Lecţia 2

Clase şi Obiecte

Presupunem că dorim să *descriem* într-un limbaj de programare un obiect ceas. În general, unui ceas îi putem seta ora, minutul şi secunda şi în orice moment putem şti ora exactă. Cum am putea realiza acest lucru?

Dacă *descriem* acest obiect (tip de date abstract) într-un limbaj de programare structural, spre exemplu C, atunci vom crea, ca mai jos, o structură *Ceas* împreună cu nişte funcţii cuplate de această structură. Cuplajul e realizat prin faptul că orice funcţie care operează aspupra unui ceas conţine în lista sa de parametri un parametru de tip *Ceas*.

```
typedef struct _Ceas {
        int ora, minut, secunda;
} Ceas;

void setareTimp(Ceas *this, int ora, int minut, int secunda) {
    this->ora = ((ora >= 0 && ora < 24) ? ora : 0);
    this->minut = ((minut >= 0 && minut < 60) ? minut : 0);
    this->secunda = ((secunda >= 0 && secunda < 60) ? secunda : 0);
}

void afiseaza(Ceas *this) {
    printf("Ora setata %d:%d:%d", this->ora, this->minut, this->secunda);
}
```

Dacă modelăm acest obiect într-un limbaj orientat pe obiecte (în acest caz, Java), atunci vom crea o clasă *Ceas* ca mai jos.

```
class Ceas {

    private int ora, minut, secunda;
```

```
    public void setareTimp(int o, int m, int s) {
        ora = ((o >= 0 && o < 24) ? o : 0);
        minut = ((m >= 0 && m < 60) ? m : 0);
        secunda = ((s >= 0 && s < 60) ? s : 0);
    }

    public void afiseaza() {
        System.out.println("Ora setata "+ora+":"+minut+":"+secunda);
    }
}
```

Se poate uşor observa din cadrul exemplului de mai sus că atât datele cât şi funcţiile care operează asupra acestora se găsesc în interiorul aceleiaşi entităti numită *clasă*. Desigur, conceptele folosite în codul de mai sus sunt încă necunoscute dar cunoaşterea lor este scopul principal al acestei lecţii.

2.1 Clase şi obiecte

2.1.1 Ce este un obiect? Ce este o clasă?

În cadrul definiţiei parţiale a programării orientate pe obiecte pe care am dat-o în lecţia anterioară se spune că "programele orientate pe obiecte sunt organizate ca şi colecţii de obiecte". Dar până acum nu s-a spus ce este, de fapt, un obiect. În acest context, este momentul să completăm Definiţia 1:

Definiţie 4 *Programarea orientată pe obiecte este o metodă de implementare a programelor în care acestea sunt organizate ca şi colecţii de obiecte care cooperează între ele,* **fiecare obiect reprezentând instanţa unei clase**.

Definiţia de mai sus este încă incompletă, forma ei completă fiind Definiţia 7 din Lecţia 5.

Atunci când un producător crează un produs, mai întâi acesta specifică toate caracteristicile produsului într-un document de specificaţii iar pe baza acelui document se crează fizic produsul. De exemplu, televizorul este un produs creat pe baza unui astfel de document de specificaţii. La fel stau lucrurile şi într-un program orientat pe obiecte: mai întâi se crează clasa obiectului (documentul de specificaţii) care înglobează toate caracteristicile unui obiect (instanţă a clasei) după care, pe baza acesteia, se crează obiectul în memorie.

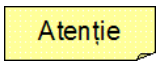
În general, putem spune că o clasă furnizează un şablon ce specifică datele şi operaţiile ce aparţin obiectelor create pe baza şablonului – în documentul de specificaţii pentru un televizor se menţionează că acesta are un tub catodic precum şi nişte butoane care pot fi apăsate.

2.1.2 Definirea unei clase

Din cele de mai sus deducem că o clasă descrie un obiect, în general, un nou tip de dată. Într-o clasă găsim date şi metode ce operează asupra datelor respective.

Pentru a defini o clasă trebuie folosit cuvântul cheie *class* urmat de numele clasei:

```
class NumeClasa {
    //Date si metode membru
}
```



O metodă nu poate fi definită în afara unei clase.

Acum haideţi să analizăm puţin exemplul dat la începutul capitolului. Clasa *Ceas* modelează tipul de date abstract Ceas. Un ceas are trei date de tip *int* reprezentând ora, minutul şi secunda precum şi două operaţii: una pentru setarea acestor atribute iar alta pentru afişarea lor. Cuvintele *public* şi *private* sunt cuvinte cheie ale căror roluri sunt explicate în Secţiunea 2.3.1.

Datele *ora, minut, secunda* definite în clasa *Ceas* se numesc *atribute*, *date-membru*, *variabile-membru* sau *câmpuri* iar operaţiile *setareTimp, afiseaza* se numesc *metode* sau *funcţii-membru*.

2.1.3 Crearea unui obiect

Spuneam mai sus că un obiect este o instanţă a unei clase. În Java instanţierea sau crearea unui obiect se face dinamic, folosind cuvăntul cheie *new*:

```
new Ceas();
```

Aşa cum fiecare televizor construit pe baza documetului de specificaţii are propriul său tub catodic, fiecare obiect de tip *Ceas* are propriile sale atribute.

2.2 Referinţe la obiecte

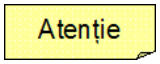
În secţiunile anterioare am vazut cum se defineşte o clasă şi cum se crează un obiect. În această secţiune vom vedea cum putem executa operaţiile furnizate de obiecte.

Pentru a putea avea acces la operaţiile furnizate de către un obiect, trebuie să deţinem o *referinţă* spre acel obiect.

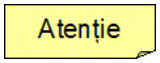
Telecomanda asociată unei lămpi electrice poate fi văzută ca o referinţă la aceasta. Atâta timp cât avem telecomanda, putem oricând prin intermediul ei să accesăm un serviciu furnizat de către lampă (spre exemplu, pornirea/oprirea sau mărirea/micşorarea intensităţii luminoase). Dacă părăsim încăperea şi dorim în continuare să controlăm lampa, luăm telecomanda şi nu lampa cu noi!

Declararea unei referinţe numite *ceas* spre un obiect de tip *Ceas* se face în felul următor:

```
Ceas ceas;
```



Limbajul Java este *case-sensitive* şi din această cauză putem avea referinţa numită *ceas* spre un obiect de tip *Ceas*.



Faptul că avem la un moment dat o referinţă nu implică şi existenţa unui obiect indicat de acea referinţă. Până în momentul în care referinţei nu i se ataşează un obiect, aceasta nu poate fi folosită.

Mai sus a fost creată o referinţă spre un obiect de tip *Ceas* dar acesteia încă nu i s-a ataşat vreun obiect şi, prin urmare, referinţa încă nu poate fi utilizată. Ataşarea unui obiect la referinţa *ceas* se face printr-o operaţie de atribuire, ca în exemplul următor:

```
ceas = new Ceas();
```

Putem avea doar telecomnda unui lămpi fără a deţine însă şi lampa. În acest caz telecomanda nu referă nimica. Pentru ca o referinţă să nu indice *vreun obiect* acesteia trebuie să i se atribuie valoarea *null*.

Valoarea *null*, ce înseamnă *nici un obiect referit*, nu este atribuită automat tuturor variabilelor referinţă la declararea lor. Regula este următoarea: dacă referinţa este un membru al unei clase şi ea nu este iniţializată în nici un fel, la instanţierea unui obiect al clasei respective referinţa va primi implicit valoarea *null*. Dacă însă referinţa este o variabilă locală ce aparţine unei metode, iniţializarea implicită nu mai funcţionează. De aceea se recomandă ca programatorul să realizeze *întotdeauna* o iniţializare explicită a variabilelor.

Acum, fiindcă avem o referinţă spre un obiect de tip *Ceas* ar fi cazul să setăm obiectului referit ora exactă. Apelul către metodele unui obiect se face prin intermediul referinţei spre acel obiect ca în exemplul de mai jos:

```
class ClientCeas {

    public static void main(String[] args) {
        Ceas ceas = new Ceas();
        int o = 15;
        ceas.setareTimp(o, 12, 20);
        ceas.afiseaza();
    }
}
```

Atenţie

Apelul metodei *afiseaza* nu este *afiseaza(ceas)* ci *ceas.afiseaza()* întrucât metoda *afiseaza* aparţine obiectului referit de *ceas* – se apelează metoda *afiseaza* pentru obiectul referit de variabila *ceas* din faţa lui.

Haideţi să considerăm exemplul de mai jos în care creăm două obiecte de tip *Ceas* precum şi trei referinţe spre acest tip de obiecte. Fiecare dintre obiectele *Ceas* create are alocată o zonă proprie de memorie în care sunt stocate valorile câmpurilor *ora, minut, secunda*. Ultima referinţă definită în exemplul de mai jos, prin atribuirea *c3 = c2* va referi şi ea exact acelaşi obiect ca şi *c2*, adică al doilea obiect creat. Vizual, referinţele şi obiectele create după execuţia primelor cinci linii de cod de mai jos sunt reprezentate în Figura 2.1.

```
class AltClientCeas {

    public static void main(String[] args) {
        Ceas c1 = new Ceas();
        c1.setareTimp(15, 10, 0);
        Ceas c2 = new Ceas();
        c2.setareTimp(15, 10, 0);
        Ceas c3 = c2;
        c3.setareTimp(8, 12, 20);
        c2.afiseaza();
    }
}
```

La un moment dat, putem avea două telecomenzi asociate unei lămpi electrice. Exact la fel se poate întâmpla şi în cazul unui program – putem avea acces la serviciile puse la dispoziţie de un obiect prin intermediul mai multor referinţe.

În contextul exemplului anterior, se pune problema ce se va afişa în urma execuţiei ultimei linii? Deoarece atât *c3* cât şi *c2* referă acelaşi obiect, se va afişa:

```
//Ora setata 8:12:20
//NU !! Ora setata 15:10:0
```

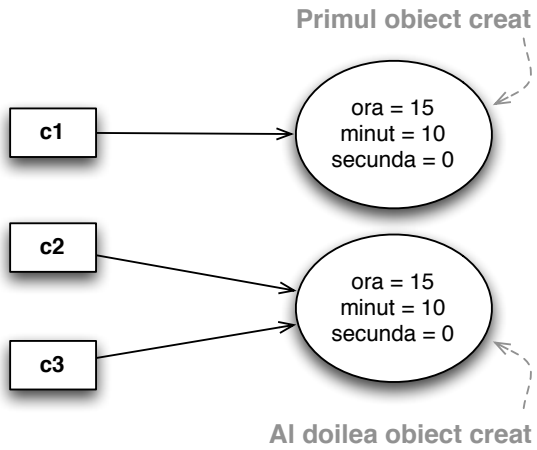


Figura 2.1: OBIECTE. REFERINŢE LA OBIECTE.

Dar ce se va afişa în urma execuţiei *c1.afiseaza()*? Deoarece *c1* referă un alt obiect ceas decât *c2*, respectiv *c3*, se va afişa:

```
c1.afiseaza();
//Ora setata 15:10:0
```

Să presupunem că am achiziţionat două lămpi de acelaşi tip. Putem oare să spunem că cele două lămpi sunt identice? Hmmm,...la prima vedere am putea spune: sigur, ele ne oferă aceleaşi servicii deci *categoric* sunt identice. Numai că atunci când vom cere cuiva să realizeze operaţia "Aprinde lampa" va trebui să precizăm şi care dintre cele două lămpi dorim să fie aprinsă. Vom preciza care lampă dorim să fie aprinsă în funcţie de *identitatea* sa – proprietatea unui obiect care îl distinge de restul obiectelor – spre exemplu, locul ocupat în spaţiu e o proprietate care le poate distinge. La fel ca cele două lămpi, şi cele două obiecte *ceas* create de noi au propria identitate – în acest caz identitatea poate fi dată de locul fizic pe care îl ocupă în memorie obiectele. Atribuirea *c3 = c2* nu a făcut altceva decât să ataşeze referinţei *c3* obiectul având aceeaşi identitate ca şi cel referit de *c2*, adică obiectul secund creat.

2.3 Componenţa unei clase

Clasele, aşa cum am vazut deja, sunt definite folosind cuvântul cheie *class*. În urmatoarele secţiuni vom vorbi despre diferite categorii de membri care pot apare în interiorul unei clase.

2.3.1 Specificatori de acces

În componenţa clasei *Ceas* se observă prezenţa cuvintelor cheie *private* şi *public*. Aceste cuvinte se numesc *specificatori de acces* şi rolul lor este de a stabili drepturile de acces asupra membrilor unei clase (atribute şi metode). În afară de cei doi specificatori de acces prezenţi în clasa *Ceas* mai există si alţii dar despre ei se va vorbi în alte lecţii.

Când discutăm despre drepturile de acces la membrii unei clase trebuie să abordăm acest subiect din două perspective: *interiorul* respectiv *exteriorul* clasei.

```
class Specificator {

    public int atributPublic;
    private int atributPrivat;

    public void metodaPublica() {
        atributPublic = 20;
        atributPrivat = metodaPrivata();
    }

    private int metodaPrivata() {
        return 10;
    }

    public void altaMetoda(Specificator s) {
        atributPublic = s.atributPrivat;
        //Corect, deoarece altaMetoda se afla in clasa Specificator
        atributPrivat = s.metodaPrivata();
        s.metodaPublica();
    }
}
```

În cadrul metodelor unei clase există acces nerestrictiv la toţi membrii clasei, atribute sau metode. Exemplul de mai sus ilustrează acest lucru.

În legătură cu accesul din interiorul unei clase trebuie spus că absenţa restricţiilor se aplică şi dacă este vorba despre membrii altui obiect, instanţă a aceleaşi clase.

```
class ClientSpecificator {

    public static void main(String[] args) {
        Specificator spec = new Specificator();
        spec.metodaPublica();
        spec.metodaPrivata(); //Eroare
        System.out.println(spec.atributPublic);
        System.out.println(spec.atributPrivat); //Eroare
    }
}
```

Membrii declaraţi cu specificatorul *private* NU sunt vizibili în afara clasei, ei fiind ascunşi. Clienţii unei clase pot accesa doar acei membri care au ca modificator de acces cuvântul *public*. Dacă încercăm să accesăm, aşa cum am făcut în metoda *main* de mai sus, membrii *private* ai clasei *Specificator* prin intermediul referinţei *spec* compilatorul va semnala o eroare.

De ce e nevoie de specificatorul de acces private?

Clasa *Ceas* descrie obiecte de tip ceas. Atributele *ora*, *minut* şi *secunda* pentru orice ceas trebuie să aibă valori cuprinse între 0 şi 23, respectiv 0 şi 59. Pentru ca un ceas să indice o oră validă metoda pentru setarea atributelor de mai sus trebuie scrisă astfel încât să nu permită setarea acestora cu o valoare eronată.

Dacă cele trei atribute ale unui obiect de tip *Ceas* ar putea fi accesate din exterior, atunci valorile lor ar putea fi setate în mod eronat.

În exemplul anterior scris în C nu putem ascunde cele trei câmpuri ale structurii *Ceas* şi în consecinţă acestea vor putea primi oricând alte valori decăt cele permise. În general, când scrieţi o clasă accesul la atributele sale trebuie să fie limitat la metodele sale membru pentru a putea controla valorile atribuite acestora.

În clasa *Ceas* cele trei atribute existente sunt de tip *int*. Dar oare nu există şi altă metodă prin care pot fi stocate cele trei caracteristici ale unui ceas? Cele trei caracterisitici ale unui ceas ar putea fi stocate, de exemplu, în loc de trei atribute întregi într-un tablou cu trei elemente întregi – dar un utilizator de ceasuri nu are voie să ştie *detaliile de implementare* ale unui obiect de tip *Ceas*!!!

Şi totuşi, ce e rău în faptul ca un utilizator de obiecte *Ceas* să ştie exact cum e implementat ceasul pe care tocmai îl foloseţe? Probabil că lucrurile ar fi *bune şi frumoase* până când proiectantul de ceasuri ajunge la concluzia că trebuie neapărat să modifice implementarea ceasurilor...şi atunci

toţi cei care folosesc ceasuri şi sunt dependenţi de implementarea acestora vor trebui să se *modifice*!!! În schimb, dacă clienţii unei clase depind de *serviciul oferit de implementator şi nu de implementarea acestuia* atunci ei nu vor fi afectaţi de modificările ulterioare aduse!!!

2.3.2 Constructori

În multe cazuri, atunci când instanţiem un obiect, ar fi folositor ca obiectul să aibă anumite atribute iniţializate.

Cum ar fi ca în momentul la care un producător a terminat de construit un ceas, ceasul să arate deja ora exactă? Beneficiul pe care l-am avea datorită acestui fapt este că am putea folosi imediat ceasul, nemaifiind nevoie să efectuăm iniţial operaţia de setare a timpului curent.

Iniţializarea atributelor unui obiect se poate face în mod automat, la crearea obiectului, prin intermediul unui *constructor*. Principalele caracteristici ale unui constructor sunt:

- un constructor are acelaşi nume ca şi clasa în care este declarat.
- un constructor nu are tip returnat.
- un constructor se apelează automat la crearea unui obiect.
- un constructor se execută la crearea obiectului şi numai atunci.

```
class Ceas {
    ...
    public Ceas(int o, int m, int s) {
        ora = ((o >= 0 && o < 24) ? o : 0);
        minut = ((m >= 0 && m < 60) ? m : 0);
        secunda = ((s >= 0 && s < 60) ? s : 0);
    }
}
```

Astfel, atunci când creăm un obiect, putem specifica şi felul în care vrem ca acel obiect să arate iniţial.

```
class ClientCeas {

    public static void main(String[] args) {
        Ceas ceas;
        ceas = new Ceas(12, 15, 15);
        ceas.afiseaza();
    }
}
```

Dacă programatorul nu prevede într-o clasă nici un constructor, atunci compilatorul va genera pentru clasa respectivă un constructor implicit fără nici un argument şi al cărui corp de instructiuni este vid. Datorită acestui fapt am putut scrie înaintea definirii propriului constructor pentru clasa *Ceas*

```
Ceas ceas;
ceas = new Ceas();
```

Dacă programatorul include într-o clasă cel puţin un constructor, compilatorul nu va mai genera constructorul implicit. Astfel, după introducerea constructorului anterior, codul de mai sus va genera erori la compilare. Să considerăm acum un alt exemplu:

```
class Constructor {

    private int camp = 17;

    public Constructor(int c) {
        camp = c;
    }

    public void afiseaza() {
        System.out.println("Camp " + camp);
    }
}
```

În clasa *Constructor* de mai sus am iniţializat atributul *camp* în momentul declarării sale cu *17*. Dar, după cum se poate observa, la instanţierea unui obiect se modifică valoarea atributului *camp*. În acest context se pune problema cunoaşterii valorii atributului *camp* al obiectului instanţiat mai jos. Va fi aceasta *17* sau *30*?

```
class ClientConstructor {

    public static void main(String[] args) {
        Constructor c;
        c = new Constructor(30);
        //Va afisa "Camp 30" datorita modului in care
        //se initializeaza campurile unui obiect
        c.afiseaza();
    }
}
```

Pentru o clasă în care apar mai multe mecanisme de iniţializare, aşa cum este cazul clasei *Constructor*, ordinea lor de execuţie este urmatoarea:

- Iniţializările specificate la declararea atributelor.

- Constructorii.

Atenţie Un constructor poate fi *private*!

2.3.3 Membri statici

Atunci când definim o clasă, specificăm felul în care obiectele de tipul acelei clase arată şi se comportă. Dar până la crearea efectivă a unui obiect folosind *new* nu se alocă nici o zonă de memorie pentru atributele definite în cadrul clasei iar la crearea unui obiect se alocă acestuia memoria necesară pentru fiecare atribut existent în clasa instanţiată. Tot până la crearea efectivă a unui obiect nu putem beneficia de serviciile definite în cadrul unei clase. Ei bine, există şi o excepţie de la regula prezentată anterior – membrii statici (atribute şi metode) ai unei clase. Aceşti membri ai unei clase pot fi folosiţi direct prin intermediul numelui clasei, fără a deţine instanţe a respectivei clase.

Spre exemplu, o informaţie de genul *câte obiecte de tipul Ceas s-au creat?* nu caracterizează o instanţă a clasei *Ceas*, ci caracterizează însăşi clasa *Ceas*. Ar fi nepotrivit ca atunci când vrem să aflăm numărul de instanţe ale clasei *Ceas* să trebuiască să creăm un obiect care va fi folosit doar pentru aflarea acestui număr – în loc de a crea un obiect, am putea să aflăm acest lucru direct de la clasa *Ceas*.

Important Un membru static al unei clase caracterizează clasa în interiorul căreia este definit precum şi toate obiectele clasei respective.

Un membru al unei clase (atribut sau metodă) este static dacă el este precedat de cuvântul cheie *static*. După cum reiese din Figura 2.2, un atribut static este un câmp comun ce are aceeaşi valoare pentru fiecare obiect instanţă a unei clase.

În continuare este prezentat un exemplu în care se numără câte instanţe ale clasei *Ceas* s-au creat.

```
class Ceas {

    private static int numarObiecte = 0;

    ...
    public Ceas(int o, int m, int s) {
        ...
        numarObiecte++;
    }
    ...
    public static int getNumarDeObiecte() {
        return numarObiecte;
    }
}
```

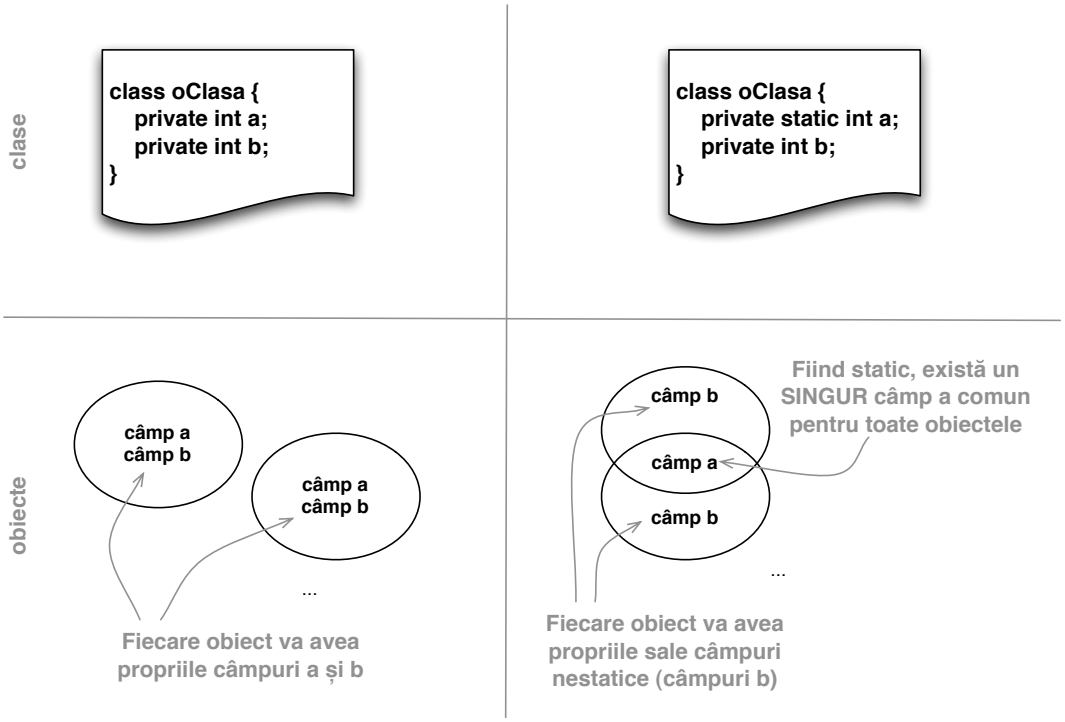


Figura 2.2: MEMBRI STATICI VERSUS MEMBRI NESTATICI.

Accesarea unui membru static al unei clase se poate face în cele două moduri prezentate mai jos. Evident, dacă metoda *getNumarDeObiecte* nu ar fi fost statică, ea nu ar fi putut fi apelată decât prin intermediul unei referinţe la un obiect instanţă a clasei *Ceas*.

```
//prin intermediul numelui clasei
Ceas.getNumarDeObiecte()
//prin intermediul unei referinte
Ceac ceas = new Ceas(12:12:50);
ceas.getNumarDeObiecte();
```

E bine ca referirea unui membru static să se facă prin intermediul numelui clasei şi nu prin intermediul unei referinţe; dacă un membru static e accesat prin intermediul numelui clasei atunci un cititor al programului va şti imediat că acel membru este static, în caz contrar această informaţie fiindu-i ascunsă.

Atenție Din interiorul unei metode statice pot fi accesaţi doar alţi membri statici ai clasei în care este definită metoda, accesarea membrilor nestatici ai clasei producând o eroare de compilare.

Spuneam anterior că un atribut static este un câmp comun ce are aceeaşi valoare pentru fiecare obiect instanţă a unei clase. Datorită acestui fapt, dacă noi setăm valoarea atributului *atributStatic* obiectului referit de *e1* din exemplul de mai jos, şi obiectul referit de *e2* va avea aceeaşi valoare corespunzătoare atributului static, adică *25*.

```
class ExempluStatic {
    public static int atributStatic = 15;
    public int atributNeStatic = 15;

    public static void main(String[] argv) {
        ExempluStatic e1 = new ExempluStatic();
        ExempluStatic e2 = new ExempluStatic();

        e1.atributStatic = 25;
        e1.atributNeStatic = 25;
        System.out.println("Valoare S" + e2.atributStatic);
        //Se va afisa 25 deoarece atributStatic este stocat intr-o
        //zona de memorie comuna celor doua obiecte
        System.out.println("Valoare NeS" + e2.atributNeStatic);
        //Se va afisa 15 deoarece fiecare obiect are propria zona
        //de memorie aferenta atributului nestatic
    }
}
```

2.3.4 Constante

În Java o constantă se declară folosind cuvintele cheie *static final* care preced tipul constantei.

```
class Ceas{
    public static final int MARCA = 323;
    ...
}
```

O constantă, fiind un atribut static, este accesată ca orice atribut static: *Ceas.MARCA*. Datorită faptului că variabila *MARCA* este precedată de cuvântul cheie *final*, acesteia i se poate atribui doar o singură valoare pe tot parcursul programului.

Un exemplu de membru *static final* este atributul *out* al clasei *System* pe care îl folosim atunci când tipărim pe ecran un mesaj.

2.4 Convenţii pentru scrierea codului sursă

Convenţiile pentru scrierea codului sursă îmbunătăţesc lizibilitatea codului, permiţându-ne să-l înţelegem mai repede şi mai bine. În acest context ÎNTOTDEAUNA:

- numele unei clase este format dintr-unul sau mai multe substantive, prima literă a fiecărui substantiv fiind o literă mare (exemplu: *Ceas*, *CeasMana*).
- numele unei metode începe cu un verb scris cu litere mici iar dacă numele metodei este format din mai multe cuvinte, prima literă a fiecărui cuvânt este mare (exemplu: *seteazaTimp*, *afiseaza*).
- numele unei constante este format dintr-unul sau mai multe cuvinte scrise cu litere mari separate, dacă este cazul, prin _ (exemplu: LATIME, LATIME_MAXIMA).

2.5 Notaţii UML pentru descrierea programelor

Atunci când scriem un program este bine ca etapa de scriere a codului să fie precedată de o etapă de proiectare, etapă în care se stabilesc principalele componente ale programului precum şi legăturile dintre ele. Componentele identificate în faza de proiectare vor fi apoi implementate.

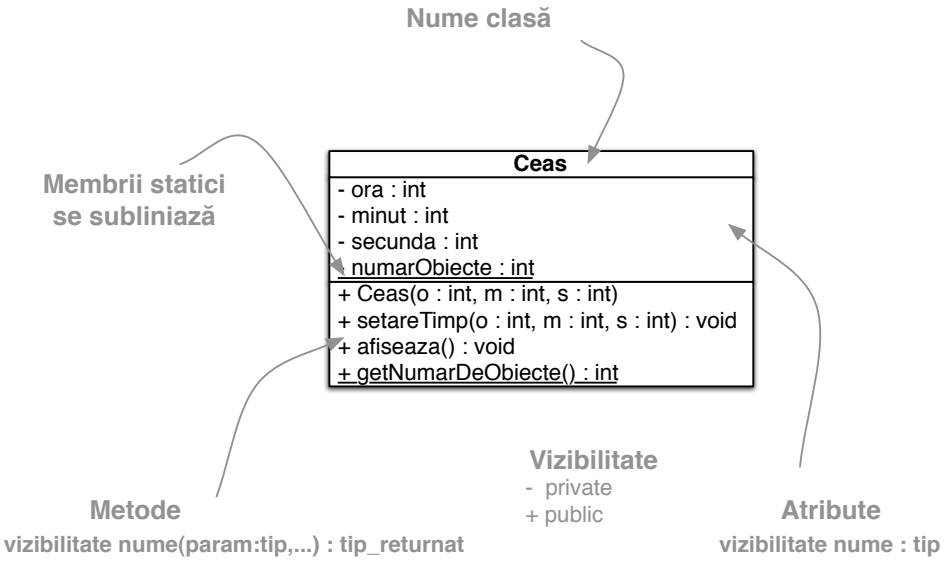


Figura 2.3: Reprezentarea unei clase în UML.

Una dintre cele mai utile notaţii pentru reprezentarea componentelor este UML (Unified Modeling Language). O reprezentare grafică în UML se numeşte diagramă. O diagramă arată principial ca un graf în care nodurile pot fi clase, obiecte, stări ale unui obiect iar arcele reprezintă relaţiile dintre nodurile existente.

Pentru a reprezenta clasele dintr-un program vom crea diagrame de clase, o clasă

reprezentându-se ca în Figura 2.3. Ca exemplu se prezintă clasa *Ceas* cu toate elementele discutate pe parcursul acestei lecţii. Diferitele tipuri de relaţii ce pot exista între două sau mai multe clase vor fi prezentate în cadrul altor lecţii.

2.6 Exerciţii

1. Creaţi o clasă cu un constructor privat. Vedeţi ce se întâmplă la compilare dacă creaţi o instanţă a clasei într-o metodă *main*.
2. Creaţi o clasă ce conţine două atribute nestatice private, un *int* şi un *char* care nu sunt iniţializate şi tipăriţi valorile acestora pentru a verifica dacă Java realizează iniţializarea implicită.
3.
```
class Motor {

    private int capacitate;

    public Motor(int c) {
           capacitate = c;
    }

    public void setCapacitate(int c) {
        capacitate = c;
    }

    public void tipareste() {
        System.out.println("Motor de capacitate " + capacitate);
    }
}
```
Fiind dată implementarea clasei *Motor*, se cere să se precizeze ce se va afişa în urma rulări secvenţei:
```
Motor m1, m2;
m1 = new Motor(5);
m2 = m1;
m2.setCapacitate(10);
m1.tipareste();
```

4. Un sertar este caracterizat de o lăţime, lungime şi înalţime. Un birou are două sertare şi, evident, o lăţime, lungime şi înalţime. Creaţi clasele *Sertar* şi *Birou* corespunzătoare specificaţiilor de mai sus. Creaţi pentru fiecare clasă constructorul potrivit astfel încât carateristicile instanţelor să fie setate la crearea acestora. Clasa *Sertar* conţine o metodă *tipareste* al cărei apel va produce tipărirea pe ecran a sertarului sub forma *"Sertar " + l + L + H*, unde *l, L, H* sunt valorile corespunzătoare lăţimii, lungimii şi înalţimii sertarului. Clasa *Birou* conţine o metodă

tipareste cu ajutorul căreia se vor tipări toate componentele biroului. Creaţi într-o metodă *main* două sertare, un birou şi tipăriţi componentele biroului.

5. Definiţi o clasă *Complex* care modelează lucrul cu numere complexe. Membrii acestei clase sunt:
   - două atribute de tip *double* pentru părţile reală, respectiv imaginară ale numărului complex
   - un constructor cu doi parametri de tip *double*, pentru setarea celor două părţi ale numărului(reală şi imaginară)
   - o metodă de calcul a modulului numărului complex. Se precizează că modulul unui număr complex este egal cu radical din (re*re+img*img) unde re este partea reală, iar img este partea imaginară. Pentru calculul radicalului se va folosi metoda statică predefinită *Math.sqrt* care necesită un parametru de tip *double* şi returneaza tot un *double*
   - o metodă de afişare pe ecran a valorii numărului complex, sub forma *re + i * im*
   - o metodă care returnează suma dintre două obiecte complexe. Această metodă are un parametru de tip *Complex* şi returnează suma dintre obiectul curent (obiectul care oferă serviciul de adunare) şi cel primit ca parametru. Tipul returnat de această metodă este *Complex*.
   - o metodă care returnează de câte ori s-au afişat pe ecran numere complexe.

   Pe lângă clasa *Complex* se va defini o clasă *ClientComplex* care va conţine într-o metoda *main* exemple de utilizare ale metodelor clasei *Complex*.

Bibliografie

1. Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
2. Harvey Deitel & Paul Deitel. *Java - How to program*. Prentice Hall, 1999, Capitolul 8, Object-Based Programming.
3. Bruce Eckel. *Thinking in Java, 4th Edition*. Prentice-Hall, 2006. Capitolul Everything is an object.
4. Martin Fowler. *UML Distilled, 3rd Edition*. Addison-Wesley, 2003.
5. Kris Jamsa. *Succes cu C++*. Editura All, 1997, Capitolul 2, Acomodarea cu clase şi obiecte.
6. *Java Code Conventions*. http://java.sun.com/docs/codeconv.