

PAOO L04

Clase	1
Constructori	5
Modificatori de acces	7
Probleme propuse	12

Clase

Clasele reprezintă elementele de bază ale unei aplicații în limbajul Java. Așa cum se știe, o clasă este un model de obiect. Ea conține câmpurile interne și metodele prin care se poate interacționa cu obiectul. Declararea unei clase este similară cu declararea unui nou tip de date, adică se descrie numai cum "arată" și ce "știe" să facă un obiect de tipul (clasa) respectiv, fără să se creeze un astfel de obiect. Când construiți un obiect dintr-o clasă, se spune că ați creat o instanță a clasei.

După cum ați văzut, tot codul pe care îl scrieți în Java se află în interiorul unei clase. Biblioteca standard Java furnizează câteva mii de clase pentru scopuri atât de diverse precum proiectarea interfeței cu utilizatorul, datele și calendarele și programarea în rețea. Cu toate acestea, în Java mai trebuie să vă creați propriile clase pentru a descrie obiectele domeniului problematic al aplicației dvs.

Încapsularea (ascunderea informațiilor) este un concept cheie în lucrul cu obiecte. În mod formal, încapsularea combină pur și simplu datele și comportamentul într-un singur pachet și ascunde detaliile de implementare de utilizatorii obiectului. Biții de date dintr-un obiect se numesc câmpuri de instanță ale acestuia, iar procedurile care operează asupra datelor sunt numite metode ale acestuia. Un obiect specific care este o instanță a unei clase va avea valori specifice ale câmpurilor de instanță ale acesteia. Setul format din acele valori reprezintă starea curentă a obiectului. Ori de câte ori invocați o metodă pe un obiect, starea acesteia se poate schimba. Cheia pentru ca încapsularea să funcționeze este ca metodele să nu acceseze direct câmpurile de instanță dintr-o clasă diferită de a lor. Programele ar trebui să interacționeze cu datele obiectului numai prin metodele obiectului. Încapsularea este modalitatea de a da unui obiect comportamentul său de „cutie neagră”, care este cheia reutilizării și fiabilității. Aceasta înseamnă că o clasă poate schimba total modul în care își stochează datele, dar atâta timp cât continuă să folosească aceleași metode pentru a manipula datele, nici un alt obiect nu va ști sau îi pasa.

Cea mai simplă formă de definire a unei clase în Java este următoarea:

```
class NumeClasa{  
    Câmpuri membru
```

Când începeți să vă scrieți propriile clase în Java, un alt principiu al OOP va face acest lucru mai ușor: Clasele pot fi construite prin **extinderea** altor clase. Java, de fapt, vine cu o „superclasă cosmică” numită Object. Toate celelalte clase extind această clasă. Când extindeți o clasă existentă, noua clasă are toate proprietățile și metodele clasei pe care o extindeți. Apoi furnizați noi metode și câmpuri de date care se aplică numai noii clase. Conceptul de extindere a unei clase pentru a obține o altă clasă se numește moștenire.

Pentru a lucra cu POO, ar trebui să puteți identifica trei caracteristici cheie ale obiectelor:

- Comportamentul obiectului — ce puteți face cu acest obiect sau ce metode îi puteți aplica?
- Starea obiectului - cum reacționează obiectul când le invocați metode?
- Identitatea obiectului - cum se distinge obiectul de alții care pot avea același comportament și stare?







Toate obiectele care sunt **instanțe** ale aceleiași clase împărtășesc o asemănare de familie, susținând același comportament. Comportamentul unui obiect este definit de metodele pe care le puteți apela. Apoi, fiecare obiect stochează informații despre cum arată în prezent. Aceasta este starea obiectului. Starea unui obiect se poate schimba în timp, dar nu spontan. O modificare a stării unui obiect trebuie să fie o consecință a apelurilor de metodă. Cu toate acestea, starea unui obiect nu îl descrie complet, deoarece fiecare obiect are o identitate distinctă. De exemplu, într-un sistem de procesare a comenzilor, două comenzi sunt distincte chiar dacă solicită articole identice. Observați că obiectele individuale care sunt instanțe ale unei clase diferă întotdeauna în identitatea lor și de obicei diferă în starea lor. Aceste caracteristici cheie se pot influența reciproc. De exemplu, starea unui obiect poate influența comportamentul acestuia.

Într-un program procedural tradițional, începeți procesul din partea de sus, cu funcția principală. Când proiectați un sistem orientat pe obiecte, nu există „top”, iar noii veniți la OOP se întreabă adesea de unde să înceapă. Răspunsul este: identificați-vă clasele și apoi adăugați metode la fiecare clasă. O regulă simplă în identificarea claselor este să căutați substantive în analiza problemei. Metodele, pe de altă parte, corespund verbelor.

De exemplu, într-un sistem de procesare a comenzilor, unele substantive sunt: Articol, Comandă, Adresă de livrare, Modalitate de plată, Cont, etc. Aceste substantive pot duce la clasele Articol, Comandă și așa mai departe. În continuare, căutați verbe. Articolele sunt adăugate la comenzi. Comenzile sunt expediate sau anulate. Plățile se aplică comenzilor. Cu fiecare verb, cum ar fi „adăugați”, „transmite”, „anulați” sau „aplicați”, identificați obiectul care are responsabilitatea majoră pentru realizarea acestuia. De exemplu, atunci când un articol nou este adăugat la o comandă, obiectul de comandă ar trebui să fie cel responsabil, deoarece știe cum stochează și sortează articolele. Adică, adaugă ar trebui să fie o metodă a clasei Comandă care ia un obiect Articol ca parametru. Desigur, „substantivul și verbul” nu este decât o regulă generală; numai experiența vă poate ajuta să decideți care substantive și verbe sunt cele importante atunci când vă construiți cursurile.

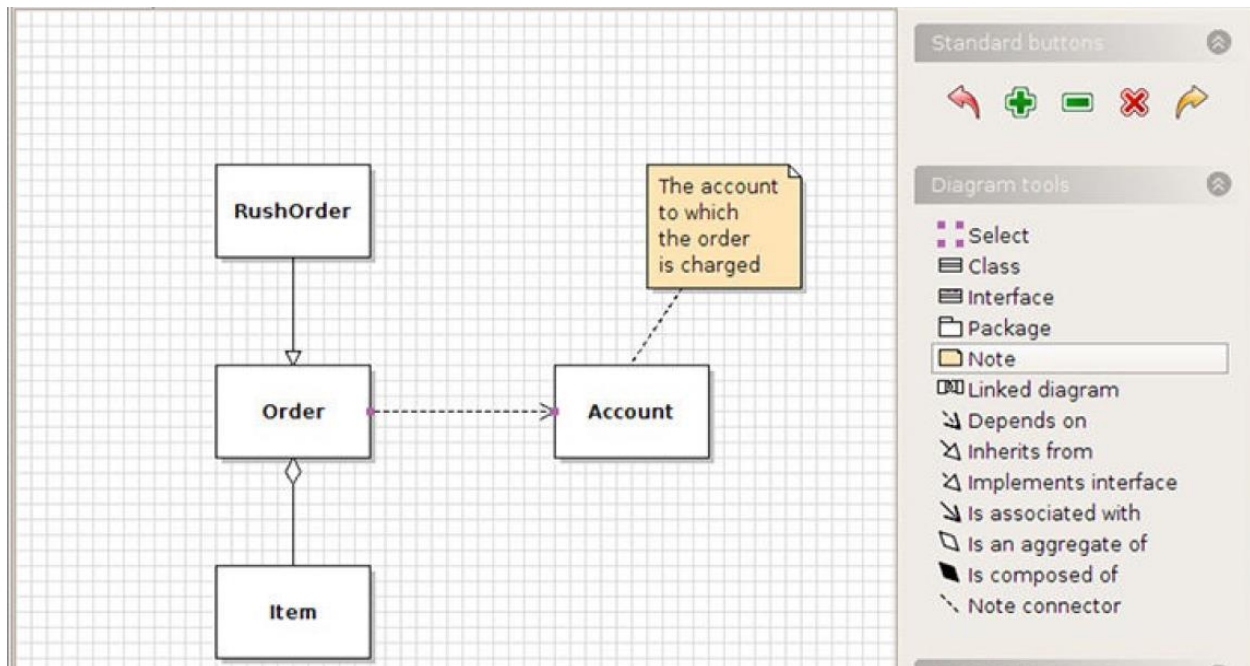
Cele mai frecvente relații între clase sunt: dependență (utilizează o/un), agregarea (deține o/un) și moștenirea (este o/un). **Relația de dependență** sau "utilizează o/un" este cea mai evidentă și, de asemenea, cea mai generală. De exemplu, clasa Comandă folosește clasa Cont deoarece obiectele Comandă trebuie să acceseze obiectele Cont pentru a verifica starea creditului. Dar clasa Articol nu depinde de clasa Cont, deoarece obiectele Articol nu trebuie să se îngrijoreze niciodată de conturile clienților. Astfel, o clasă depinde de o altă clasă dacă metodele folosesc sau manipulează obiecte din acea clasă. Încercați să minimizați numărul de clase care depind unele de altele. Ideea este că, dacă o clasă A nu este conștientă de existența unei clase B, nu este, de asemenea, preocupată de orice modificări aduse B. (Și asta înseamnă că modificările aduse B nu introduc erori în A.) În terminologia ingineriei software, doresc să minimizeze cuplarea dintre clase.

Agregarea, sau relația "are o/un", este ușor de înțeles deoarece este concretă; de exemplu, un obiect Comandă conține obiecte Articol. A conține înseamnă că obiectele din clasa A conțin obiecte din clasa B.

Relationship	UML Connector
Inheritance	
Interface implementation	
Dependency	
Aggregation	
Association	
Directed association	

Notății UML frecvente pentru reprezentarea relațiilor dintre clase

Moștenirea, sau relația "este o/un", exprimă o relație între o clasă mai specială și o clasă mai generală. De exemplu, o clasă ComandăRapidă moștenește clasa Comandă. Clasa specializată ComandăRapidă are metode speciale pentru gestionarea priorităților și o metodă diferită pentru calcularea taxelor de transport, dar celelalte metode, cum ar fi adăugarea de articole și facturarea, sunt moștenite de la clasa Comandă. În general, dacă clasa A extinde clasa B, clasa A moștenește metode din clasa B, dar are mai multe capacități. Mulți programatori folosesc notația UML (Unified Modeling Language) pentru a desena diagrame de clasă care descriu relațiile dintre clase.



Limbajul Java permite **supraîncărcarea (overloading)** metodelor. În aceeași clasă, se pot defini mai multe metode cu același nume, care diferă prin *semnătură* (numărul și tipul parametrilor). Existența acestui mecanism este util pentru a defini comportări diferite la primirea aceluiași mesaj (numele metodei) în funcție de modul (contextul) de apelare.

Pentru a lucra cu obiecte, mai întâi le construiești și specificezi starea lor inițială. Pentru a lucra cu obiecte, mai întâi le construiești și specificezi starea lor inițială. Apoi aplici metode obiectelor. Utilizând definiția clasei, se pot face instanțieri de obiecte având comportarea descrisă de către clasă. La creare se face alocarea de memorie pentru câmpuri și se execută prelucrările indicate de constructorul clasei. Constructor este o metoda care are același nume cu clasa respectivă și este apelat implicit la crearea obiectului. **Singurul context în care un constructor poate să fie referit (apelat ca metodă) este din alt constructor al aceleiași clase**, dar acest lucru se poate realiza numai în anumite condiții speciale. Ca și în C++, constructorii nu au rezultate și, ca urmare, nu se specifică tipul rezultatului în definirea lor.

```
public class Computer
{
    public String processor;
    public int memory;
}
```

În clasa *Computer* au fost declarate variabilele *processor* și *memory*. Prima variabilă este de tip referință la un obiect de tip *String*, iar a doua este de tipul predefinit *int*. Dacă nu se prevede altceva, (prin intermediul unui constructor, de exemplu) inițializarea celor două variabile se face cu *null* pentru referință (o valoare care indică faptul că variabila respectivă nu referă un obiect) și respectiv zero.

Deoarece nu a fost prevăzut nici un mecanism pentru a face acces la aceste atribute (variabile membre) prin intermediul unor metode, deocamdată variabilele au fost declarate cu atributul public (acest tip de atribut, numit și modificador de acces, precizează domeniul de valabilitate al atributelor respective). În mod corespunzător, variabilele *processor* și *memory* sunt accesibile din orice metodă, din orice clasă.

Se poate declara acum o referință în care se poate obține o referință la un obiect de tip *Computer*:

```
Computer pc;
```

Se creează o referință la un obiect de tip *Computer*, fără să se creeze și un obiect instanță pentru clasa *Computer*. În urma declarării, referința este inițializată cu *NULL*.

Pentru a crea un obiect se va executa instrucțiunea:

```
pc = new Computer();
```

Constructorii

La crearea obiectului, s-a considerat că nu sunt necesare prelucrări speciale, în mod corespunzător se consideră că se execută implicit numai un *constructor* al superclasei (în cazul nostru se creează un obiect de tip *Object*) și se adaugă câmpurile curente (*processor* și *memory*) la obiectul astfel construit.

Dacă la instanțierea obiectului sunt necesare prelucrări specifice clasei curente (nu superclasei), trebuie să se definească un **constructor**. Un constructor este o metodă specială care este apelată la crearea obiectului. Ca și în C++, metoda are același nume ca și clasa căreia îi aparține și nu întoarce nici un rezultat.

Un constructor nu poate să fie apelat din alte metode, cu excepția altor constructori (dar utilizând notații speciale).

Dacă într-o clasă nu se definește nici un constructor (ca în prima declarare a clasei *Computer*), se consideră că există un constructor fără argumente (numit și *constructor implicit*) al cărui corp conține numai un apel al constructorului, implicit al superclasei. Dacă într-o clasă se definește cel puțin un constructor, constructorul fără argumente nu mai este furnizat automat.

Constructorii nu pot avea modificatori de acces ca *abstract*, *native*, *static*, *synchronized* sau *final*.

```

public class Computer
{
    public String processor;
    public int memory;

    Computer()
    {
    }

    Computer( String p)
    {
        processor = p;
    }
}

```

În exemplul de mai sus au fost definiți doi constructori, care diferă prin numărul de argumente. În funcție de apel, se va utiliza un constructor sau altul. Astfel, dacă se execută o instrucțiune ca:

```
Computer pc = new Computer("Intel core i5-7400");
```

se va utiliza primul constructor (constructorul cu argument) care creează un obiect de tip *Computer* cu numele "Intel core i5-7400".

Dacă se execută instrucțiunea:

```
Computer pc = new Computer();
```

atunci se va apela al doilea constructor (constructorul fără argumente).

Cel de al doilea constructor nu inițializează nici un câmp. Dacă cel de-al doilea constructor nu ar fi fost specificat, atunci pentru clasa *Computer* nu ar exista un constructor fără argumente (deoarece există deja un constructor definit). Selecția constructorilor se face în faza de compilare pe baza semnăturii apelului.

Pentru a referi câmpurile obiectului, în cadrul metodelor non-stactice dintr-o clasă se poate utiliza referința generică **this**, care desemnează obiectul curent. Astfel, pentru a referi câmpul *nume* în corpul unei metode se poate folosi numele *nume* sau notația *this.nume* (atunci când datorită definirii unei variabile locale sau a unui parametru cu același nume, domeniul de valabilitate al câmpului respectiv este ascuns).

Utilizarea notației bazate pe *this* este necesară și în alte contexte, de exemplu dacă este necesară transmiterea obiectului curent ca argument în invocarea unei metode (eventual pentru un alt obiect).

Într-un constructor se poate apela ca primă instrucțiune un constructor al clasei curente sau un constructor al clasei pe care clasa curentă o extinde. Este singura poziție din definiția unui constructor în care poate să apară un apel al unui alt constructor. Un astfel de apel apare sub forma:

`this(listaDeArgumente);` // constructor din aceeași clasă

unde *listaDeArgumente* reprezintă lista valorilor parametrilor constructorului invocat.

De exemplu, dacă se dorește stabilirea numelui implicit "x86" pentru orice obiect de tip *Computer*, al doilea constructor din exemplul clasei *Computer* poate să fie redefinit sub formă:

```
Computer()  
{  
    this("x86");  
}
```

A fost posibilă definirea a doi constructori pentru clasa *Computer*, deoarece limbajul Java permite supraîncărcarea metodelor, adică existența mai multor metode cu același nume dar cu semnături diferite.

Modificatori de acces

Pentru clasa *Computer* cele două variabile membre au fost declarate publice, cu alte cuvinte ele pot să fie modificate explicit. Dar, astfel de prelucrări nu sunt în spiritul programării orientate obiect. Ideea ar fi că variabilele membre ale obiectelor trebuie să fie ascunse și modificarea lor să se poată face numai prin intermediul metodelor definite odată cu clasa respectivă.

Să considerăm o altă declarație pentru clasa *Computer* care respectă principiul încapsulării.

```
public class Computer  
{  
    private String processor;  
    private int memory;  
  
    Computer()  
    {  
        this("x86");  
    }  
  
    Computer( String p)  
    {  
        processor = p;  
        memory = 0;  
    }  
}
```

```

void AddMemory(int m)
{
    memory += m;
}

int GetMemory()
{
    return memory;
}

String GetProcessor()
{
    return processor;
}
}

```

Variabilele *processor* și *memory* nu mai sunt accesibile din exterior decât prin intermediul metodelor de acces. Atributele publice și private descriu nivelul de acces la variabilele respective, iar ele nu sunt singurele posibile.

Dacă pentru o clasă se definește cel puțin un constructor (pentru a nu fi considerat un constructor implicit) și **toți constructorii sunt declarați cu atributul *private***, atunci nu se pot crea instanțe ale clasei din afara sa. O astfel de situație poate să fie interesantă dacă, de exemplu, o clasă este definită pentru a declara o serie de variabile globale (câmpuri cu atributul static) sau dacă sunt necesare metode statice care să fie utilizate similar unor funcții C (fără să se facă o instanțiere). Exemplul tipic este clasa *System* pentru care este definit un singur constructor care are atributul *private*. Această clasă conține câmpuri statice cum sunt *in* sau *out* care sunt utilizate fără să existe o instanțiere a clasei *System*.

În exemplul următor este prezentat modul în care se poate, totuși, obține o instanțiere a unui obiect dintr-o clasă pentru care s-a declarat numai un constructor cu atributul *private*. Instanțierea obiectului se face în metoda *Create()* a clasei *Test*. O încercare de instanțiere directă a unui obiect ar produce o eroare în faza de compilare. Apelul metodei statice *Create()* are ca rezultat o referință la un obiect de tipul *Test*. Pentru acest obiect, se pot invoca metode nestatice (*Processing()*).

```

class Test
{
    private static int number = 0;
    private Test()
    {
        number++;
    }
    void Processing()

```



```

    {
        System.out.println("Processing");
    }
    static Test Create()
    {
        return new Test();
    }
}

public class UseTest
{
    public static void main(String args[])
    {
        Test t = Test.Create();
        t.Processing();
    }
}

```

În exemplul de mai sus, variabilele membre au fost declarate cu atributul *private*. Asta înseamnă că variabilele respective nu pot să fie referite decât din metodele clasei respective. Pe lângă modificatorii de acces **private** și **public**, limbajul Java lucrează și cu modificatorii de acces **protected** (asigură acces la nivelul unei ierarhii) și **default** (package - nespecificat) (asigură acces la nivelul unui pachet).

Când implementăm o clasă, facem toate câmpurile de date **private**, deoarece date publice sunt periculoase. Dar cum rămâne cu metodele? Deși majoritatea metodelor sunt publice, metodele private sunt utile în anumite circumstanțe. Uneori, poate doriți să împărtășiți codul pentru calcule în metode auxiliare separate. De obicei, aceste metode de ajutor nu ar trebui să facă parte din interfața publică — s-ar putea să fie și ele aproape de implementarea curentă sau necesită un protocol special sau un ordin de apelare. De aceea se preferă implementarea unor astfel de metode ca private. Pentru a implementa o metodă privată în Java, schimbați pur și simplu cuvântul cheie *public* în *private*. Făcând o metodă privată, nu aveți nici o obligație să o păstrați disponibilă dacă vă schimbați implementarea. Metoda poate fi mai greu de implementat sau inutilă dacă reprezentarea datelor se modifică; însă acest lucru este irelevant. Ideea este că atâta timp cât metoda este privată, designerii clasei pot fi asigurați că nu este niciodată folosit în altă parte, așa că pur și simplu îl pot scăpa. Dacă o metodă este publică, nu puteți renunța pur și simplu la refacerea implementării, deoarece alt cod ar putea apela metoda respectivă.

Puteți defini un câmp de instanță ca **final**. Un astfel de câmp trebuie inițializat când obiectul este construit. Adică, trebuie să garantați că valoarea câmpului a fost setată la finalul fiecărui constructor. După aceea, câmpul poate să nu fie modificat din nou. De exemplu, câmpurile de identificare ale unei clase pot fi declarate ca final deoarece nu se schimbă niciodată după ce obiectul este construit. Modificatorul final este util în special pentru câmpurile al căror tip este primitivă sau o clasă invariabilă. (O clasă este invariabilă (engl. immutable) dacă niciuna dintre metodele sale nu modifică vreodată obiecte, de exemplu - clasa String.)

Pentru clasele modificabile (engl. mutable), modificatorul final poate crea confuzie. De exemplu, să luăm în considerare un domeniu. În cazul lor cuvântul cheie final înseamnă doar că referința la obiect stocată în variabila declarată cu final nu va mai referi niciodată la un alt obiect. În schimb proprietățile obiectului vor putea fi modificate.

În toate programele eșantion pe care le-ați văzut, metoda principală este etichetată cu modificator **static**. Acum suntem gata să discutăm despre semnificația acestui modificator. Termenul „static” are o istorie curioasă. La început, cuvântul cheie static a fost introdus în C pentru a desemna variabilele locale care nu sunt distruse la ieșirea dintr-un bloc. În acest context, termenul „static” are sens: variabila persistă atunci când se iese dintr-un bloc și poate fi folosită atunci când se reintră în blocul respectiv. Apoi static a fost utilizat în C, pentru a desemna variabile globale și funcții care nu pot fi accesate din alte fișiere. Cuvântul cheie static a fost pur și simplu reutilizat pentru a evita introducerea unui nou cuvânt cheie. În cele din urmă, C++ a reutilizat cuvântul cheie pentru o a treia interpretare, fără legătură cu precedentele - pentru a desemna variabile și funcții care aparțin unei clase, dar nu unui obiect anume al clasei. Acesta este același sens pe care îl are cuvântul cheie în Java.

Dacă definiți un **câmp** ca fiind **static**, atunci există doar un astfel de câmp pe clasă. În contrast, fiecare obiect are propria copie a câmpurilor de instanță nestatice. De exemplu, să presupunem că vrem să atribuim fiecărui obiect un număr unic de identificare. Adăugăm un ID câmp de instanță și un câmp static `nextId` în clasa respectivă

```
class Object
{
    private static int nextId = 1;
    private int id;
    . . .
}
```

Fiecare obiect are acum propriul său câmp de id, dar există doar unul câmp `nextId` care este partajat de toate instanțele clasei. Să privim din altă perspectivă: dacă există 1.000 de obiecte, atunci există 1.000 de câmpuri de instanță `id`, câte unul pentru fiecare obiect și un singur câmp static `nextId`. Chiar dacă nu există obiecte, câmpul static `nextId` este prezent. Aceasta aparține clasei, nu oricărui obiect individual.

Variabilele statice sunt destul de rare. Cu toate acestea, **constantele statice** sunt mai frecvente. De exemplu, clasa `Math` definește o constantă statică:

```
public class Math{
    . . .
    public static final double PI = 3,14159265358979323846;
    . . .
}
```

Puteți accesa această constantă în programele dvs. ca `Math.PI`. Dacă cuvântul cheie static ar fi fost omis, atunci `PI` ar fi fost o instanță domeniului clasei de matematică. Adică, veți avea nevoie de un obiect din această clasă pentru a accesa `PI` și fiecare obiect `Math` ar avea propria copie a lui `PI`. O altă constantă statică pe care ați folosit-o de multe ori este `System.out`. Este declarat în clasa `System` după cum urmează:

```
public class System{
    . . .
    public static final PrintStream out = . . . ;
    . . .
}
```

```
}
```

După cum am menționat de mai multe ori, nu este niciodată o idee bună să aveți câmpuri publice, pentru că toată lumea le poate modifica. Cu toate acestea, constantele publice (adică, câmpurile de tip final) sunt în regulă. Deoarece out a fost declarat final, nu puteți realoca alt flux de ieșire către el. S-ar putea să vă întrebați cum o metodă poate modifica valoarea unei variabile finale. Însă metoda setOut este o metodă nativă, neimplementată în Java. Metodele native pot ocoli mecanismele pentru controlul accesului din Java. Această soluție nu este recomandată în implementările proprii.

Metodele statice sunt metode care nu operează asupra obiectelor. De exemplu, metoda pow din clasa Math este o metodă statică. Expresia Math.pow(x, a) calculează puterea x^a . Nu folosește nici un obiect Math pentru a-și îndeplini sarcina. Cu alte cuvinte, nu are nici un parametru implicit. Vă puteți gândi la metodele statice ca la metode care nu au acest parametru (într-o metodă nestatică, parametrul this se referă la parametrul implicit al metodei). O metodă statică a clasei Obiect nu poate accesa câmpul instanței id deoarece nu operează asupra unui obiect. Cu toate acestea, o metodă statică poate accesa un câmp static. Iată un exemplu de astfel de metodă statică:

```
public static int getNextId()
{
    return nextId; // returnează câmpul static
}
...
int n = Obiect.getNextId();
```

Ați fi putut omite cuvântul cheie static pentru această metodă? Da, dar atunci ar trebui să aveți o referință la obiect de tip Employee pentru a invoca metodă. Este legal să folosiți un obiect pentru a apela o metodă statică. De exemplu, dacă ob este un obiect al clasei Obiect, atunci puteți apela ob.getNextId() în loc de Obiect.getNextId(). Totuși, găsim această notație confuză. Metoda getNextId nu se uită deloc la ob atunci când calculează rezultatul. Vă recomandăm să utilizați numele claselor, nu obiecte, pentru a invoca metode statice. Se vor utiliza metode statice în două situații:

- Când o metodă nu trebuie să acceseze starea obiectului, deoarece toate sunt necesare parametrilor sunt furnizați ca parametri expliți (de exemplu: Math.pow).
- Când o metodă trebuie să acceseze doar câmpurile statice ale clasei (exemplu: Obiect.getNextId).

Probleme propuse

Problema 1

Să se definească o clasă *Complex* pentru operații cu numere complexe și să se testeze metodele implementate. Clasa va avea doi constructori astfel:

- cu două argumente (partea reală și imaginară)
- fără argumente

Metodele necesare sunt: adunare, înmulțire, ridicare la putere naturală, *Equals* și *toString* (șir de forma (re,im)).

Problema 2

Să se definească o clasă *Stiva* pentru stive de numere întregi, reprezentate prin vectori intrinseci. Datele clasei:

- un vector de întregi
- indicele elementului din vârful stivei (ultimul introdus)

Metodele clasei:

- Constructor fara parametri (dimensiunea implicită a stivei = 100)
- Constructor cu parametru dimensiunea stivei
- *void Push(int)* : pune un întreg dat pe stivă
- *int Pop()* : scoate elementul din vârful stivei
- *boolean isEmpty()* : verifica daca stiva este goala

Considerați cazurile de stiva plină (la push) și stiva goală (la pop). Metoda statică *main* pentru verificarea operațiilor cu o stivă va fi inclusă în clasa *Stiva*.

Problema 3

Să se definească o clasă *Matrix* pentru operații uzuale cu matrice pătratice de numere reale: adunare, înmulțire, *toString*. Să se scrie un program pentru ridicarea la o putere întreagă a unei matrice pătratice.

Problema 4

Să se definească o clasă *IntSet* pentru o mulțime de numere întregi, care conține un vector intrinsec de întregi.

Metodele clasei:

- Constructor cu argument dimensiune mulțime
- *void Add(int)* : adaugare element, dacă nu există
- *boolean Contains(int)* : verifica dacă un număr dat se află sau nu în mulțime
- *String toString()* : mulțimea de elemente transformată în șir de caractere

Să se scrie un program care creează o mulțime folosind clasa *IntSet* și testează operațiile de mai sus prin adăugări succesive și afișări.

Problema 5

Să se definească o clasă *Graph* pentru grafuri orientate, în care arcele nu au asociat un cost, iar nodurile sunt numerotate de la 1.

Datele clasei (private):

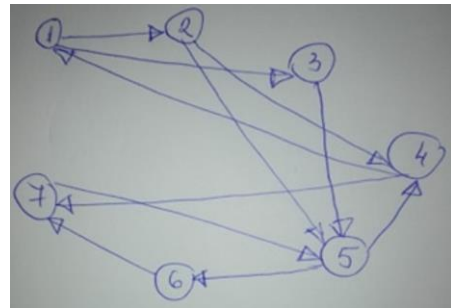
- o matrice cu componente de tip boolean (matricea de adiacență)
- numărul de noduri

Metodele clasei:

- Constructor cu un parametru întreg (numărul de noduri din graf)
- *int Size()* : întoarce numărul de noduri din graf
- *void AddArc(int v, int w)* : adaugă un arc la graf (între v și w)
- *boolean IsArc(int v, int w)* : verifică dacă există arc între nodurile v și w
- *void Print()* : afișarea matricei de adiacență (cu 1 și 0)
- *void Dfs(int v, boolean vazut[])* : vizitare în adâncime din nodul v.

Pentru verificare se va crea un graf prin adăugări succesive de arce. Se va afișa matricea de adiacență a grafului și se va afișa ordinea nodurilor la explorarea în adâncime din fiecare nod al grafului:

```
DFS_recurziv(A, L, M, i)
//A - matricea de adiacenta
//L - lista nodurilor care formeaza parcurgerea din i
//M - vector de etichete vizitat/nevizitat - initial 0
//i - nodul de start
insert(L, i)
M[i] := 1
for k = 1 to n do
    if A[i,k] = 1 then
        if M[k] = 0 then
            DFS_recurziv(A, L, M, k)
        end-if
    end-if
end-for
```



```
C:\WINDOWS\system32\cmd.exe

Matricea de adiacenta:
0 1 1 0 0 0 0
0 0 0 1 1 0 0
0 0 0 0 1 0 0
1 0 0 0 0 0 1
0 0 0 1 0 1 1
0 0 0 0 0 0 1
0 0 0 0 1 0 0

Traversals:
DFS rec: 0 1 3 6 4 5 2
DFS iter: 0 1 3 6 4 5 2
BFS iter: 0 1 2 3 4 6 5
BFS rec: 0 1 2 3 4 6 5

Press any key to continue . . .
```