Programare orientată pe obiecte

1. Despre curs
2. Concepte şi paradigme în POO
3. Mediul Java
4. Variabile și tipuri
5. Operatori artimetici și expresii

Cadre didactice

- Curs
  - Ion Giosan
    - E-mail: Ion.Giosan@cs.utcluj.ro
    - Web page: http://users.utcluj.ro/~igiosan/teaching_poo.html
- Laborator
  - Ion Giosan (grupa 6)
  - Angela Modrîngă, Bogdan Potra (grupele 7, 8)
  - Sergiu Nicula, Cristina Rădulescu (grupa 9)

Conținutul cursului

- Concepte şi paradigme în programarea orientată pe obiecte
- Abstracţiuni şi tipuri de date abstracte
- Caracteristicile limbajului Java
- Tipurile de date primitive şi structurile de control în Java
- Clase şi obiecte. Pachete
- Moştenirea şi polimorfismul
- Interfeţe Java
- Reprezentarea în UML a claselor și legăturilor dintre ele
- Excepţii şi tratarea lor
- Colecții Java
- Interfețe utilizator (*GUI*s) în Java
- Testarea și depanarea programelor
- Introducere în sistemul I/E Java
- Fire de lucru (*threads*) în Java

Evaluare. Referinţe

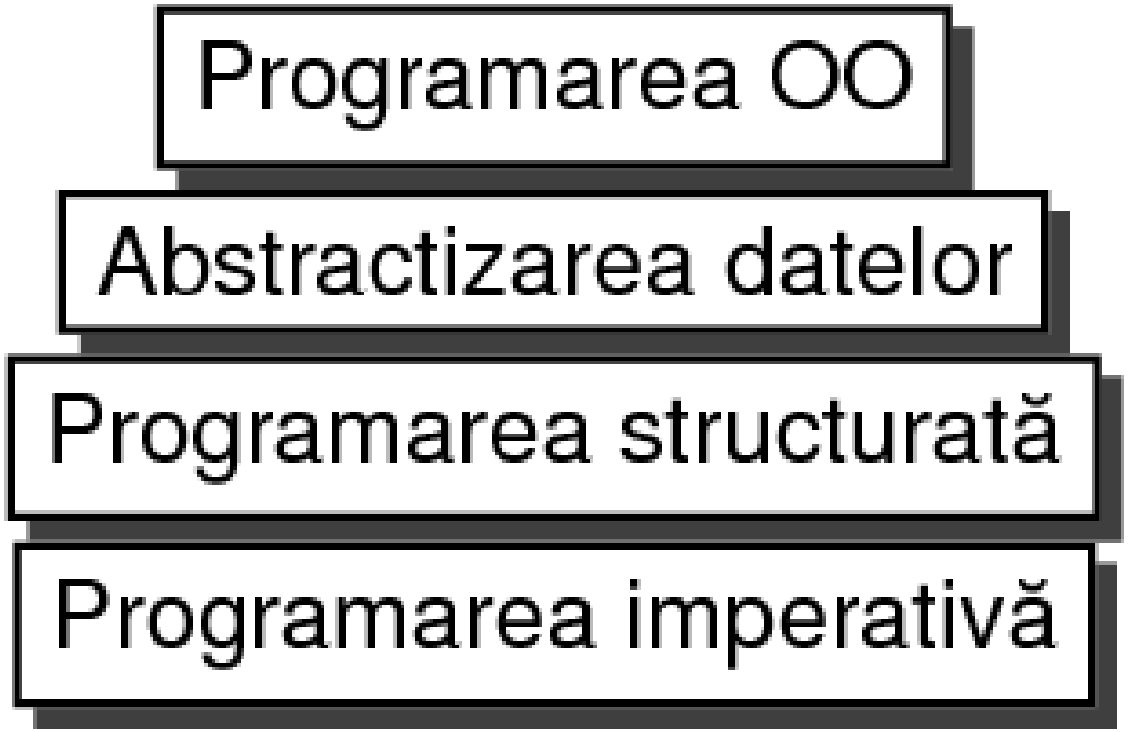
- Evaluare
  - Nota finală = 0.4*Laborator + 0.5*ExamenScris + 0.1*TesteCurs
    - Obligatoriu Laborator >=5
    - Obligatoriu ExamenScris >=5
  - Bonusuri
    - Prezență sporită la cursuri
- Referinţe
  - **Bruce Eckel, *Thinking in Java*, 4th edition, Prentice Hall, 2006**
  - **Kathy Sierra, Bert Bates, *SCJP Sun Certified Programmer for Java 6*, Mc Graw Hill, 2008**
  - Tutoriale Java și Documentaţia Java de la Oracle
  - Tutoriale UML introductive
  - Documentaţia Eclipse

Paradigme de programare

- **Paradigmă de programare**
  - Un model care descrie esenţa şi structura computaţiei
  - Un stil fundamental de a programa
  - Oferă şi determină viziunea pe care o are programatorul asupra execuţiei programului

    Exemple:
    - în programarea funcţională un program poate fi conceput ca fiind o secvenţă de evaluări de funcţii, fără stări
    - în POO, programatorii pot concepe programele ca fiind o colecţie de obiecte care interacţionează

Paradigme de programare

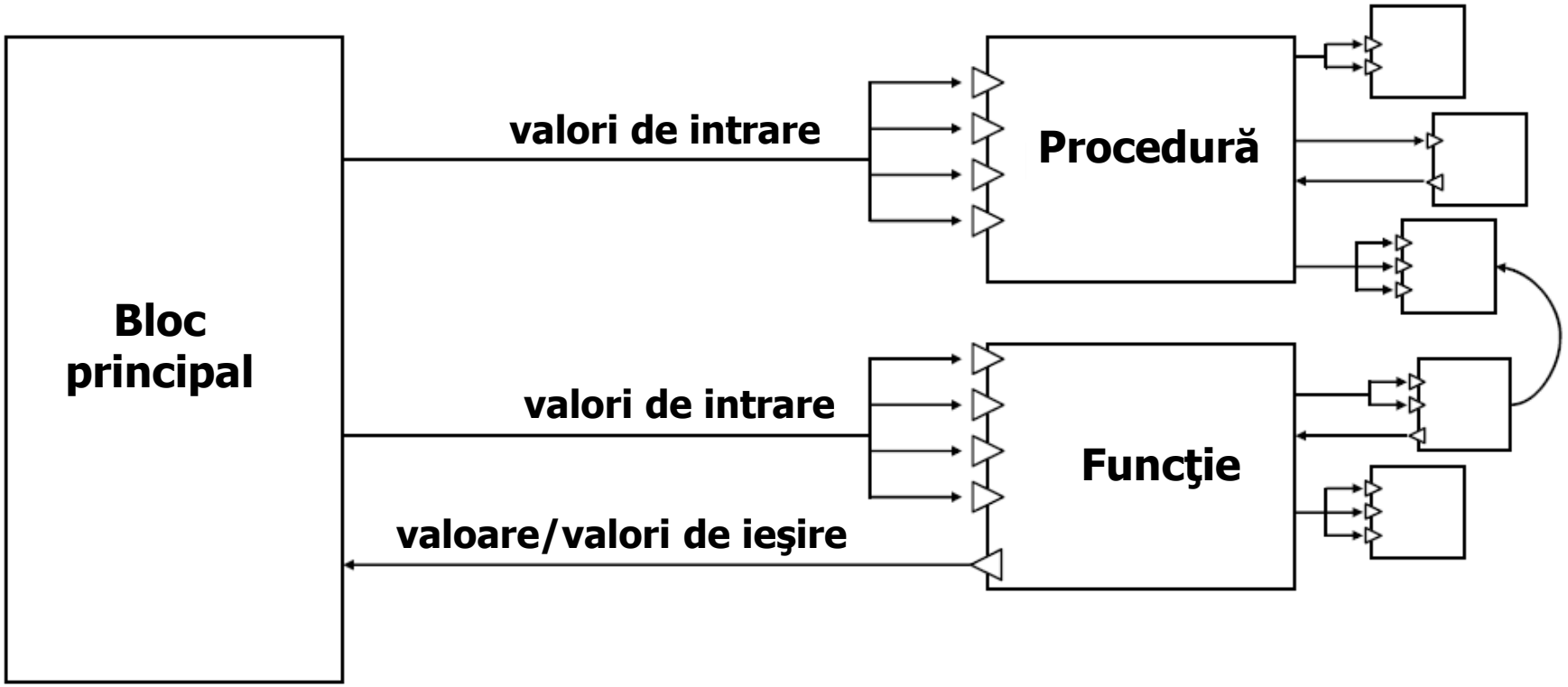


Programarea imperativă

- Un calculator în modelul tradiţional von Neumann
  - Unitate centrală de prelucrare
  - Memorie
  - Efectuează secvenţe de instrucţiuni atomice care accesează, operează asupra valorilor stocate la locaţii de memorie adresabile individual şi le modifică
- O computaţie este o serie de operaţii aritmetice şi de efecte laterale, cum sunt atribuirile sau transferurile de date care modifică starea unităţii de stocare, intrarea sau ieşirea
- Este de subliniat importanţa atribuirilor şi a variabilelor pe post de containere pentru paradigma imperativă
- Exemple de limbaje: Fortran, Pascal, C, Ada.

Programarea structurată

Program procedural = definiții de date și apeluri de funcții/proceduri



Programarea structurată

- Abstractizarea operaţiilor
  - Structura unui modul
    - Interfaţa
      - Date de intrare
      - Date de ieşire
      - Descrierea funcţionalităţii
    - Implementarea
      - Date locale
      - Secvenţe de instrucţiuni
  - Sintaxa limbajului
    - Organizarea codului în blocuri de instrucţiuni
    - Definiţii de funcţii şi proceduri
    - Extinderea limbajului cu noi operaţii
    - Apeluri la proceduri şi funcţii noi

Beneficiile programării structurate

- Uşurează dezvoltarea software
  - Evită repetarea realizării aceluiaşi lucru
  - Munca de programare este descompusă în module independente
  - Proiectare Top-down: descompunerea în subprobleme
- Facilitează întreţinerea software
  - Codul este mai uşor de citit
  - Independenţa modulelor
- Favorizează reutilizarea software

Programarea structurată. Exemplu

```
int main()
{
   double u1, u2, m;
   u1 = 4;
   u2 = -2;
   m =sqrt (u1*u1 + u2*u2);
   printf("%lf\n", m);
   return 0;
}
```

```
double module(double u1, double u2)
{
   double m;
   m =sqrt (u1*u1 + u2*u2);
   return m;
}
int main()
{
   printf("%lf\n", module(4, -2));
   return 0;
}
```

Abstractizarea datelor

- Impunerea unei separări clare între *proprietăţile abstracte* ale unui *tip de dată* şi *detaliile concrete* ale *implementării* lui
  - Proprietăţi abstracte: acelea care sunt vizibile codului client care foloseşte tipul de dată – *interfaţa* cu tipul de dată
  - Implementarea concretă: este păstrată în totalitate privată şi ea se poate într-adevăr schimba, spre exemplu pentru a încorpora îmbunătăţiri ale performanţelor în timp

Tipuri abstracte de date

- Abstractizarea datelor + abstractizarea operaţiilor
  - Un tip de dată abstract:
    - **Structură de date** care stochează informaţii pentru a reprezenta un anumit concept
    - **Funcţionalitate:** set de operaţii care pot fi aplicate tipului de dată
  - Sintaxa limbajului
    - Modulele sunt asociate tipurilor de date
    - Sintaxa nu este neapărat nouă faţă de programarea modulară

Exemplu de tip de dată abstract în C

```
struct vector {
   double x;
   double y;
}
void construct (vector *v, double v1, double v2)
{
   v->x = v1;
   v->y = v2;
}
double module(vector v)
{
   double m;
   m = sqrt (v.x*v.x + v.y*v.y);
   return m;
}
```

```
int main()
{
   vector v;
   construct(&v, 4, 2);
   printf("%lf\n", module(v));
   return 0;
}
```

Extensibilitatea tipului de dată abstract

```
...

double product(vector v, vector w) {
   return v.x*w.x + v.y*w.y;
}
int main() {
   vector v;
   construct(&v, 4, 2);
   construct(&w, -1, 7);
   printf("%lf\n", product(v, w));
   return 0;
}
```

Beneficiile tipurilor de date abstracte

- Conceptele din domeniul real sunt reflectate în cod
- *Încapsulare:* complexitatea internă, datele şi detaliile operaţiilor sunt ascunse
- Utilizarea tipului de dată este *independentă de implementarea sa* internă
- Oferă o mai mare modularitate
- Sporeşte uşurinţa întreţinerii şi reutilizării codului

Paradigma orientării pe obiecte

- Paradigma programării structurate a avut iniţial succes (1975-85)
  - Dar a început să eşueze la produse mai mari (> 50,000 LOC)
  - Avea probleme de întreţinere post-livrare (astăzi această întreţinere necesită, de la 70 la 80% din efortul total)
  - Motivul: Metodele structurate sunt fie
    - orientate pe operaţii
    - orientate pe atribute
    - …**dar nu amândouă**

Paradigma orientării pe obiecte

- O simulare a domeniului unei probleme prin abstractizarea informaţiilor de comportament şi stare din obiecte din lumea reală

- Conceptele POO
  - Clase și Obiecte
  - Abstractizare și Încapsulare
  - Transmitere de mesaje
  - Moştenire și Polimorfism

Paradigma orientării pe obiecte

- POO consideră că atât atributele cât şi operaţiile au importanţă egală
- O viziune simplistă a unui obiect poate fi:
  - Obiect = componentă software care incorporează atât atributele cât şi operaţiile care se pot efectua asupra atributelor şi care suportă moştenirea
- Exemplu:
  - Cont bancar
    - Date: soldul contului
    - Acţiuni: depune, retrage, determină soldul

Paradigma orientării pe obiecte

- Totul este reprezentat prin obiecte

  (obiect = o variabilă mai specială ce încapsulează atât date cât și operații cu aceste date)
- Obiectele comunică între ele prin trimitere/ primire de mesaje (mesaj = apel de metodă)
- Obiectele au propria lor memorie
- Orice obiect are un tip ⇔ orice obiect e o instanță a unei clase (unde 'clasa' este sinonim cu 'tip')
- Toate obiectele de un anumit tip pot trimite sau primi aceleași mesaje

Punctele tari ale POO

- Ascunderea informaţiei => întreţinerea post-livrare este mai sigură
  - Şansele apariţiei erorilor regresive sunt reduse (în software nu se repetă erori cunoscute)
- Dezvoltarea este mai uşoară
  - Obiectele au în general corespondente fizice =>
    - Simplifică modelarea (un aspect cheie al POO)

Punctele tari ale POO

- Obiectele bine proiectate sunt unităţi independente
  - Tot ce se referă la obiectul real modelat este în obiect — *încapsulare*
  - Comunicarea se face prin schimb de *mesaje*
  - Această independenţă promovează reutilizarea codului

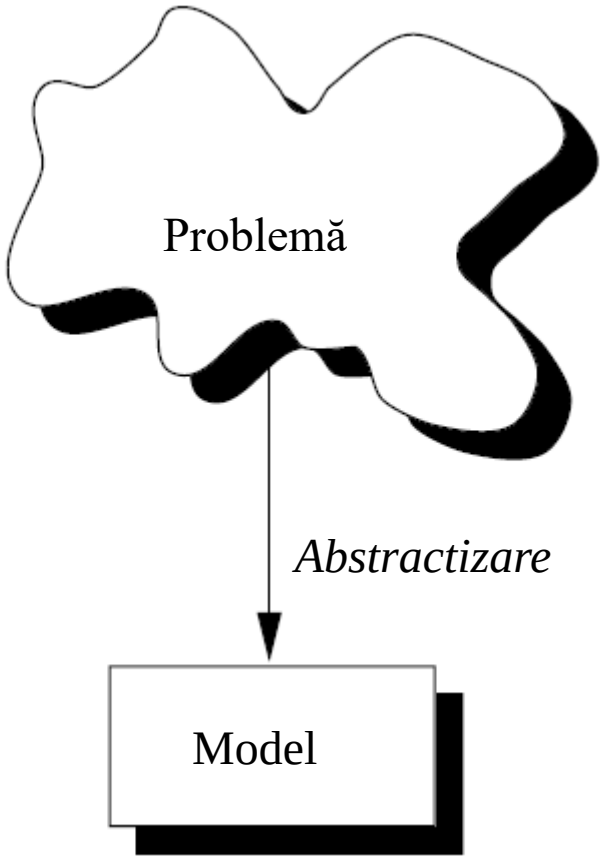
Programarea orientată pe obiecte

- Oferă
  - suport sintactic pentru tipurile de date abstracte
  - facilităţi asociate cu ierarhiile de clase
- Schimbă punctul de vedere: programele sunt apendice ale datelor
- Introduc un concept nou: ***obiect*** = tip de dată abstract cu *stare* (atribute) şi *comportament* (operaţii)

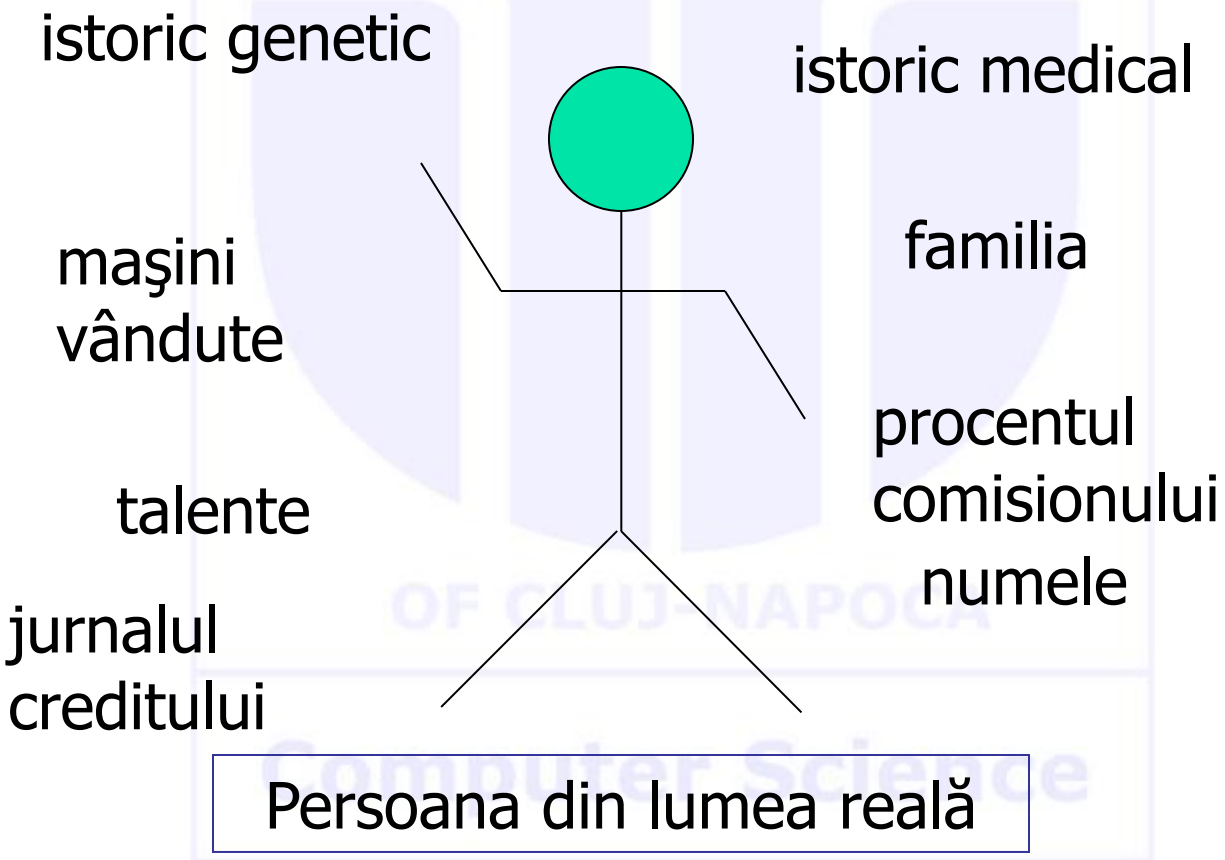
Concepte POO

- *Abstractizare* = proces de filtrare a detaliilor neimportante ale obiectului astfel încât să rămână doar caracteristicile importante

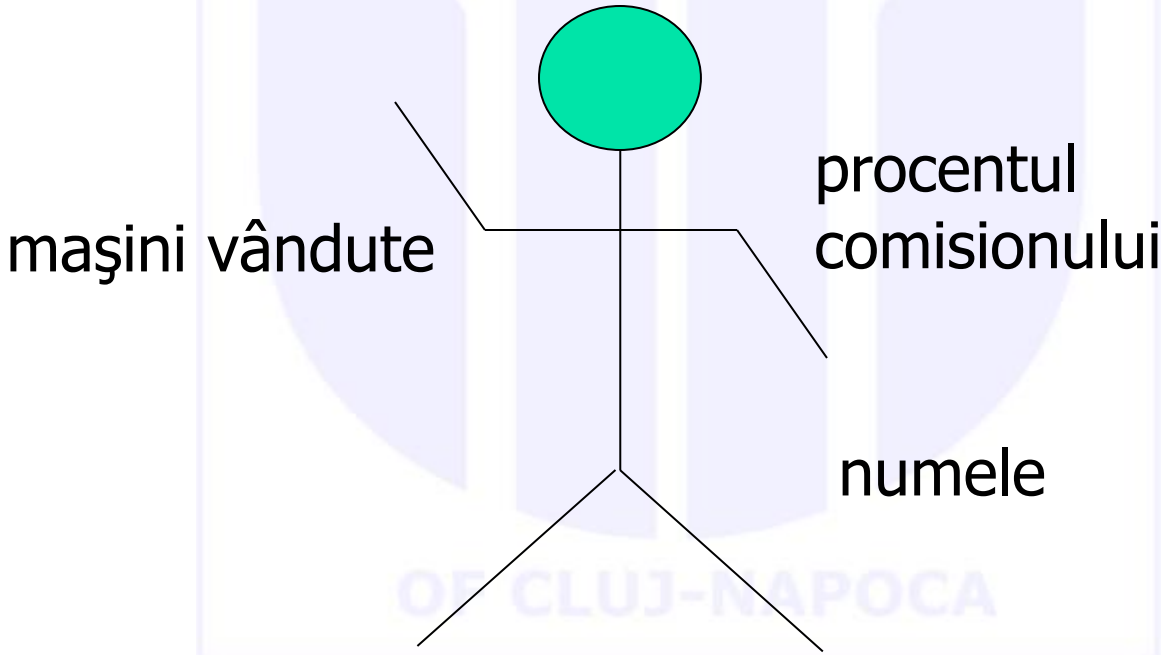
- Ne ocupăm doar de datele care prezintă interes pentru problema noastră



Abstractizare. Exemplu

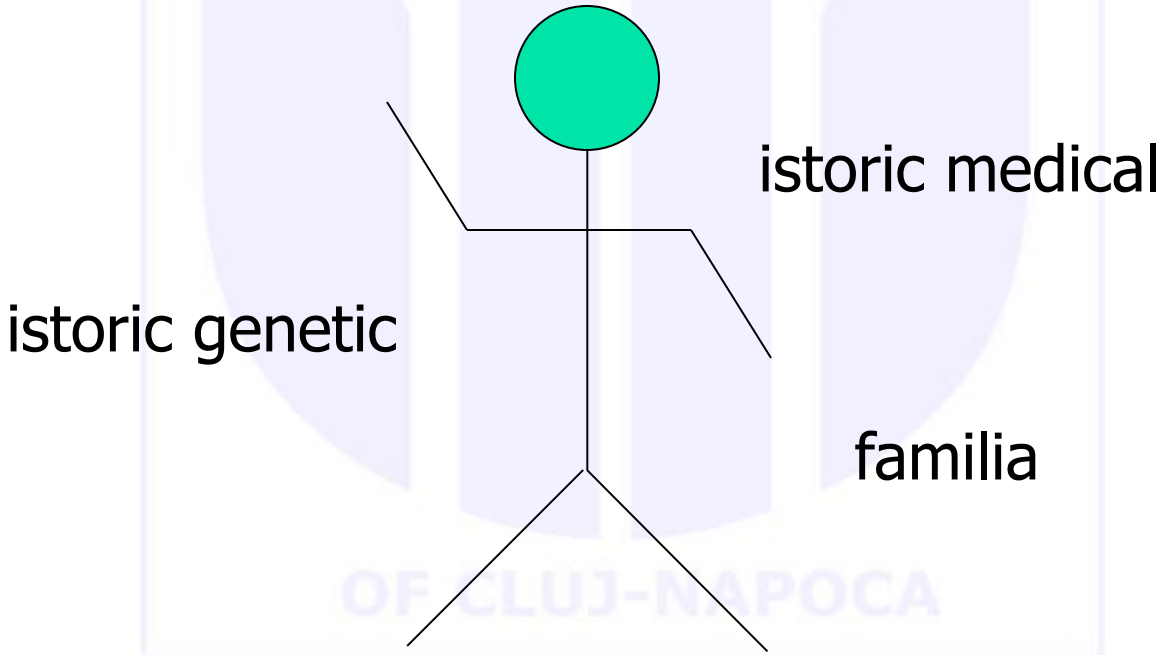


Abstractizare. Exemplu



Abstractizarea unui tip `PersoanaVinzator` pentru un Sistem de urmărire a vânzărilor

Abstractizare. Exemplu



Abstractizarea unui tip **`Pacient`** dintr-o baze de date medicală

Ce sunt obiectele software?

- Blocurile de construcţie a sistemelor software
  - Program = colecţie de obiecte care interacţionează
  - Obiectele cooperează pentru a finaliza o sarcină
    - pentru aceasta, ele comunică trimiţându-si "mesaje" unul altuia
- Obiectele modelează lucruri *tangibile*
  - Persoană
  - Bicicletă
  - Cal
  - Bancă etc.

Ce sunt obiectele software?

- Obiectele modelează *lucruri conceptuale*
  - întâlnire
  - dată calendaristică
- Obiectele modelează *procese*
  - aflarea drumului printr-un labirint
  - sortarea unui pachet de cărţi de joc
- Obiectele au
  - *proprietăţi*: trăsături (caracteristici) care descriu obiectele
  - *capabilităţi*: ce pot face, cum se comportă

Proprietăţile obiectului: starea

- *Proprietăţile* : determină cum acţionează un obiect
  - Pot fi constante (nu se schimbă) sau variabile
  - Pot fi ele însele obiecte — pot primi mesaje
  - Ex. Capacul *borcanului cu gem* şi gemul în sine sunt obiecte
- Proprietăţile pot fi:
  - atribute: lucruri care ajută la descrierea unui obiect
  - componente: lucruri care sunt "parte a" unui obiect
  - asocieri: lucruri despre care ştie un obiect, dar care nu sunt parte a acelui obiect

Proprietăţile obiectului: starea

- *Stare*: colecţie a tuturor proprietăţilor obiectului; se schimbă dacă o proprietate se schimbă
  - unele nu se schimbă, d.e. volanul unei maşini
  - altele se schimbă, d.e. culoarea maşinii
- Exemplu: proprietăţile *borcanelor cu gem*
  - atribute: culoare, material, miros
  - componente: capac, container, etichetă
  - asocieri: un *borcan cu gem* poate fi asociat cu încăperea în care se află

Capabilităţile obiectelor: acţiuni

- Obiectele au *capabilităţi* (*comportamente*) care le permit să efectueze acţiuni specifice
  - obiectele sunt deştepte — ele "ştiu" cum să facă anumite lucruri
  - un obiect face ceva *doar dacă un alt obiect îi spune* să-şi folosească una dintre capabilităţi
- Capabilităţile pot fi:
  - *constructori*: stabilesc starea iniţială a proprietăţilor obiectului
  - actiuni: modifică proprietăţile obiectului
  - interogări: furnizează răspunsuri bazate pe proprietăţile obiectului

Capabilităţile obiectelor: acţiuni

- Exemple: *borcanele cu gem* sunt capabile să efectueze acţiuni specifice
  - constructor: să fie creat
  - actiuni: adaugă/golește gem
  - interogări: răspunde dacă este închis sau deschis capacul, dacă borcanul este plin sau gol

Clase şi instanţe

- Concepţia noastră curentă: fiecare obiect corespunde direct unui *anumit* obiect din realitate, d.e., un atom sau un automobil anume
- Dezavantaj: mult prea nepractic să lucrăm cu obiecte în acest fel deoarece
  - ele pot fi infinit de multe
  - nu dorim să descriem fiecare individ separat, deoarece indivizii au multe lucruri în comun
- Clasificarea obiectelor scoate în evidenţă ce este comun între mulţimi de obiecte similare
  - mai întâi să *descriem ce este comun*
  - apoi să "ştampilăm" oricâte copii

Clase ale obiectelor

- *Clasa unui obiect*
  - categoria obiectului
  - defineşte capabilităţile şi proprietăţile comune unei mulţimi de obiecte individuale
    - toate *borcanele cu gem* se pot deschide, închide şi goli
  - defineşte un şablon pentru crearea de *instanţe de obiect*
    - unele *borcane cu gem* pot fi din plastic, pot fi colorate, de o anumită mărime etc.

Clase ale obiectelor

- Clasele implementează capabilităţile ca *metode*
  - secvenţe de instrucţiuni în Java
  - obiectele cooperează trimiţând mesaje altor obiecte
  - fiecare mesaj "invocă o metodă"
- Clasele implementează proprietăţile ca *variabile instanţă*
  - locaţie de memorie alocată obiectului, care poate păstra o valoare care se poate schimba

Instanţe de obiecte

- *Instanţele de obiecte* sunt obiecte individuale
  - realizate din şablonul clasei
  - o clasă poate reprezenta un număr nedefinit de instanţe de obiect
  - realizarea unei instanţe de obiect constituie *instanţierea* obiectului respectiv
- Prescurtare:
  - clasă: clasa obiectului
  - instanţă: instanţa obiectului (a nu se confunda cu variabilele instanţă)

Instanţe de obiecte

- Instanţe diferite ale, d.e., clasei **`BorcanCuGem`** pot avea:
  - culoare şi poziţie diferită
  - diverse tipuri de gem în interior
- Astfel că, *variabilele instanţă* ale lor au valori diferite
  - Notă: instanţele de obiect conţin variabile instanţă — două moduri de folosire diferite, dar înrudite, a cuvântului *instanţă*
- Instanţele individuale au identităţi individuale
  - aceasta permite altor obiecte să trimită mesaje unui obiect dat
  - fiecare instanţă este unică, chiar dacă are aceleaşi capabilităţi
    - Exemplu: clasa studenţilor de la acest curs

Mesaje pentru comunicarea între obiecte

- Instanţele nu sunt izolate — ele trebuie să comunice cu altele pentru a-şi realiza sarcina
  - proprietăţile le permit să ştie despre alte obiecte
- Instanţele trimit mesaje una alteia pentru a invoca o capabilitate (adică, pentru a executa o sarcină)
  - metoda reprezintă codul care implementează mesajul
  - spunem "apelează metoda" în loc de "invocă capabilitatea"

Mesaje pentru comunicarea între obiecte

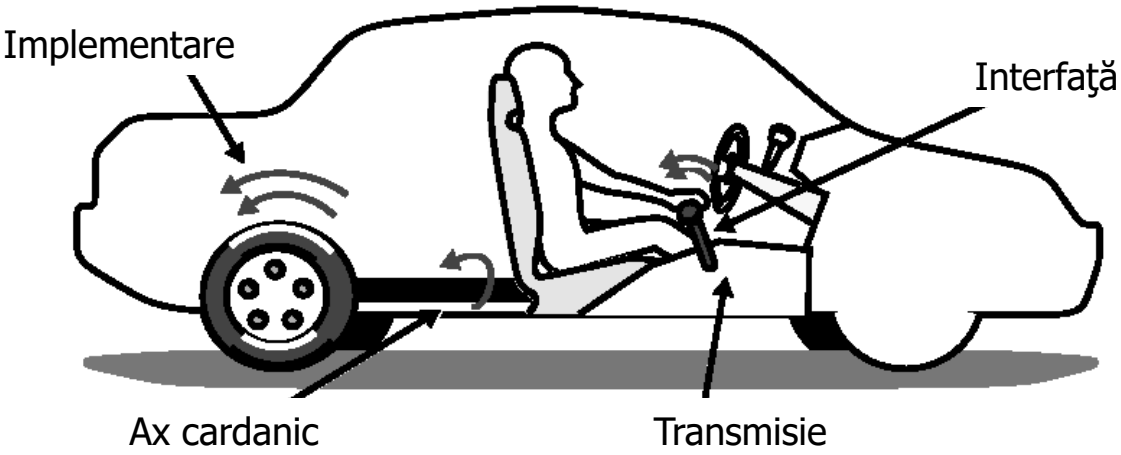
- Fiecare mesaj necesită:
  - *un emiţător (expeditor)*: obiectul care iniţiază acţiunea
  - *un receptor*: instanţa a cărei metode este invocată
  - *numele mesajului*: numele metodei apelate
  - opţional *parametri*: informaţii suplimentare necesitate de metodă pentru a opera
- Receptorul poate (dar nu este nevoit) să trimită un răspuns
  - prin *tipurile returnate*

Încapsulare

- Un automobil *încapsulează* multă informaţie
  - chiar literal, prin complexitatea construcţiei sale
- Dar nu este nevoie să ştii cum funcţionează o maşină pentru a o conduce
  - Volanul şi schimbarea vitezelor constituie interfaţa
  - Motorul, transmisia, axul cardanic, roţile, . . . , sunt implementarea (ascunsă)

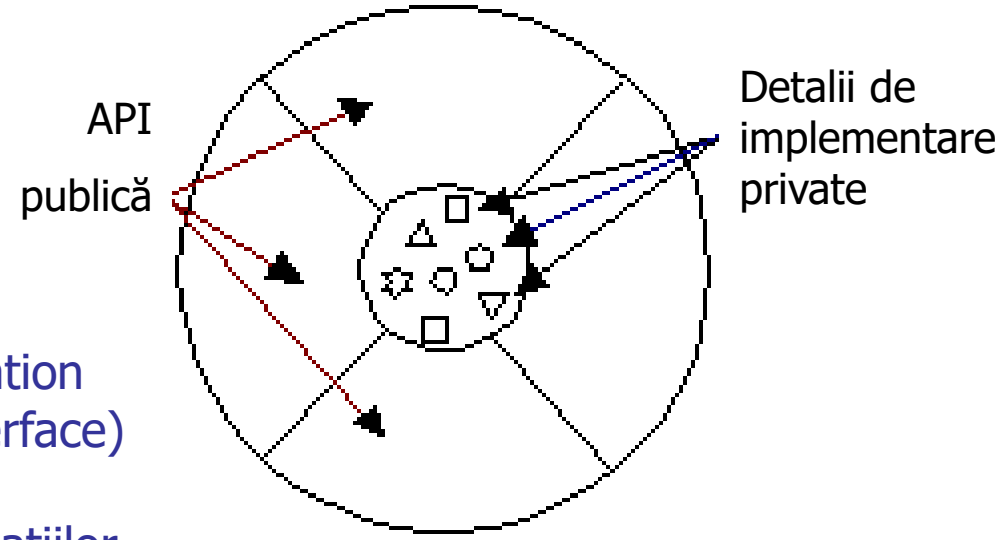
Încapsulare

- Asemănător, nu e nevoie să ştim cum funcţionează un obiect pentru a-i trimite mesaje
- Dar, este nevoie să ştim ce mesaje înţelege (adică, care îi sunt capabilităţile)
  - clasa instanţei determină ce mesaje îi pot fi trimise



Încapsulare

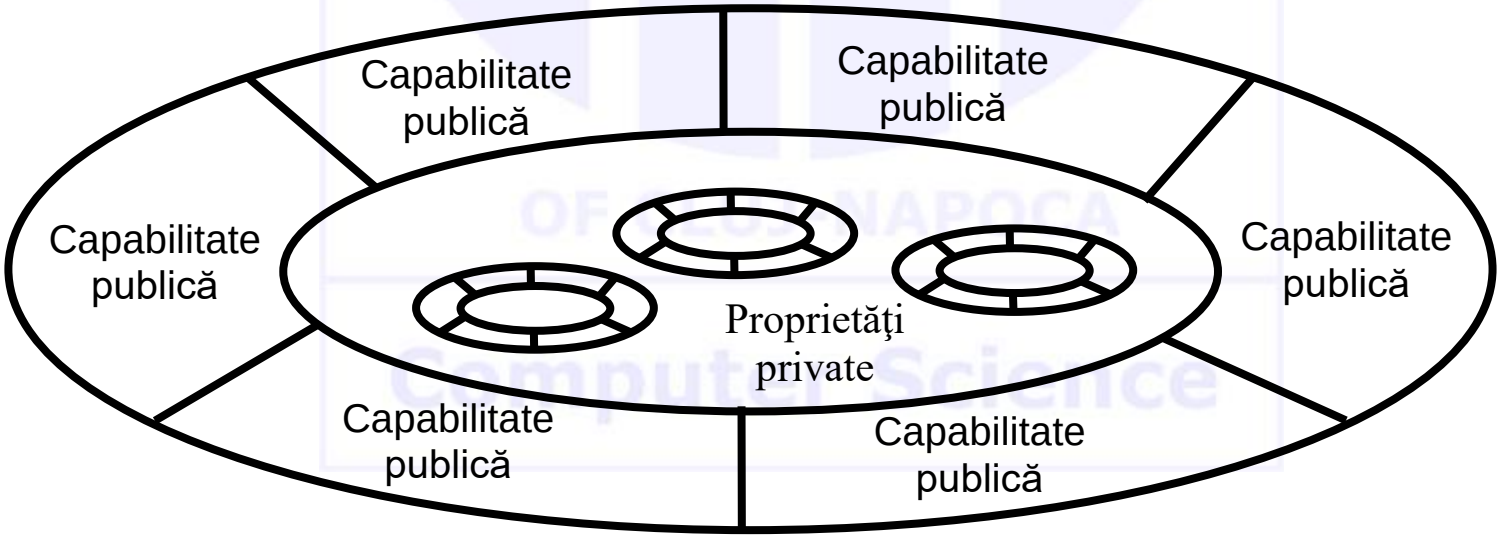
- Închiderea datelor într-un obiect
  - Datele nu pot fi accesate direct din afară
- Oferă securitatea datelor



Notă. API (Application Programming Interface) = interfață pentru programarea aplicațiilor

Vederile unei clase

- Obiectele separă *interfaţa* de *implementare*
  - obiectul este "cutie neagră"; ascunde funcţionarea şi părţile interne
  - interfaţa protejează implementarea împotriva utilizării greşite

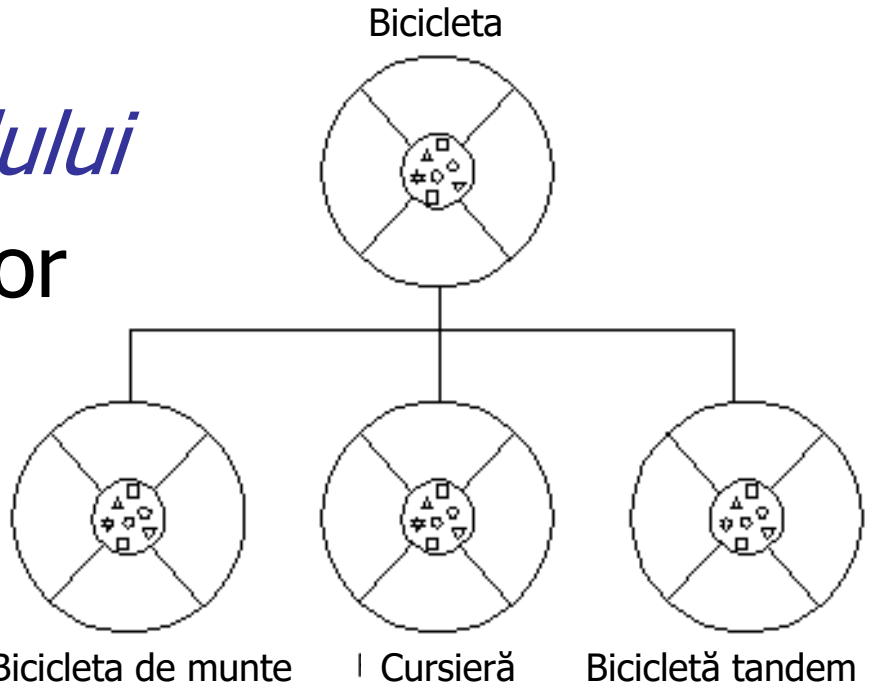


Vederile unei clase

- Interfaţa: vedere *publică*
  - permite instanţelor să coopereze unele cu altele fără a şti prea multe detalii
  - ca un *contract*: constă dintr-o listă de capabilităţi şi documentaţie pentru cum să fie folosite
- Implementarea: vedere *privată*
  - proprietăţile care ajută capabilităţile să-şi îndeplinească sarcinile

Moştenirea

- O clasă *(subclasă)* poate *moşteni* atribute şi metode dintr-o altă clasă (superclasă)
- Subclasele furnizează comportament specializat
- Oferă *reutilizarea codului*
- Evită *duplicarea* datelor



Polimorfism

- Abilitatea de a lua multe forme
- *Aceeaşi metodă* folosită într-o superclasă poate fi *suprascrisă* în subclase pentru a da o *funcţionalitate diferită*
- D.e. Superclasa 'Poligon' are o metodă numită, *aflaSuprafata*

  aflaSuprafata în subclasa 'Triunghi' → a=x*y/2

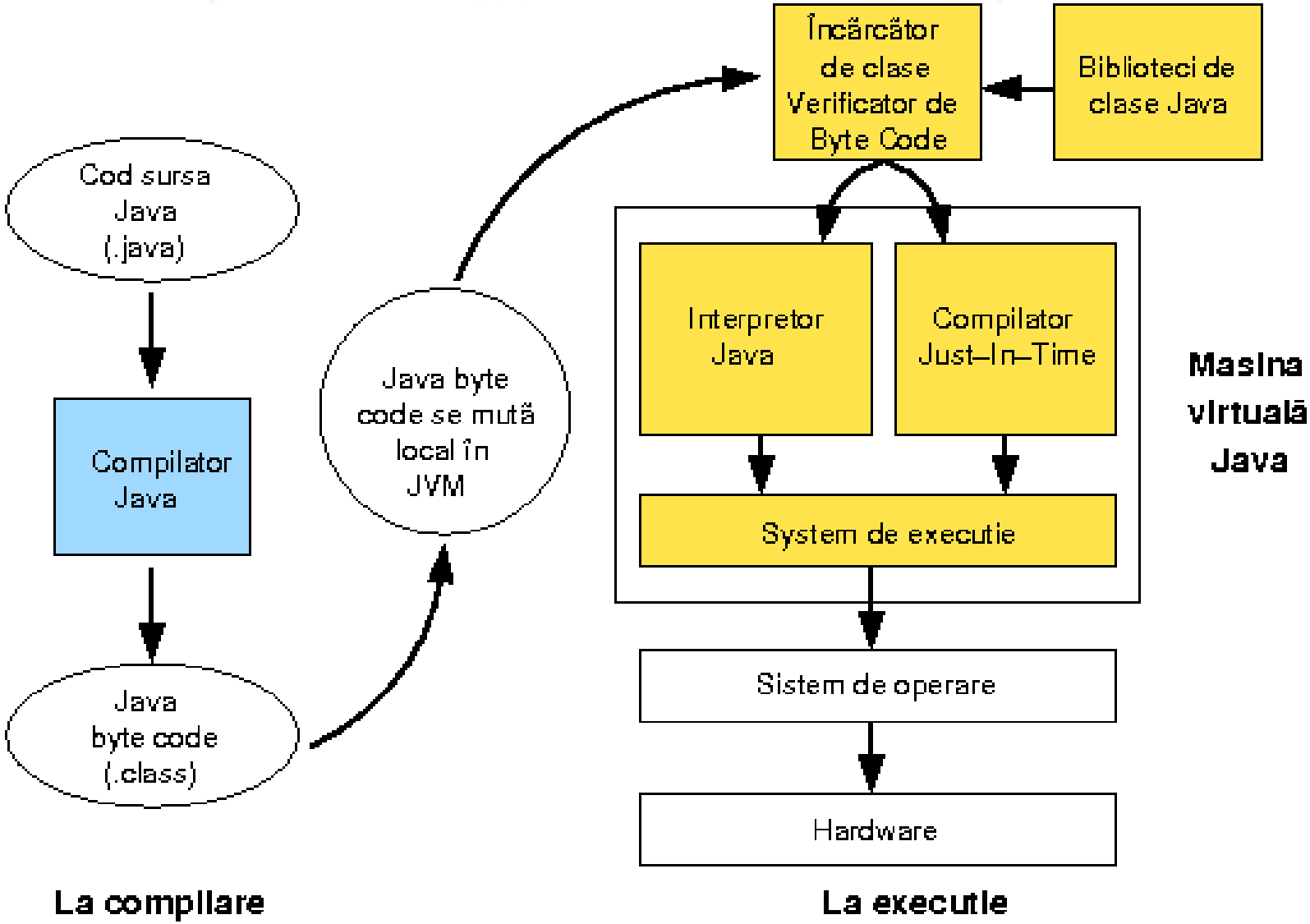
  aflaSuprafata în subclasa 'Dreptunghi' → a=x*y

Java. Caracteristici

- James Gosling şi Patrick Naughton (conducătorii echipei care a dezvoltat limbajul), au definit limbajul Java ca fiind

"Un limbaj simplu, orientat pe obiecte, care "înţelege" reţelele de calculatoare, interpretat, robust, sigur, neutru faţă de arhitecturi, portabil, de înaltă performanţă, multi-fir, dinamic"

Mediul Java

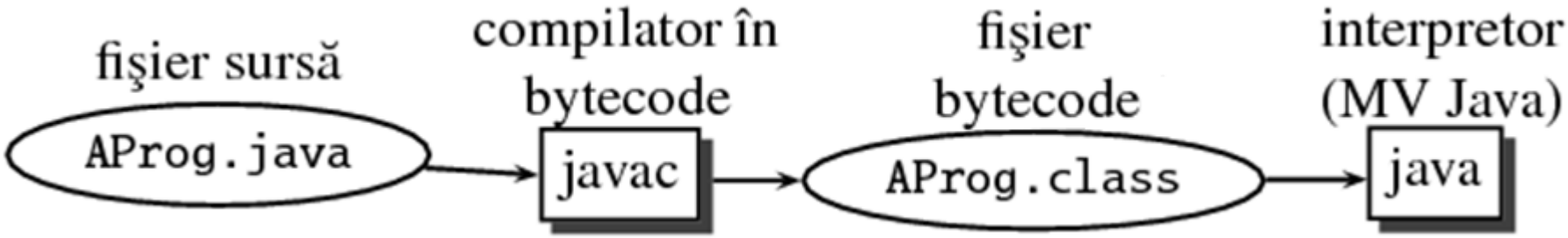

Încărcător de clase Verificator de Byte Code
Biblioteci de clase Java
Cod sursa Java (.java)
Compilator Java
Java byte code (.class)
Java byte code se mută local în JVM
Interpretor Java
Compilator Just–In–Time
System de executie
Sistem de operare
Hardware
Masina virtuală Java
La compilare
La executie

Executarea programelor Java

- Java foloseşte un proces în doi paşi
- Compilează programul în *bytecodes*
  - *bytecode* este apropiat de formatul instrucţiunilor în limbaj maşină, dar nu chiar la fel — este un "limbaj maşină" generic
  - nu corespunde nici unui procesor real
- *Maşina virtuală* (VM) interpretează bytecode în limbaj maşină nativ şi-l rulează
  - există maşini virtuale diferite pentru calculatoare diferite, deoarece bytecode nu corespunde unei maşini reale

Executarea programelor Java

- Se utilizează *acelaşi* bytecode Java pe *calculatoare diferite* fără a recompila codul sursă
  - fiecare VM interpretează acelaşi bytecode Java
  - aceasta permite să se execute programe Java prin simpla obţinere a bytecodes din pagini de Web
- Aceasta face codul Java să ruleze peste-platforme
  - marketing-ul spune, "Scrie o dată, rulează oriunde!"
  - adevărat pentru "Java pur", nu pentru variante



Aplicaţii Java

- Tipuri de aplicaţii Java
  - *aplicaţii de-sine-stătătoare*
  - *applets/servlets*

- O *aplicaţie de-sine-stătătoare* sau un program "obişnuit" este o clasă care are o metodă numită **`main`**
  - Când se lansează programul Java respectiv, *sistemul de execuţie* invocă automat metoda numită **`main`**
  - Toate *applicaţiile de-sine-stătătoare* încep din metoda **`main`**

Applet-uri vs. Aplicații de sine stătătoare

- Un *applet* Java este un program Java destinat a fi rulat dintr-un browser de Web
  - Applet-urile folosesc întotdeauna o interfaţă cu ferestre
  - Este o aplicaţie având caracteristici limitate, care necesită resurse de memorie limitate şi portabilă între sistemele de operare

- Aplicaţiile *de-sine-stătătoare* pot folosi atât interfaţa cu ferestre cât și I/E de consolă (adică în modul text)

Încărcătorul de clase

- Programele Java sunt divizate în unităţi mai mici numite *clase*
  - Fiecare definiţie de clasă este în mod normal într-un fişier separat şi compilată separat
- *Încărcătorul de clase* este un program care leagă *bytecode*-ul claselor necesare pentru a rula un program Java
  - în alte limbaje de programare, corespondentul său este *editorul de legături (link-editor)*

Cuvinte cheie Java (cuvinte rezervate)

- Cuvinte care nu pot fi folosite la altceva decât în modul predefinit din limbaj
  - *abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, extends, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while*
  - *null, true, false* – predefinite ca literali

Compilarea unui program sau a unei clase Java

- Fiecare definiţie de clasă trebuie să se afle într-un fişier al cărui nume este numele clasei urmat de extensia **`.java`**
  - Exemplu: Clasa **`UnProgram`** trebuie să se afle în fişierul numit **`UnProgram.java`**
- Fiecare clasă este compilată folosind comanda **`javac`** urmată de numele fişierului care conţine clasa

```
javac UnProgram.java
```

  - Rezultatul este un program în byte-code cu acelaşi nume ca al clasei, urmat de extensia **`.class`**

```
UnProgram.class
```

Rularea unui program Java

- Un program Java poate fi *rulat* (**`java`**) după ce i-au fost compilate toate clasele
  - Rulaţi doar clasa care conţine o metodă **`main`** (sistemul va încărca şi rula celelalte clase automat, dacă mai sunt)
  - Metoda **`main`** are semnătura

```
public static void main(String[ ] args)
```

  - Comanda de lansare a programului trebuie urmată doar de numele clasei (fără extensii)

```
java UnProgram
```

Convenţii pentru nume

- Începeţi numele de variabile, metode şi obiecte cu o literă mică, indicaţi limitele "cuvintelor" cu o litera mare şi pentru celelalte caractere folosiţi doar litere şi cifre ("camelcase")

```
vitezaMaxima    rataDobanzii    oraSosirii
```

- Începeţi numele de clase cu majusculă şi pentru restul identificatorului aplicaţi regula de mai sus

```
UnProgram    OClasa    String
```

Declararea variabilelor

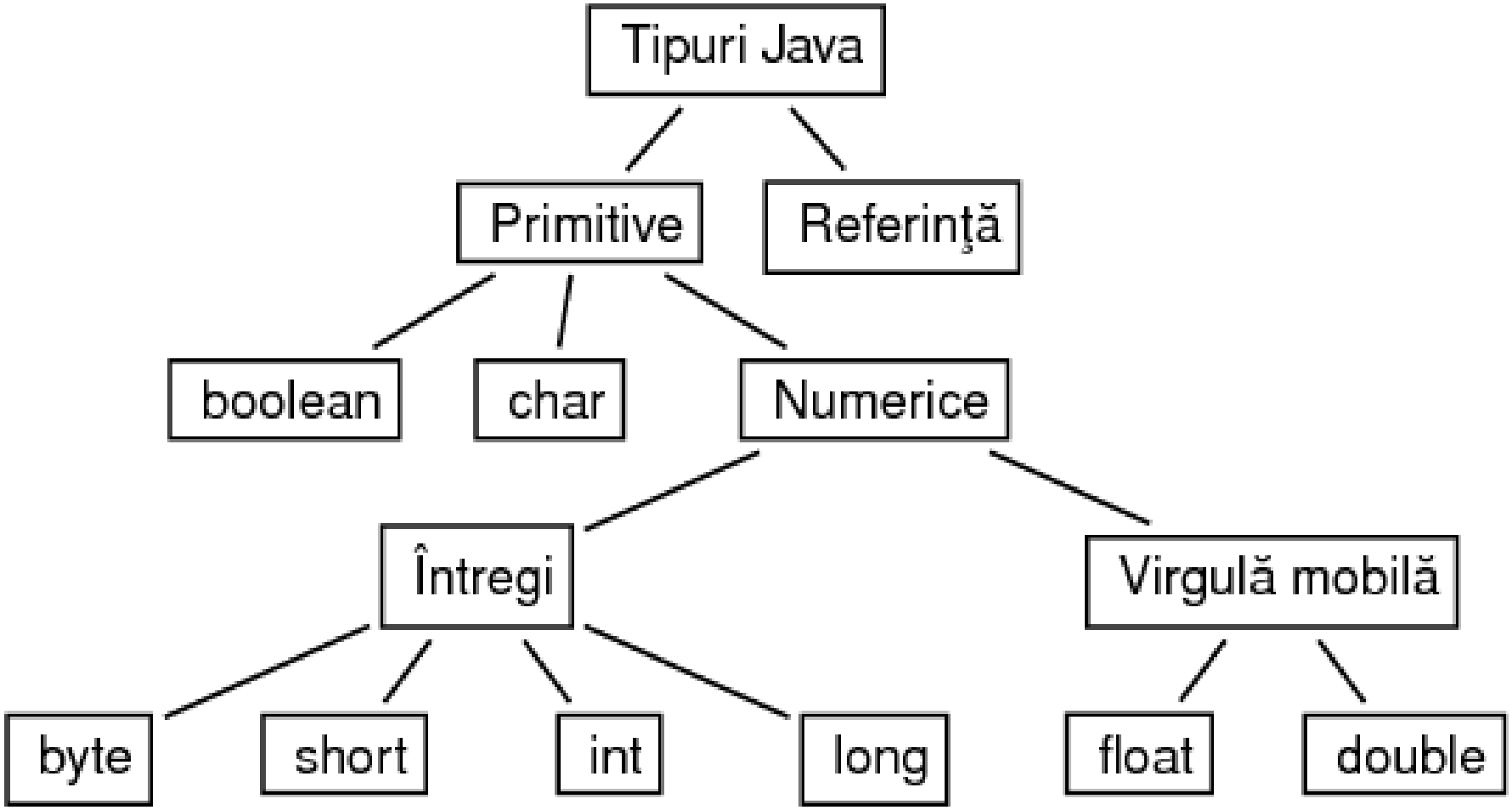
- Fiecare variabilă dintr-un program Java trebuie *declarată* înainte de utilizare
  - Declaraţia informează compilatorul asupra tipului de dată care va fi stocat în variabilă
  - Tipul variabilei este urmat de unul sau mai multe nume separate de virgule şi terminat cu punct şi virgulă
  - Variabilele se declară de obicei chiar înainte de folosire sau la începutul unui bloc (indicat de o acoladă deschisă {)
  - Tipurile simple în Java sunt numite *tipuri primitive*

```
int numarulDeCai;
double oLungime, lungimeaTotala;
```

Modificatori de acces pentru variabile/metode

- **private**
  - variabila/metoda este vizibilă local în cadrul clasei
  - *"Vizibilă doar mie"*
- **protected**
  - variabila/metoda poate fi văzută din toate clasele, subclasele şi celelalte clase din acelaşi pachet (package)
  - *"Vizibilă în familie"*
- **public**
  - variabila/metoda poate fi văzută din toate clasele
  - *"Vizibilă tuturor"*
- Modificatorul de acces implicit, nu are cuvânt cheie
  - **public** pentru ceilalţi membrii din acelaşi pachet
  - **private** pentru oricine din afara pachetului
  - numită şi *acces în pachet*
  - *"Vizibilă în vecinătate"*

Tipuri Java


Tipuri Java
Primitive
Referinţă
boolean
char
Numerice
Întregi
Virgulă mobilă
byte
short
int
long
float
double

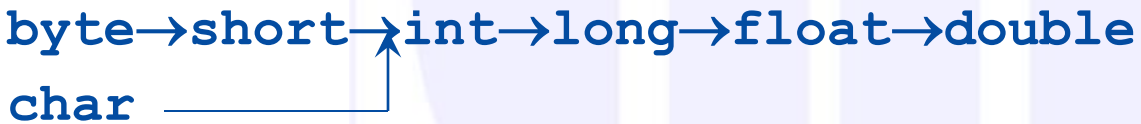
Tipuri primitive

Tip primitiv	Biţi	Minimum	Maximum	Wrapper type
boolean	—	—	—	**Boolean**
char	16-bit	Unicode 0	Unicode 2^{16}- 1	**Character**
byte	8-bit	-128	+127	**Byte**
short	16-bit	-2^{15}	$+2^{15}-1$	**Short**
int	32-bit	-2^{31}	$+2^{31}-1$	**Integer**
long	64-bit	-2^{63}	$+2^{63}-1$	**Long**
float	32-bit	IEEE754	IEEE754	**Float**
double	64-bit	IEEE754	IEEE754	**Double**
void	—	—	—	**Void**

Toate tipurile numerice sunt cu semn

Compatibilitate la asignare

- Mai general, o valoare de orice tip din lista următoare poate fi asignată unei variabile de orice alt tip care apare la dreapta ei



  - Gama valorilor de la dreapta este mai largă
- Este necesară o conversie de tip explicită (*type cast*) pentru a asigna o valoare de un tip la o variabilă care apare la stânga ei în lista de mai sus (d.e., `double` la `int`)
- Observaţi că în Java un `int` nu poate fi asignat la o variabilă de tip `boolean`, nici un `boolean` la o variabilă de tip `int`

Operatori aritmetici şi expresii

- Ca în majoritatea limbajelor, şi în Java se pot forma *expresii* folosind variabile, constante şi operatori aritmetici
  - Operatori aritmetici; **+** (adunare), **–** (scădere), ***** (înmulţire), **/** (împărţire), **%** (modulo, rest)
  - Se poate folosi o expresie oriunde este legal să se folosească o valoare de tipul produs de expresie

Operatori aritmetici şi expresii

- Dacă se combină un operator aritmetic cu operanzi de tipul **`int`**, atunci tipul rezultat este **`int`**
- Dacă se combină un operator aritmetic cu unul sau doi operanzi de tipul **`double`**, atunci tipul rezultat este **`double`**
- La combinarea de operanzi de tip diferit, tipul rezultat este cel mai din dreapta din lista de mai jos care se află în expresie

`byte`→**`short`**→**`int`**→**`long`**→**`float`**→**`double`**

`char` ↗

  - Excepţie: Dacă tipul rezultat este **`byte`** sau **`short`** (potrivit regulii date), atunci tipul produs va fi de fapt un **`int`**

Reguli de precedenţă şi asociativitate

- La determinarea ordinii operaţiilor adiacente, operaţia cu precedenţă mai mare (şi argumentele sale aparente) este grupată înaintea operaţiei de precedenţă mai mică

  `base + rate * hours` se evaluează ca

  `base + (rate * hours)`

- La precedenţă egală, ordinea operaţiilor este determinată de regulile de *asociativitate*

Posibilă problemă: Erorile de rotunjire la numerele în virgulă mobilă

- Numerele în virgulă mobilă sunt, în general, doar valori aproximative
  - Matematic, numărul în virgulă mobilă 1.0/3.0 este egal cu 0.3333333 . . .
  - Un calculator are o cantitate limitată de memorie
    - Poate stoca 1.0/3.0 ca ceva în genul lui 0.3333333333, puţin mai puţin decât o treime
  - De fapt numerele sunt stocate binar, dar consecinţele sunt aceleaşi: numerele în virgulă mobilă pot pierde precizie

Împărţirea întreagă şi cea în virgulă mobilă

- Dacă unul sau amândoi operanzii sunt în virgulă mobilă, împărţirea dă un rezultat în virgulă mobilă

  `15.0/2` se evaluează la `7.5`

- Cum ambii operanzi întregi, împărţirea dă un întreg
  - O eventuală parte fracţionară este ignorată
  - Nu se fac rotunjiri

    `15/2` se evaluează la `7`

- Aveţi grijă ca cel puţin un operand să fie în virgulă mobilă dacă este nevoie de partea fracţionară

Conversia de tip explicită

- O *conversie de tip explicită (type cast)* ia o valoare de un tip şi produce o valoare "echivalentă" de celălalt tip
  - Dacă `n` şi `m` sunt întregii de împărţit şi e nevoie de partea fracţionară, atunci cel puţin un operand trebuie să fie în virgulă mobilă **înainte** de efectuarea operaţiei

```
double ans = n / (double)m;
```

  - La fel ca în C, tipul dorit este pus între paranteze imediat înaintea variabilei de convertit
  - Tipul şi valoarea variabilei de convertit nu se schimbă

Conversia de tip explicită

- La conversia explicită de la virgulă mobilă la întreg, numărul este trunchiat, nu rotunjit
  - `(int)2.9` se evaluează la `2`, nu `3`
- La asignarea valorii unui întreg la o variabilă în virgulă mobilă, Java realizează o conversie explicită de tip automată numită *coerciţie de tip*

```
double d = 5;
```

- Nu este legal să se atribuie un `double` la un `int` fără o conversie explicită

```
int i = 5.5; // Ilegal
int i = (int)5.5 // Corect
```

Operatorii increment şi decrement

- Când oricare dintre operatorii ++ sau −− precede o variabilă şi este o parte a expresiei, expresia este evaluată folosind valoarea modificată a variabilei
  - Dacă `n` este `2`, atunci `2*(++n)` se evaluează la `6`
- Când oricare dintre operatori urmează unei variabile şi este parte a expresiei, expresia este evaluată folosind valoarea originală şi abia apoi se schimbă valoarea variabilei
  - Dacă `n` este `2`, atunci `2*(n++)` se evaluează la `4`