:

Declararea variabilelor membre
`[modificatori] TipDeDate numeVariabila [ = valoareInitiala ] ;`

unde un modificator poate fi :

- un specificator de acces : `public, protected, private`
  (vezi "Specificatori de acces pentru membrii unei clase")
- unul din cuvintele rezervate: `static, final, transient, volatile`

```
Ex:     double x;
        protected static int n;
        public String s = "abcd";
        private Point p = new Point(10, 10);
        final long MAX = 100000L;
```

`static` Declararea variabilelor de	Prezenta lui declara ca o variabila este variabila de clasa si nu de instanta:
	`int x ; //variabila de instanta`
	`static int nrDeObiecteCreate; //variabila de clasa`
	(vezi "Membri de instanta si membri de clasa")

clasa	

final
Declararea constantelor
	Indica faptul ca valoarea variabilei nu mai poate fi schimbata, cu alte cuvinte este folosit pentru declararea constantelor.
	`final double PI = 3.14 ;`
	`...`
	`PI = 3.141 -> eroare la compilare`
	Prin conventie numele variabilelor finale se scriu cu litere mari. Folosirea lui `final` aduce o flexibilitate sporita în lucrul cu constante, în sensul ca valoarea unei variabile nu trebuie specificata neaparat la declararea ei (ca în exemplul de mai sus), ci poate fi specificata si ulterior, dupa care ea nu va mai putea fi modificata.
	`class Test {`
	`  final int MAX;`
	`  Test() {`
	`  MAX = 100;  // legal`
	`  MAX = 200;  // ilegal -> eroare la compilare`
	`  }`
	`}`

transient	Este folosit la serializarea obiectelor, pentru a specifica ce variabile membre ale unui obiect nu participa la serializare (vezi "Serializarea obiectelor")
volatile	Este folosit pentru a semnala compilatorului sa nu execute anumite optimizari asupra membrilor unei clase. Este o facilitate avansata a limbajului Java.

Implementarea metodelor

Metodele sunt responsabile cu descrierea comportamentului unui obiect. Generic, o metoda se declara astfel:

Declararea metodelor membre

```
[modificatori] TipReturnat numeMetoda ( [argumente] )
        [throws TipExceptie]
{
        //corpul metodei
}
```

unde un modificator poate fi :

- un specificator de acces : `public, protected, private`
  (vezi "Specificatori de acces pentru membrii unei clase")
- unul din cuvintele rezervate: `static, abstract, final, native, synchronized`

static
Declararea metodelor de clasa
	Prezenta lui declara ca o metoda este metoda de clasa si nu de instanta:
	`void metoda1() ;  //metoda de instanta`
	`static void metoda2(); //metoda de clasa`
	(vezi "Membri de instanta si membri de clasa")

abstract Declararea metodelor abstracte	O metoda abstracta este o metoda care nu are implementare si trebuie sa faca parte dintr-o clasa abstracta.(vezi "Clase si metode abstracte")
final	Specifica faptul ca acea metoda nu mai poate fi supradefinita în subclasele clasei în care ea este definita ca fiind finala. (vezi "Metode finale")

native	In cazul în care aveti o librarie însemnata de functii scrise în alt limaj de programare decât Java (C de exemplu), acestea pot fi refolosite din programele Java.
synchronized	Este folosit în cazul în care se lucreaza cu mai multe fire de executie iar metoda respectiva se ocupa cu partajarea unei resurse comune. Are ca efect construirea unui semafor care nu permite executarea metodei la un moment dat decât unui singur fir de executie. (vezi "Fire de executie")

Tipul returnat de o metoda

Metodele pot sau nu sa returneze o valoare (un obiect) la terminarea lor. Tipul returnat poate fi atât un tip primitiv de date (int, double, etc.) sau o referinta la un obiect al unei clase. In cazul în care o metoda nu returneaza nimic atunci la rubrica *TipReturnat* trebuie obligatoriu sa apara cuvântul cheie **void**. `(Ex: void afisareRezultat() )`
Daca o metoda trebuie sa returneze o valoare acest lucru se realizeaza prin intermediul instructiunii **return**, care trebuie sa apara în toate situatiile de terminare a functiei.

```
double radical(double x) {
        if (radical >= 0)
                return Math.sqrt(x);
        else
                System.out.println("Eroare - numar negativ!");
                //Eroare la compilare - lipseste return pe aceasta ramura
}
```

In cazul în care în declaratia functiei tipul returnat este un tip primitiv de date, valoarea returnata la terminarea functiei trebuie sa aiba obligatoriu acel tip, altfel va fi furnizata o eroare la compilare. Daca valoarea returnata este o referinta la un obiect al unei clase, atunci clasa obiectului returnat rebuie sa coincida sau sa fie o subclasa a clasei specificate la declararea metodei. De exemplu, fie clasa `Poligon` si subclasa acesteia `Patrat`.

```
Poligon metoda1( ) {
        Poligon p = new Poligon();
        Patrat t = new Patrat();
        if (...)
                return p;       // legal
        else
                return t;       // legal
}
Patrat metoda2( ) {
        Poligon p = new Poligon();
        Patrat t = new Patrat();
        if (...)
                return p;       // ilegal
        else
                return t;       // legal
}
```

Trimiterea parametrilor catre o metoda

Signatura unei metode este data de numarul si tipul argumentelor primite de acea metoda:

```
metoda([tip1 argument1] [, tip2 argument2] ...  )
```

Tipul de date al unui argument poate fi orice tip valid al limbajului, atât tip primitiv de date cât si referinta la un obiect.

```
Ex: adaugarePersoana(String nume, int varsta, float salariu)
String este tip referinta, int si float sunt tipuri primitive
```

Spre deosebire de alte limbaje, în Java nu pot fi trimise ca parametri ai unei metode referinte la alte metode (functii), însa pot fi trimise obiecte care sa contina implementarea acelor metode, pentru a fi

apelate. De asemenea, în Java o metoda nu poat primi un numar variabil de argumente, ceea ce înseamna ca apelul unei metode trebuie sa se faca cu specificarea exacta a numarului si tipurilor argumentelor.
Numele argumentelor primite trebuie sa difere între ele si nu trebuie sa coincida cu numele nici uneia din variabilele locale ale metodei. Pot însa sa coincida cu numele variabilelor membre ale clasei caz în care diferentierea se va face prin intermediul variabile **`this`**.

```
class Cerc {
        int x, y, raza;
        public Cerc(int x, int y, int raza) {
                this.x = x;
                this.y = y;
                this.raza = raza;
        }
}
```

Atentie: In Java argumentele sunt trimise doar prin valoare (*pass-by-value*)

Acest lucru însemna ca metoda receptioneaza doar valorile variabilelor primite ca parametri. Când argumentul are tip primitiv de date metoda nu-i poate schimba valoarea decât local (în cadrul metodei); la revenirea din metoda variabila are aceeasi valoare ca la apelul inittial al metodei (modificarile facute în cadrul metodei sunt pierdute).
Când argumentul este de tip referinta metoda nu poate schimba referinta obiectului însa poate apela metodele acelui obiect si poate modifica orice variabila membra accesibila.
Asadar, daca dorim ca o metoda sa schimbe starea(valoarea) unui argument primit, atunci el trebuie sa fie neaparat de tip referinta (trebuie sa fie un obiect!). De exemplu, sa consideram clasa `Cerc` descrisa anterior si dorim sa implementam o metoda care sa returneze parametrii cercului:

Varianta incorecta

```
...
int x = -1, y = -1, r = -1;
cerc.aflaParametri(x, y, r);
System.out.println("x = " + x + ", y = " + y + ", raza = " + r);
. . .
```

In acesct scop în clasa `Cerc` ar trebui sa avem o metoda de genul:

```
class Cerc {
        int x, y, raza;
        ...
        public void aflaParametri(int valx, int valy, int valr) {
                //metoda nu functioneaza cum ar trebui!
                valx = x;
                valy = y;
                valr = raza;
        }
}
```

Aceasta metoda însa nu va realiza lucrul propul întrucât ea primeste doar valorile variabilelor `x, y si r` adica `(-1, -1, -1)` si nu referinte la ele (adresele lor de memorie) astfel încât sa le poata modifica valoarea. In concluzie, metoda nu realizeaza nimic pentru ca nu poate schimba valorile unor variabile aflate în afara corpului ei.

Varianta corecta
Definim o clasa suplimentara care descrie parametrii pe care dorim sa-i aflam:

```
class CParam {
        public int x, y, raza;
}
```

```
class Cerc {
        int x, y, raza;
        public void aflaParametri(CParam param) {
                param.x = x;
                param.y = y;
                param.raza = raza;
        }
}

...
CParam p = new CParam();
cerc.aflaParametri(p);
System.out.println("x = " + p.x + ", y = " + p.y + ", raza = " + p.raza);
```

Supraîncarcarea si supradefinirea metodelor

Sunt doua concepte extrem de utile ale POO si se refera la:

- *supraîncarcarea (overloading)* : în cadrul unei clase pot exista metode cu acelasi nume cu conditia ca signaturile lor sa fie diferite (lista de argumente primite sa difere fie prin numarul argumentelor, fie prin tipul lor) astfel încât la apelul functiei cu acel nume sa se poata stabili în mod unic care dintre ele se executa. Fenomenul de supradefinire a metodelor se mai numeste si *polimorfism*.
- *supradefinirea (overriding)*: o subclasa a unei clase poate rescrie o metoda a clasei parinte, prin implementarea unei metode cu acelasi nume si aceeasi signatura ca ale superclasei.

Exemplificare:

```
class A {
        void metoda() {
                System.out.println("A: metoda fara parametru");
        }
        //supraincarcare - polimorfism
        void metoda(int arg) {
                System.out.println("A: metoda cu un parametru");
        }
}

class B extends A {
        //supradefinire
        void metoda() {
                System.out.println("B: metoda fara parametru");
        }
}
```

O metoda supradefinita poate sa:

- ignore complet codul metodei corespunzatoare din superclasa (cazul de mai sus)

```
B b = new B();
b.metoda();    -> afiseaza "B: metoda fara parametru"
```

- sa extinda codul metodei parinte, executând înainte de codul propriu si functia parinte.

```
class B extends A {
        //supradefinire prin extensie
        void metoda() {
                super.metoda();
                System.out.println("B: metoda fara parametru");
        }
```

```
}
. . .
B b = new B();
b.metoda();    -> afiseaza :
"A: metoda fara parametru"
"B: metoda fara parametru"
```

Metode finale

Specifica faptul ca acea metoda nu mai poate fi supradefinita în subclasele clasei în care ea este definita ca fiind finala. Acest lucru este util daca respectiva metoda are o implementare care nu trebuie schimbata sub nici o forma în subclasele ei, fiind critica pentru consistenta starii unui obiect. De exemplu studentilor unei universitati trebuie sa li se calculeze media finala în functie de notele obtinute la examene în aceeasi maniera, indiferent de facultatea la care sunt.

```
class Student {
        . . .
        final float calcMedie(int nrExamene, float note[], float ponderi[]) {
                ...
        }
        . . .
}
class StudentInformatica extends Student{
        float calcMedie(int nrExamene, float note[], float ponderi[]) {
                return 10.00;
        }
}//eroare la compilare!
```

Specificatori de acces pentru membrii unei clase

Sunt cuvinte rezervate ce controleaza accesul celorlate clase la membrii unei clasei. Specificatorii de acces pentru variabilele si metodele unei clase sunt: `public, protected, private` si cel implicit (`package`), iar nivelul lor de acces este dat în tabelul de mai jos:

Specificator	Clasa	Subclasa	Pachet	Toti
private	X			
protected	X	X**	X	
public	X	X	X	X
package*	X	X		

Exemple de declaratii:

```
private int secretPersonal;
proteceted String secretDeFamilie;
public Vector elemente;
long doarIntrePrieteni ;       -> package

private void metodaInterna();
public double calculeazaRezultat();
```

Obs1(*): Daca nu este specificat nici un modificator de acces, implicit nivelul de acces este la nivelul pachetului (package). Asadar, modificatorul "package" nu apare explicit în declararea unei variabile/metode (în cazul în care apare, compilatorul va furniza o eroare).
Obs2(**): In cazul în care declaram un membru "protected" atunci accesul la acel membru din subclasele clasei în care a fost declarata variabila depinde si de pachetul în care se gaseste subclasa: daca sunt în acelasi pachet accesul este permis, daca nu sunt în acelasi pachet accesul nu este permis decât pentru obiecte de tipul subclasei.

Membri de instanta si membri de clasa

O clasa Java poate contine doua tipuri de variabile si metode :

- *de instanta*: declarate **fara** modificatorul **`static`**, specifice fiecarei instante si
- *de clasa*: declarate cu modificatorul **`static`**, specifice clasei

Variabile

Când declarati o variabila membra cum ar fi `x` în exemplul de mai jos:

```
class MyClass {
        int x ; //variabila de instanta
}
```

se declara de fapt o variabila de instanta, cee ce însemna ca la fiecare creare a unei instante a clasei `MyClass` sistemul aloca o zona de memorie separata pentru memorarea valorii lui x.

```
MyClass o1 = new MyClass();
o1.x = 100;
MyClass o2 = new MyClass();
o2.x = 200;
System.out.println(o1.x) -> afiseaza 100
System.out.println(o2.x) -> afiseaza 200
```

Asadar, fiecare obiect nou creat va putea memora valori diferite pentru variabilele sale de instanta. Pentru variabilele de clasa (statice) sistemul aloca o singura zona de memorie la care au acces toate instantele clasei respective, ceea ce înseamna ca daca un obiect modifica valoarea unei variabile statice ea se va modifica si pentru toate celelate obiecte.

```
class MyClass {
        int x ; //variabila de instanta
        static long n; //variabila de clasa
}
. . .
MyClass o1 = new MyClass();
MyClass o2 = new MyClass();
o1.n = 100;
System.out.println(o2.n) -> afiseaza 100
o2.n = 200;
System.out.println(o1.n) -> afiseaza 200
```

Metode

Similar ca la variabile, metodele declarate fara modificatorul `static` sunt metode de instanta iar cele declarate cu `static` sunt metode de clasa (statice). Diferenta între cele doua metode este urmatoarea:

- metodele de instanta opereaza atât pe variabilele de instanta cât si pe cele statice ale clasei
- metodele de clasa opereaza doar pe variabilele statice ale clasei

```
class MyClass {
        int x ; //variabila de instanta
        static long n; //variabila de clasa
        void metodaDeInstanta() {
                n ++;           //legal
                x --;           //legal
        static void metodaStatica() {
                n ++;           //legal
                x --;           //ilegal
}
```

Intrucât metodele de clasa nu depind de starea obiectelor clasei respective, apelul lor se poate face astfel:

```
MyClass.metodaStatica();       //legal, echivalent cu
MyClass obj = new MyClass();
obj.metodaStatica();           //legal
```

spre deosebire de metodele de instanta care nu pot fi apelate decât unei instante a clasei respective:

```
MyClass.metodaDeInstanta(),           //ilegal
MyClass obj = new MyClass();
obj.metodaDeInstanta();                       //legal
```

Utilitatea membrilor de clasa (statici)

Sunt folositi pentru a pune la dispozitie valori si metode independente de starea obiectelor dintr-o anumita clasa.

1. **Declararea constantelor**

```
class MyClass {
        final double PI = 3.14; //variabila finala de instanta
}
```

La fiecare instantiere a clasei `MyClass` va fi rezervata zona de memorie pentru variabilele finale ale obiectului respectiv, ceea ce este o risipa întrucât aceste constante au aceleasi valori pentru toate instantele clasei. Declararea corecta a constantelor trebuie asadar facuta cu modificatorii **`static si final`**, pentru a le rezerva o singura zona de memorie, comuna tuturor obiectelor:

```
class MyClass {
        static final double PI = 3.14; //variabila finala de clasa
}
```

5. **Numararea obiectelor unei clase**

```
class MyClass {
        static long nrInstante = 0;
        MyClass() {    //constructorul este apelat la fiecare instantiere
                nrInstante ++;
        }
}
```

Folosind variabile statice putem controla diversi parametri legati de crearea obiectelor unei clase

12. **Implementarea functiilor globale**
Spre deosebire de C++, în Java nu putem avea functii globale definite ca atare, întrucât "orice este un obiect". Din acest motiv si metodele care au o functionalitate globala trebuie implementate în cadrul unei clase. Acest lucru se va face prin intermediul metodelor de clasa (globale), deoarece acestea nu depind de starea particulara a obiectelor din clasa respectiva. De exemplu, sa consideram functia globala `sqrt` care extrage radicalul unui numar si care se

gaseste în clasa `Math`. Daca nu ar fi fost functie de clasa, apelul ei ar fi trebuit facut astfel (incorect, de altfel):

```
13.         Math obj = new Math();
14.         double rad121 = obj.sqrt(121);
```

ceea ce ar fi fost extrem de neplacut pentru programatori. Fiind însa functie statica ea poate fi apelata prin: `Math.sqrt(121)` .

Initializarea variabilelor de clasa

Se poate face în momentul declararii lor :

```
class MyClass {
        static final double PI = 3.14;
        static long nrInstante = 0;
        static final double EPS = 0.01;
}
```

sau prin intermediul unui *bloc static de initializare*:

```
class MyClass {
        static {
                final double PI = 3.14;
                long nrInstante = 0;
                final double EPS = 0.01;
        }
}
```

Clase imbricate

O clasa imbricata este, prin definitie, o clasa membra a unei alte clase

```
        class ClasaDeAcoperire{
                . . .
                class ClasaImbricata {
                        . . .
                }
        }
```

Folosirea claselor imbricate se face atunci când o alta clasa are nevoie în implementarea ei de o alta clasa si nu exista nici un motiv pentru care clasa imbricata sa fie declarata de sine statatoare (nu mai este folosita nicaieri).

```
public class Pachet {  //clasa de acoperire
        class Continut {       //clasa imbricata
                private String marfa;
                private float cantitate;
                Continut (String marfa, float cantitate) {
                        this.marfa = marfa;
                        this.cantitate = cantitate;
                }
        }
```

```
        class Destinatie {     //clasa imbricata
                private String dest;
                private int termen;
                Destinatie(String undePleaca, int inCateZile) {
                dest = undePleaca;
                        termen = inCateZile;
                        }
        }

        public void trimite(String marfa, float cant, String dest,
                Continut c = new Continut(marfa, cant);
                Destinatie d = new Destinatie(dest, termen);
        }

        public static void main(String[] args) {
                Pachet p = new Pachet();
                p.trimite("banane", 100, "Romania", 7);
        }
}
```

Ca membra a unei clase, o clasa imbricata are un privilegiu special fata de celelalte clase: acces nelimitat la variabilele clasei de acoperire, chiar daca acestea sunt private.

Clase interne

Ca orice alta clasa o clasa imbricata poate fi declarata statica sau nu. O clasa imbricata nestatica se numeste clasa interna.

```
class ClasaDeAcoperire{
        . . .
        static class ClasaImbricataStatica {
                . . .
        }
        class ClasaInterna {
                . . .
        }
}
```

Diferentierea acestor denumiri se face deoarece:

- o "clasa imbricata" reflecta relatia sintactica a doua clase; codul unei clase apare în interiorul dodului altei clase;
- o "clasa interna" reflecta relatia dintre instantele a doua clase, în sensul ca o instanta a unei clase interne nu poate exista decât în cadrul unei instante a clasei de acoperire.

In general cele mai folosite clase imbricate sunt clasele interne
Asadar, o clasa interna este o clasa imbricata ale carei instante nu pot exista decât în cadrul instantelor clasei de acoperire si care are acces direct la toti membrii clasei sale de acoperire.

Identificarea claselor imbricate

Dupa cum stim orice clasa produce la compilare asa numitele "unitati de compilare", care sunt fisiere având numele clasei respective si extensia `.class`, si care contin toate informatiile despre clasa respectiva.
Pentru clasele imbricate aceste unitati de compilare sunt denumite astfel: numele clasei de acoperire, urmat de simbolul '$' apoi de numele clasei imbricate.

```
class ClasaDeAcoperire{
        class ClasaInterna1 {}
        class ClasaInterna2 {}
}
```

Pentru exemplul de mai sus vor fi generate trei fisiere:

```
ClasaDeAcoperire.class
ClasaDeAcoperire$ClasaInterna1.class
ClasaDeAcoperire$ClasaInterna2.class
```

In cazul în care clasele imbricate au la rândul lor alte clase imbricate (situatie mai putin uzuala) denumirea lor se face dupa aceeasi metoda : adaugarea unui '$' si apoi numele clasei imbricate.

Clase si metode abstracte

Uneori în proiectarea unei aplicatii este necesar sa reprezentam cu ajutorul claselor concepte abstracte care sa nu poata fi instantiate si care sa foloseasca doar la dezvoltarea ulterioara a unor clase ce descriu obiecte concrete. De exemplu în pachetul `java.lang` exista clasa abstracta `Number` care modeleaza conceptul generic de "numar". Intr-un program nu avem însa nevoie de numere generice ci de numere întregi, reale, etc. Clasa `Number` serveste ca superclasa pentru clase cum ar fi `Byte, Double, Float, Integer, Long si Short` ce implementeaza obiecte pentru descrierea numerelor de un anumit tip. Asadar clasa `Number` reprezinta un concept abstract si nu vom putea instantia obiecte de acest tip:

```
Number numarGeneric = new Number(); //eroare
```

Declararea unei clase abstracte

Se face folosind cuvântul rezervat **`abstract`** în declaratia clasei:

```
abstract class ClasaAbstracta {
        . . .
}
```

Daca vom încerca sa instantiem un obiect al clasei `ClasaAbstracta` vom obtine o eroare la compilarea programului de genul:`class ClasaAbstracta is an abstract class. It can't be instantiated`.

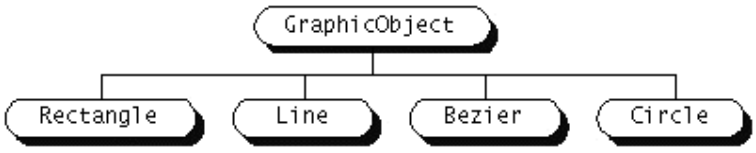
Metode abstracte

Spre deosebire de clasele obisnuite care trebuie sa furnizeze implementari pentru toate metodele declarate o clasa abstracta poate contine metode fara nici o implementare. Metodele fara nici o implementare se numesc metode abstracte si pot aparea doar în clase abstracte.
In fata unei metode abstracte trebuie sa apara cuvântul cheie `abstract`, altfel va fi furnizata o eroare de compilare.

```
abstract void metodaAbstracta();      //corect
void metoda(); //incorect
(Method metoda requires a method body; otherwisw declare it abstract)
```

In felul acesta o clasa abstracta poate pune la dispozitia subclaselor sale un model complet pe care trebuie sa-l implementeze, furnizând chiar implementarea unor metode comune tuturor claselor sau lasând explicitarea unor metode fiecarei subclase în parte.
Un exemplu elocvent de folosire a claselor si metodelor abstracte este descrierea obiectelor grafice într-o maniera orientata-obiect.

- Obiecte grafice : linii, dreptunghiuri, cercuri, curbe Bezier, etc
- Stari comune : pozitia(originea), dimensiunea, culoarea, etc
- Comportament : mutare, redimensionare, desenare, colorare, etc.



Pentru a folosi starile si comportamentele comune acestor obiecte în avantajul nostru putem declara o clasa generica `GraphicObject` care sa fie superclasa pentru celelalte clase.

Metodele abstracte vor fi folosite pentru implementarea comportamentului specific fiecarui obiect, cum ar fi desenarea iar cele obisnuite pentru comportamentul comun tuturor, cum ar fi schimbarea originii.
Implementarea clasei abstracte GraphicObject:

```
abstract class GraphicObject {
        int x, y;
        . . .
        void moveTo(int newX, int newY) {     //metoda normala
                . . .
        }
        abstract void draw();                 //metoda abstracta
}                                                       //nici o
implementare
```

Atentie: Orice subclasa non-abstracta a unei clase abstracte trebui sa furnizeze implementari ale metodelor abstracte din superclasa.
Implementarea claselor pentru obiecte grafice ar fi:

```
class Circle extends GraphicObject {
        void draw() {
                . . .   //obligatoriu implementarea
        }
}
```

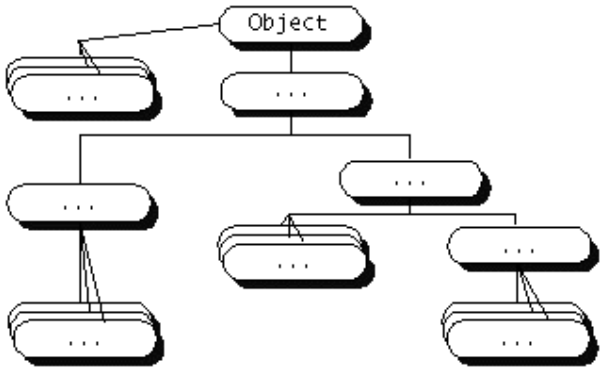
```
class Rectangle extends GraphicObject {
        void draw() {
                . . .   //obligatoriu implementarea
        }
}
```

Observatii:
O clasa abstracta poate sa nu aiba nici o metoda abstracta.
O metoda abstracta nu poate aparea decât într-o clasa abstracta.
Orice clasa care are o metoda abstracta trebuie declarata ca fiind abstracta.

Clasa Object

Orice clasa are o superclasa

Dupa cum am vazut la crearea unei clase clauza "extends" specifica faptul ca acea clasa este o subclasa a altei clase, numita superclasa. O clasa poate avea o singura superclasa (Java nu suporta mostenirea multipla) si chiar daca nu specificati clauza "extends" la crearea unei clase ea totusi va avea o superclasa. Cu alte cuvinte, in Java orice clasa are o superclasa si numai una.



Superclasa tuturor celorlalte clase este clasa **`Object`**, care este radacina ierarhiei de clase în Java. Clasa `Object` este si superclasa implicita a claselor care nu specifica o alta superclasa. Declaratia `class MyClass {}` este echivalenta cu `class MyClass extends Object {}`.

Clasa Object

Este cea mai generala dintre clase, orice obiect fiind, direct sau indirect, descendent al acestei clase. Defineste si implementeaza comportamentul comun al tuturor claselor Java cum ar fi:

- posibilitatea compararii obiectelor între ele
- specificarea unei reprezentari ca sir de caractere a unui obiect
- returnarea clasei din care face parte un obiect
- notificarea altor obiecte ca o variabila de conditie s-a schimbat, etc

Fiind sublcasa a lui `Object`, orice clasa poate supradefini metodele clasei `Object` care nu sunt finale. Metode care pot fi supradefinite sunt: **`clone, equals/hashCode, finalize ,toString`**

`clone`	Acesasta metoda este folosita pentru duplicarea obiectelor (creearea unor clone). Clonarea unui obiect presupune crearea unui nou obiect de acelasi tip si care sa aiba aceeassi stare (aceleasi valori pentru variabilele sale)
`equals, hashCode`	Aceste metode trebuie supradefinite împreuna. In metoda `equals` este scris codul pentru compararea a doua obiecte. Implicit (implementarea din clasa Object) aceasta metoda compara referintele obiectelor. Uzual este redefinita pentru a testa daca starile obiectelor coincid sau daca doar o parte din variabilele lor coincid. Metoda `hashCode` returneaza un cod întreg pentru fiecare obiect pentru a testa consistenta obiectelor: acelasi obiect trebuie sa returneze acelasi cod pe durata executiei programului. Daca doua obiecte sunt egale conform metodei `equals` atunci apelul metodei `hashCode` pentru fiecare din cele doua obiecte trebuie sa returneze acelasi întreg.
`finalize`	In aceasta metoda se scrie codul care "curata dupa un obiect" înainte de a fi eliminat din memorie de colectorul de gunoaie.(vezi "Distrugerea obiectelor")

`toString`	Este folosita pentru a returna o reprezentare ca sir de caractere a unui obiect. Este utila pentru concatenarea sirurilor cu diverse obiecte în vederea tiparirii.
	`MyObject obj = new MyObject();`
	`System.ot.println("Obiect=" + obj); //echivalent cu`
	`System.ot.println("Obiect=" + obj.toString());`

Exemplu de supradefinire a metodelor clasei `Object`

```
class MyObject extends Object {
  //clasa in care este supradefinita metoda equals
  public int x;
  public boolean equals(Object obj) {
          if (obj == null) return false;
          if (!(obj instanceof MyObject)) return false;
          if ( ((MyObject) obj).x == x)
                  return true;
          else
                  return false;
  }
  public synchronized Object clone() {
          MyObject m = new MyObject();
          m.x = x;
          return m;
  }
  public final synchronized String toString() {
          return "[" + x + "]";
  }
  public void finalize() {
          System.out.println("Finalizare");
          x = -1;
  }
}

class OtherObject {
  public int x;
  //clasa in care NU este supradefinita metoda equals
}
```

```
class TestObject {       //clasa principala - contine metoda main
  public static void main(String args[]) {
          OtherObject o1 = new OtherObject();
          OtherObject o2 = new OtherObject();
          o1.x = 1;
          o2.x = 1;
          if (o1.equals(o2))
                  System.out.println("test1: o1 == o2");
          else
                  System.out.println("test1: o1 != o2");//corect
          //Desi x=1 pt ambele obiecte egalitatea se obtine doar cand
          //adresele lor de memorie coincid deoarece nu este implementata
          //metoda equals

          o2 = o1;
          if (o1.equals(o2))
                  System.out.println("test2: o1 == o2");//corect
          else
                  System.out.println("test2: o1 != o2");

          ////////////////////////////////////////////
          MyObject m1 = new MyObject();
          MyObject m2 = new MyObject();
          m1.x = 1;
          m2.x = 1;
          if (m1.equals(m2))
                  System.out.println("test3: m1 == m2");//corect
          else
                  System.out.println("test3: m1 != m2");
          //x=1 pt ambele obiecte -> metoda equals returneaza true

          MyObject m3;
          m3 = (MyObject) m1.clone();
          System.out.println("Obiectul clonat: " + m3); //echivalent cu
          System.out.println("Obiectul clonat: " + m3.toString());
          //Se tipareste: Obiectul clonat: [1]
  }
}
```