Programare orientată pe obiecte

1. Clase învelitoare (*wrapper classes*)
2. Câteva observaţii despre operatori
3. Structuri de control în Java
4. Clase şi Obiecte

Clase învelitoare

- În POO, clasele învelitoare sunt clasele care încapsulează tipurile primitive, astfel încât să se poată manipula ca niște obiecte și pentru a le putea 'atașa' metode (în special cu rol de conversie)

Tipul primitiv	Clasa Învelitoare	Argumentele constructorilor
byte	Byte	byte sau String
short	Short	short sau String
int	Integer BigInteger	int sau String String
long	Long	long sau String
float	Float	float, double sau String
double	Double BigDecimal	double sau String String
char	Character	char
boolean	Boolean	boolean sau String

Clase învelitoare

- *Boxing* (împachetare): procesul de trecere de la o valoare primitivă la un obiect al clasei sale învelitoare
  - Pentru a converti valoarea la una "echivalentă" de tip clasă, creaţi un obiect de tipul corespunzător folosind valoarea primitivă ca argument
  - Noul obiect va conţine o variabilă instanţă care stochează o copie a valorii primitive
  - Spre deosebire de alte clase, clasele învelitoare nu au constructori fără argumente

```
Integer integerObject = new Integer(42);
```

Clase învelitoare

- *Unboxing* (despachetare): procesul de trecere de la un obiect al unei clase învelitoare la valoarea corespunzătoare a unui tip primitiv
  - Metodele de convertire a unui obiect din clasele **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, şi **`Character`** la valorile primitive corespunzătore sunt (în ordine) **`byteValue`**, **`shortValue`**, **`intValue`**, **`longValue`**, **`floatValue`**, **`doubleValue`**, şi **`charValue`**
  - Nici una dintre aceste metode nu necesită argumente

```
int i = integerObject.intValue();
```

Boxing şi Unboxing automat

- Începând cu versiunea 5.0, Java poate face automat împachetare/despachetare
- În loc să creăm un obiect din clasa învelitoare (ca mai înainte), se poate face o conversie de tip automată

```
Integer integerObject = 42;
```

  - Java nu garantează că două referințe de astfel de obiecte care împachetează automat o aceeași valoare primitivă sunt diferite!
- În loc să trebuiască să invocăm metoda corespunzătoare (ca `intValue`, `doubleValue`, `charValue` etc.) pentru a converti un obiect din clasa învelitoare la tipul său asociat, valoarea primitivă poate fi recuperată automat

```
int i = integerObject;
```

Boxing == și *equals()*

- Pentru a verifica dacă două variabile de tip referință au aceeași valoare, vom folosi metoda *equals()*
- Exemplu:

```
Integer i1 = 42; //împachetare automată
Integer i2 = 42; //împachetare automată
if (i1 != i2) // nu este garantat că sunt diferite!
   System.out.println("Obiecte diferite");
if (i1.equals(i2)) // garantat că sunt identice!
   System.out.println("Variabile cu valori egale");
```

Afișare la ieșirea standard:

```
Obiecte diferite (nu este garantat că se afișează!)
Variabile cu valori egale
```

Constante şi metode statice în clasele învelitoare

- Constante pentru valori maxime şi minime pentru tipurile primitive
  - Exemple: `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Double.MAX_VALUE`, `Double.MIN_VALUE` etc.

- Clasa `Boolean` are nume pentru două constante de tipul `Boolean`
  - `Boolean.TRUE` şi `Boolean.FALSE` sunt obiectele booleene corespunzătoare valorilor `true` şi `false` ale tipului primitiv `boolean`

Constante şi metode statice în clasele învelitoare

- Metode statice pentru conversia unei reprezentări corect formate ca şir de caractere a unui număr la numărul de un tip dat
  - Metodele `Integer.parseInt`, `Long.parseLong`, `Float.parseFloat` şi `Double.parseDouble` sunt pentru (în ordine) `int`, `long`, `float` şi `double`
  - Exemplu

```
double x = Double.parseDouble("123.9");
```

- Metode statice pentru conversia duală celei anterioare
  - Exemplu

```
String x = Double.toString(123.99);
```

- Clasa `Character` conţine un număr de metode utile la prelucrarea şirurilor de caractere

Referinţe Java

- Toate obiectele sunt create în memorie (pe *heap* – zona de memorie unde se face alocarea dinamică)
- Pentru a accesa datele dintr-un obiect sau a lucra cu un obiect, este nevoie de o variabilă pe stivă, variabilă care poate stoca o referinţă la adresa obiectului
- Despre variabilele care stochează referinţe la adrese de obiecte se spune că păstrează tipul de date *referinţă*
- Exemplu

```
PlatitorTaxe t = new PlatitorTaxe(1111111120633,
                                  30000);
```

Clasa `String`

- Nu există un tip primitiv pentru şiruri în Java
- Clasa **`String`** este o clasă Java predefinită folosită la stocarea şi prelucrarea şirurilor de caractere
- Obiectele de tipul **`String`** sunt compuse din şiruri de caractere scrise între ghilimele
  - Orice şir scris între ghilimele este de clasă **`String`**

```
"Curs de POO"
```

- Unei variabile de tipul **`String`** i se poate da valoarea unui obiect **`String`**

```
String s = "Curs de POO";
```

Concatenarea şirurilor de caractere

- *Concatenare*: folosind operatorul + operator aplicat asupra a două şiruri pentru a le conecta şi a forma un şir mai lung
  - Dacă

    `String salut` = `"Bună ziua, "`, şi

    `String javaClass` = `"viitori colegi"`,

    atunci `salut + javaClass`

    este stringul `"Bună ziua, viitori colegi"`

- Orice număr de şiruri pot fi concatenate

- La combinarea unui şir de caractere cu aproape orice alt tip, rezultatul este un şir de caractere
  - `"Răspunsul este " + 42` se evaluează la `"Răspunsul este 42"`

Metode ale clasei String

- Clasa **`String`** conţine multe metode utile la aplicaţiile care prelucrează şiruri de caractere
  - O metodă a clasei **`String`** se apelează prin numele unui obiect **`String`**, un punct, numele metodei şi o pereche de paranteze între care sunt cuprinse argumentele (dacă sunt)
  - O valoare returnată de o metodă **`String`** se poate folosi oriunde se poate folosi o valoare de tipul returnat

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " +
                   greeting.length());
```

  - *Poziţia* sau *indexul* unui caracter într-un şir se numără întotdeauna de la zero

Metode ale clasei `String`

Lungimea şirului

`i =`	*s*`.length()`	lungimea şirului *s*.

Comparaţii (notă: folosiţi aceste metode în loc de == şi !=)

`i =`	*s*`.compareTo(`*t*`)`	compară cu t. Returnează <0 dacă s<t, 0 dacă ==, >0 dacă s>t
`i =`	*s*`.compareToIgnoreCase(`*t*`)`	la fel ca mai sus, dar fără a ţine seama de majuscule/minuscule
`b =`	*s*`.equals(`*t*`)`	true dacă cele două şiruri au valori egale
`b =`	*s*`.equalsIgnoreCase(`*t*`)`	la fel ca mai sus, dar fără a ţine seama de majuscule/minuscule
`b =`	*s*`.startsWith(`*t*`)`	true dacă *s* începe cu *t*
`b =`	*s*`.startsWith(`*t*`, `*i*`)`	true dacă *t* apare începând cu indexul i
`b =`	*s*`.endsWith(`*t*`)`	true dacă *s* se termină cu *t*

Metode ale clasei `String`

Căutare (notă : Toate metodele "indexOf" returnează -1 dacă şirul/caracterul nu este găsit)

`i =`	`s.indexOf(t)`	indexul primei apariţii lui String *t* in *s*.
`i =`	`s.indexOf(t, i)`	indexul lui String *t* la sau după poziţia *i* în *s*.
`i =`	`s.indexOf(c)`	indexul primei apariţii caracterului *c* în *s*.
`i =`	`s.indexOf(c, i)`	indexul caracterului *c* la sau după poziţia *i* din *s*.
`i =`	`s.lastIndexOf(c)`	indexul ultimei apariţii lui *c* în *s*.
`i =`	`s.lastIndexOf(c, i)`	indexul ultimei apariţii lui *c* pe sau înainte de poziţia *i* în *s*.
`i =`	`s.lastIndexOf(t)`	indexul ultimei apariţii lui *t* în *s*.
`i =`	`s.lastIndexOf(t, i)`	indexul ultimei apariţii a lui *t* pe sau înainte de poziţia *i* în *s*.

Metode ale clasei `String`

Obţinerea de părţi

`c =`	*s*`.charAt(`*i*`)`	caracterul de la poziţia *i* din *s*.
`s1 =`	*s*`.substring(`*i*`)`	subşirul de la indexul *i* până la sfârşitul lui *s*.
`s1 =`	*s*`.substring(`*i*`, `*j*`)`	subşirul de la indexul *i* până înainte de indexul *j* din *s*.

Crearea unui nou şir din original

`s1 =`	*s*`.toLowerCase()`	nou String cu toate caracterele minuscule
`s1 =`	*s*`.toUpperCase()`	nou String cu toate caracterele majuscule
`s1 =`	*s*`.trim()`	nou String fără spaţii albe la început şi sfârşit
`s1 =`	*s*`.replace(`*c1*`, `*c2*`)`	nou String cu toate *c2*-urile înlocuite prin *c1*

Metode ale clasei `String`

Metode statice pentru conversia la String

`s =`	`String.`**`valueOf(`***`x`***`)`**	Converteşte x la String, unde x este *orice* valoare de tip (primitiv sau obiect).
`s =`	`String.`**`format(`***`f, x...`***`)`**	[Java >=5] Foloseşte formatul *f* pentru a converti un număr variabil de parametri, *x* la un şir.

- Lista nu este exhaustivă

Seturi de caractere

- *ASCII*: Set de caractere folosit de multe limbaje de programare, care conţine toate caracterele folosite în mod normal pe o tastatură pentru limba engleză plus câteva caractere speciale
  - Fiecare caracter este reprezentat de un cod numeric
- *Unicode*: Set de caractere folosit de limbajul Java care include tot setul ASCII plus multe dintre caracterele folosite cu alfabete nelatine
  - Exemple

```
char c='\u0103'; // litera 'ă'
String s="\u00eencoto\u015fm\u0103ni\u0163i"; // încotoşmăniţi
```

O porţiune din codul Unicode

	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074

Numirea constantelor

- În loc de numere "anonime", declaraţi constante simbolice (cu nume) şi folosiţi-le numele

```
public static final double CM_PER_INCH = 2.54;
public static final int HOURS_PER_DAY = 24;
```

  - Previne schimbarea nedorită a valorii
  - Uşurează modificarea valorii
- Convenţia de nume pentru constante
  - Toate literele majuscule
  - Limitele de cuvinte marcate prin liniuţa de subliniere

Comentariile

- Comentariu de o *linie*
  - Începe cu simbolurile `//` şi provoacă ignorarea a ceea ce urmează până la sfârşitul liniei
  - Folosit de cel care scrie codul sau de cel care îl modifică
- Comentariul *bloc*
  - La fel ca în C (perechea `/*`, `*/`)
  - Furnizează documentaţie utilizatorilor programului

Documentarea programului

- Java include programul **`javadoc`** care extrage automat documentaţia din comentariile bloc din clasele definite, dacă
  - Începutul comentariului are un asterisc suplimentar (`/**`)

- Un program bine scris este autodocumentat
  - Structura sa se clarifică prin alegerea numelor de identificatori şi modelul de indentare

Operatori

- Sunt trataţi în detaliu în lucrarea de laborator
- Câteva diferenţe faţă de C:
  - Concatenarea pentru String: operatorul +
  - Operatorul pe biţi >>>
    - D.e. n >>> p; // deplasează biţii lui *n* spre dreapta cu *p* poziţii. În poziţiile de rang superior se inserează zerouri.
  - Operatori pentru lucrul cu obiecte – vor fi tratați în detaliu mai târziu

Despre precedenţa operatorilor

Precedenţa operatorilor	

`. [] (args) post ++ --`
`! ~ unary + - pre ++ --`
`(type) new`
`* / %`
`+ -`
`<< >> >>>`
`< <= > >= instanceof`
`== !=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`= += -= etc`
	Tineţi minte precedenţa pentru operatorii unari,
	* / %
	+ -
	comparaţii
	&& ||
	= *asignări*
	Folosiţi paranteze () pentru ceilalţi

Instrucţiunea **if**

- Instrucţiunea **if** specifică ce bloc de cod să se execute în funcţie de rezultatul evaluării unei condiţii numită *expresie booleană*

```
if (<expresie booleană>)
  <then> Instr1
else
  <else> Instr 2
```

Sau:

```
(<expresie booleană>)? Instr1 : Instr2
```

ex: `int max = (a>b)? a:b;`

- **`<expresie booleană>`** este o expresie condiţională care se evaluează la true sau false.
  - Sintaxă similară limbajului C, dar atenţie la ce este o expresie booleană în Java

Compararea obiectelor

- La compararea a două ***variabile***, se compară *conţinutul* lor
- În cazul ***obiectelor***, *conţinutul* este ***adresa*** unde este stocat obiectul (adică se vor compara referinţele)
  - Şirurile de caractere în Java sunt obiecte ale clasei String
  - Clasa String furnizează metode de comparare
- Cea mai bună metodă în ceea ce priveşte compararea obiectelor este să *definim metode de comparare* pentru clasa respectivă

Sugestii pentru **if**

- Începeţi testul cu cazul cel mai relevant
  - Face codul mai uşor de citit
- Nu uitaţi de clauza else!
- Evitaţi condiţiile complicate
- Divizaţi condiţiile în variabile/funcţii booleene
- Încercaţi să folosiţi condiţii pozitive
- Exemplu – se preferă a doua variantă

```
if (!node.isFirst() && node.value() != null)
  instr1
else
  instr2

if (node.isFirst() || node.value() == null)
  instr2
else
  instr1
```

Instrucţiunea **switch**

- Sintaxa pentru instrucţiunea **switch**

```
switch ( < expresie aritmetică>){
  <case eticheta_1>: <case corp 1>
  ...
  <case eticheta_n>: <case corp n>
}
```

- Tipul de dată al `<expresie aritmetică>` trebuie să fie **char, byte, short, int**

Instrucţiunea **switch**

- Exemplu

```
int month = 8;
String monthString;
switch (month) {
    case 1:  monthString = "January";
             break;
    case 2:  monthString = "February";
             break;
    //. . .

    case 12: monthString = "December";
             break;
    default: monthString = "Invalid month";
             break;
}
System.out.println(monthString);
```

Instrucţiunea **switch**

- Dacă există o valoare care se potriveşte cu valoarea expresiei, atunci se execută corpul acesteia. Altfel execuţia continuă cu instrucţiunea care urmează instrucţiunii **switch**
- Instrucţiunea **break** face să nu se execute porţiunea care urmează din **switch**, ci să se continue cu ceea ce vine după **switch**
- **break** este necesar pentru a executa instrucţiunile dintr-un caz, şi numai unul
  - iarăşi, ca în C

Sugestii pentru **switch**

- Ordonaţi cazurile (logic sau alfabetic)
  - Prevedeţi întotdeauna cazul implicit (`default`)
  - Întotdeauna folosiţi `break` între cazuri
  - Încercaţi să păstraţi mică dimensiunea instrucţiunii `switch`
  - Divizaţi cazurile de mari dimensiuni în funcţii

Instrucţiuni repetitive

- *Instrucţiunile repetitive* controlează un bloc de cod de executat un număr fixat de ori sau până la îndeplinirea unei anumite condiţii
- Ca şi C, Java are trei instrucţiuni repetitive:
  - **while**
  - **do-while**
  - **for**
- Instrucţiunile repetitive sunt numite şi *instrucţiuni de ciclare*, blocul de instrucţiuni care se repetă (`<instructiuni>` în cele ce urmează) este cunoscut sub numele de *corpul ciclului*

Instrucţiunea **while**

- În Java, instrucţiunile **while** au formatul general:

```
while ( <expresie booleana> )
  <instructiuni>
```

- Cât timp **<expresie booleana>** este true, se execută corpul ciclului
- Într-o *buclă controlată prin contor*, corpul ciclului este executat de un număr fixat de ori
- Într-o buclă *controlată printr-o santinelă*, corpul ciclului este executat în mod repetat până când se întâlneşte o valoare stabilită, numită *santinelă*

Capcane în scrierea instrucţiunilor repetitive

- La instrucţiunile repetitive este important să se asigure terminarea ciclului la un moment dat
- Tipurile de probleme potenţiale de ţinut minte:
- **Bucla infinită**

```
int item = 0;
while (item < 5000) {
  item = item * 5;
}
```

  - Deoarece **item** este iniţializat la 0, **item** nu va fi niciodată mai mare de 5000 (0 = 0 * 5), astfel că bucla nu se va termina niciodată

Capcane în scrierea instrucţiunilor repetitive

- **Bucla infinită**

```
int count = 1;
while (count != 10)
{
  count = count + 2;
}
```

- În acest exemplu, (al cărui instrucţiune **while** este în buclă infinită), contorul va fi 9 şi 11, dar nu 10.
- **Eroarea prin depăşire de capacitate** apare atunci când se încearcă asignarea unei valori mai mari decât valoarea maximă pe care o poate stoca variabila

Capcane în scrierea instrucţiunilor repetitive

- **Depăşirea de capacitate**
- În Java, o depăşire a capacităţii de reprezentare nu provoacă terminarea programului
  - La tipurile **float** şi **double** se atribuie variabilei o valoare care reprezintă infinit
  - La tipul **int**, o valoare pozitivă devine negativă, iar una negativă devine pozitivă, ca şi cum valorile ar fi plasate pe un cerc cu maximul şi minimul învecinate
- Numerele reale nu se folosesc în teste exacte sau incrementări, deoarece valorile lor sunt reprezentate cu aproximaţie
- **Eroarea prin depăşire cu unu** este o altă capcană frecventă

Instrucţiunea **do-while**

- Instrucţiunea **while** este o *buclă cu pre-testare* (testul se face la intrarea în buclă)
  - Corpul buclei poate să nu fie executat niciodată
- Instrucţiunea **do-while** este o *buclă cu post-testare*
  - Corpul ciclului este executat cel puţin o dată
- Formatul instrcuţiunii **do-while** este:

```
do
  <instructiuni>
while (<expresie booleană>);
```

- **<instructiuni>** se execută până când **<expresie booleană>** devine falsă

Controlul repetitiv o buclă-şi-jumătate (buclă infinită întreruptă)

- Atenţie la două lucruri la folosirea buclelor întrerupte
  - **Pericolul ciclării infinite.** Expresia booleană a instrucţiunii **`while`** este **`true`** întotdeauna. Dacă nu există o instrucţiune **`if`** care să forţeze ieşirea din ciclu, rezultatul va fi o buclă infinită
  - **Puncte de ieşire multiple.** Se poate totuşi, chiar dacă e mai complicat să se scrie un control buclă cu mai multe ieşiri (**`break`**). Practica recomandă pe cât posibil curgerea *o-intrare o-ieşire* a controlului

Instrucţiunea **for**

- Formatul instrucţiunii **for** este:

```
for (<initializare>; <expresie booleană>; <increment>)
      <instrucţiuni>
```

- Exemplu:

```
int sum = 0;
for (int i = 1; i <=100; i++){
  sum += i;
}
```

- Variabila `i` din exemplu se numeşte *variabilă de control* (contorizează numărul de repetiţii)
- `<increment>` poate fi orice cantitate
- Din nou, ca în C

Sugestii pentru ciclurile **for**

- Sunt ideale atunci când numărul de iteraţii este cunoscut
  - Folosiţi o instrucţiune pentru fiecare parte
  - Declaraţi variabilele de ciclu în antet (reduce vizibilitatea şi interferenţa)
  - Nu se recomandă modificarea variabilei de control în corpul ciclului

break cu etichetă

- **break** se foloseşte în **bucle** şi **switch**
  - semnificaţiile sunt diferite
- **break** poate fi şi urmat de o etichetă, *L*
  - încearcă să transfere controlul la o instrucţiune etichetată cu *L*
  - Dacă nu există instrucţiuni etichetate cu *L*, apare o eroare de compilare
  - Un `break` cu etichetă permite ieşirea din mai multe bucle imbricate
  - Eticheta trebuie să *preceadă* bucla cea mai exterioară din care se doreşte ieşirea
  - Această formă nu există în C

Exemplu pentru **break** cu etichetă

```
int n;
read_data:
while(…) {
   …
   for (…) {
       n= Console.readInt(…);
       if (n < 0) // nu se poate continua
             break read_data; // break out of data loop
       …
   }
}
// verifica dacă este succes sau esec
if (n < 0) {
   // trateaza situatiile cu probleme
}
else {
   // am ajuns aici normal
}
```

continue cu etichetă

- Forma etichetată a instrucţiunii **`continue`** sare peste iteraţia curentă a unei bucle exterioare marcate cu o etichetă dată

- Eticheta trebuie să *preceadă* bucla cea mai exterioară din care se doreşte ieşirea

```
public class ContinueWithLabelDemo {
    public static void main(String[] args) {
        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;
        int max = searchMe.length() - substring.length();
     test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++)!= substring.charAt(k++)){
                    continue test;
                }
            }
            foundIt = true;
            break test;
        }
        System.out.println(foundIt ? "Found it" : "Didn't
                                                  find it");
    }
}
```

for pentru iteraţii peste colecţii şi tablouri

- Creat special pentru iteraţii peste colecţii şi tablouri (vom reveni asupra instrucţiunii) – Java 5
- Nu funcţionează oriunde (d.e. nu se pot accesa indicii de tablouri)
- Exemplu

```
public class ForEachDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = {32, 87, 3, 589, 12, 1076};
        for (int element : arrayOfInts) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

```
Afișare la ieșirea standard:
32 87 3 589 12 1076
```

Anatomia unei clase

```
public class Taxi{
   private int km;
   public Taxi(){
      km = 0;
   }

   public int getKm(){
      return km;
   }
    public void drive(int km){
      this.km += km;
   }
}
```

Antetul clasei

Variabile instanţă (câmpuri)

Constructori

Metode

Constructor:

Scop	Iniţializează starea unui obiect: - se iniţializează toate atributele obiectului - dacă acestea nu se iniţializează explicit, ele vor fi iniţializate implicit cu valorile iniţiale din Java, în funcţie de tipul lor
Nume	La fel cu numele clasei Prima literă mare
Cod	`public Taxi(){…}`
Ieşire	Nu este tip de retur în antet
Intrare	0 sau mai mulţi parametri
Utilizare	`> Taxi cab;` `> cab = new Taxi();`
# de apelări	Cel mult o dată pe obiect; invocat de operatorul "new"

Constructori multipli

```
public class Taxi{
 private int km;
 private String driver;

 public Taxi(){
    km = 0;
    driver = "Unknown";
 }

 public Taxi(int km,String d){
    this.km = km;
    driver = d;
 }
```

O operaţie "new" încununată de succes creează un obiect pe heap şi execută constructorul al cărui listă de parametri "corespunde" listei sale de argumente (ca număr, tip, ordine).

```
> Taxi cab1;
> cab1 = new Taxi();
```

```
> Taxi cab2;
> cab2 = new Taxi(10,"Jim");
```

Folosirea corespunzătoare a constructorilor

- Un constructor ar trebui *întotdeauna* să creeze obiectele într-o stare *validă*
  - Constructorii nu trebuie să facă nimic altceva decât să creeze obiecte
  - Dacă un constructor nu poate garanta că obiectul construit este valid, atunci ar trebui să fie private şi accesat prin intermediul unei metode de fabricare
  - Notă: în general, termenul de **metodă de fabricare** este folosit pentru a se referi la orice metodă al cărei scop principal este crearea de obiecte

Folosirea corespunzătoare a constructorilor

- O ***metodă de fabricare*** **(factory method)**: metodă statică care apelează un constructor
  - Constructorul este de obicei private
  - Metoda de fabricare poate determina dacă să invoce sau nu constructorul
  - Metoda de fabricare poate arunca o excepție, sau poate face altceva potrivit în cazul în care i se dau argumente ilegale sau nu poate crea un obiect valid
  - Exemplu

```
public static Person create(int age) {
  if (age < 0)
       throw new IllegalArgumentException("Too young!");
   else
       return new Person(age);
}
```

Metodă:

Scop	Execută comportamentul obiectului
Nume	Un **verb**; Începe cu literă mică

Cod	`public void turnLeft(){`
	`…`
	`}`

Ieşire	Pentru ieşire este nevoie de un tip returnat
Intrare	0 sau mai mulţi parametri
Utilizare	`> cab.turnLeft();`
# de apelări	nelimitat pentru un obiect

Ce poate face o metodă?

- O metodă poate
  - Schimba starea obiectului său
  - Raporta starea obiectului său
  - Opera asupra numerelor, a textului, a fişierelor, graficii, paginilor web, hardware, …
  - Crea alte obiecte
  - Apela o metodă a unui alt obiect:
    ```
    obiect.metoda(args)
    ```
  - Apela o metodă din aceeaşi clasă:
    ```
    this.metoda(args);
    ```
  - Se poate autoapela (recursivitate)

Declararea unei metode

```
public tip_returnat numeMetoda(0+ parametri) {
    ...
}
```

- **Nume:** Verb care începe cu literă mică
- **Ieşire:** e nevoie de un tip returnat
- **Intrare:** 0 sau mai mulţi parametri
- **Corp:**
  - Între acolade
  - Conţine un număr arbitrar de instrucţiuni
  - Poate conţine declaraţii de "variabile locale"
- **Cum se apelează:** operatorul "punct":

```
numeObiect.numeMetoda(argumente)
```

Metode accesoare şi mutatoare

```
public class Taxi{
  private int km;

  public Taxi(){
    km = 0;
 }

  // raportează # km
  public int getKm(){
    return km;
  }

 // setează (schimbă) # km
 public void setKm(int m){
    km = m;
 }
}
```

Apeluri de metode accesoare (de obţinere)/ mutatoare(de setare)

```
> Taxi cab1;
> cab1 = new Taxi();
> cab1.getKm()
0
> cab1.setKm(500);
> cab1.getKm()
500
```

Intrarea pentru o metodă

- O metodă poate primi 0 sau mai multe intrări
- Intrările pe care le aşteaptă o metodă sunt specificate via lista de "parametri formali" `(tip nume1, tip nume2, …)`
- La apelul unei metode, numărul, ordinea şi tipul argumentelor trebuie să se potrivească cu parametrii corespunzători

Declararea metodei (cu parametri)	Apelul metodei (cu argumente)
`public void meth1(){..}`	`obj.meth1()`
`public int meth2(boolean b){..}`	`obj.meth2(true)`
`public int meth3(int x,int y,Taxi t){..}`	`obj.meth3(3,4,cab)`

Ieşirea unei metode

- O metodă poate să nu aibă ieşire (void) sau să aibă ceva
- Dacă nu returnează nimic (nu are ieşire), tipul returnat este "void"

```
public void setKm(int km){..}
```

- Dacă are ieşire:
  - Tipul returnat este non-void (d.e. int, boolean, Taxi)

```
public int getkm(){..}
```

  - Trebuie să conţină o instrucţiune return cu o valoare

```
// valoarea returnată trebuie să se
// potrivească cu tipul returnat
return km;
```

Modificatori de acces pentru metode

- **private** – nu poate fi folosită de toate clasele; metoda este vizibilă doar în cadrul clasei
- **default (nu se specifică nimic)** – nu poate fi folosită de toate clasele; metoda este vizibilă doar în pachetul din care face parte clasa
- **protected** – nu poate fi folosită de toate clasele; metoda este vizibilă doar în pachetul din care face parte clasa și în subclasele acesteia chiar dacă sunt în alte pachete
- **public**- cel mai frecvent folosit; metoda este vizibilă tuturor
- **static** – nu e nevoie de obiecte pentru a folosi acest fel de metode
  - Dacă declaraţia metodei conţine modificatorul **static**, este vorba de o metodă la nivelul clasei
  - Metodele la nivelul clasei pot accesa doar constantele şi variabilele clasei

Supraîncărcarea unei metode

Supraîncărcare:

- Implică folosirea unui termen pentru a indica semnificaţii diverse
- Scrierea unei metode cu acelaşi nume, dar cu *argumente diferite*
- Supraîncărcarea unei metode Java înseamnă scrierea mai multor metode cu acelaşi nume
- Exemplu:

```
public int test(int i, int j){
    return i + j;
}
public int test(int i, byte j){
    return i + j;
}
```

Ambiguitate la supraîncărcare

- La supraîncărcarea unei metode există riscul **ambiguităţii**
- O situaţie ambiguă este când compilatorul nu poate determina ce metodă să folosească
- Exemplu:

```
public int test(int units, int pricePerUnit)
{
    return units * pricePerAmount;
}
public long test(int numUnits, int weight)
{
    return (long)(units * weight);
}
```

- O supraîncărcare validă necesită cel puțin un argument de tip diferit sau un număr diferit de argumente

Supraîncărcarea constructorilor

- Java furnizează automat un constructor la crearea unei clase
- Programatorii pot scrie constructori proprii, inclusiv constructori care primesc argumente
  - Asemenea argumente sunt folosite la iniţializare atunci când valorile obiectelor pot diferi
- Exemplu

```
class Client {
  private String nume;
  private int numarCont;
  public Client(String n) {
    nume = n;
  }
  public Client(String n, int a) {
    nume = n;
    numarCont = a;
  }
```

Supraîncărcarea constructorilor

- Dacă o clasă este instanţiabilă, Java furnizează automat un constructor
- La crearea unui constructor propriu, constructorul furnizat automat *încetează să existe*
- Ca şi la alte metode, constructorii pot fi *supraîncărcaţi*
  - Supraîncărcarea constructorilor oferă o cale de a crea obiecte cu sau fără argumente iniţiale, după necesităţi

Referinţa `this`

- Compilatorul accesează câmpurile de date corespunzătoare ale obiectului, deoarece i se trimite implicit o referinţă **`this`** la câmpurile și metodele claselor

```
class Student{
 private int studentID;
 public Student(int studentID){
    this.studentID = studentID;
 }
 public Student(){
     this(0);
 }
 public getStudentID(){
   return this.studentID; //return studentID;
 }
}
```

- Metodele statice (metodele la nivel de clasă) nu au o referinţă **`this`** deoarece nu au un obiect asociat

Clasă instanţiabilă

- O clasă este *instanţiabilă* dacă putem crea instanţe ale clasei respective
- Exemple: clasele învelitoare ale tipurilor primitive, **String, Scanner,...** sunt toate instanţiabile, în timp ce clasa **Math** nu este
- Fiecare obiect este membru al unei clase
  - catedra este un obiect membru al clasei Catedra
  - relaţii de tipul **este-o**

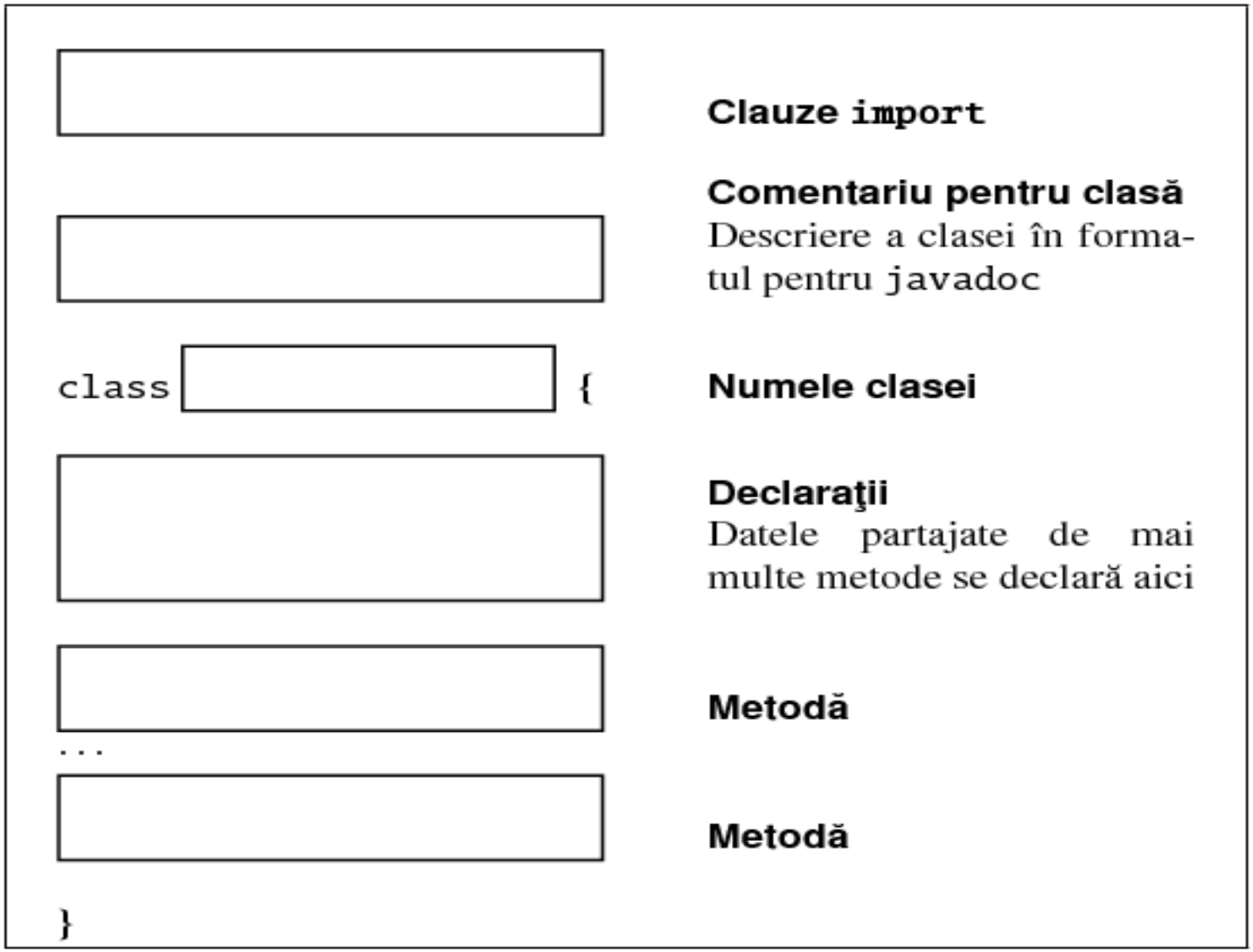
Utilitatea conceptului de clasă

- Clasa reprezintă șablonul cu atributele și functionalitățile obiectelor
- Toate obiectele au atribute predictibile deoarece obiectele sunt membre ale anumitor clase
- Pentru crearea unui obiect de tipul unei clase, trebuie să creaţi clase din care să se instanţieze obiectele
  - Exemplu - clasa Taxi și instanța t: Taxi t = new Taxi();
- Se pot scrie alte clase care să folosească obiectele
  - Exemplu - se scrie un program/o clasă pentru a conduce taxiul la aeroport; foloseşte clasa Taxi pentru a crea un obiect Automobil de condus

Crearea unei clase

- Trebuie:
  - Să daţi un nume clasei
  - Să determinaţi ce date şi metode vor fi parte a clasei
- Modificatorii de acces pentru clase cuprind:
  - **public**
    - Cel mai folosit; clasa este vizibilă tuturor
    - Clasa poate fi **extinsă** sau folosită ca bază pentru alte clase
  - **final** – folosit doar în circumstanţe speciale (clasa este complet definită şi nu sunt necesare definiri de subclase)
  - **abstract** – folosit doar în circumstanţe speciale (clasa este incomplet definită – conţine o metodă neimplementată)

Şablon de program pentru codul unei clase



Blocuri şi vizibilitate (scop)

- **Bloc**: în orice clasă sau metodă, conține codul inclus între o pereche de acolade
- Porţiunea de program în care se poate referi o variabilă constituie **domeniul de vizibilitate** (**scop**) al variabilei
- O variabilă există, sau devine vizibilă la declarare
- O variabilă încetează să existe sau devine invizibilă la sfârşitul blocului în care a fost declarată
- Dacă declaraţi o variabilă într-o clasă şi folosiţi acelaşi nume pentru a declara o variabilă într-o metodă a clasei, variabila declarată în metodă are precedenţa sau **suprascrie**, prima variabilă

Variabile la nivel de clasă

- Variabile la nivel de clasă: variabile care sunt partajate de fiecare instanţă a unei clase
- Numele instituţiei = "T.U. Cluj-Napoca"
- Fiecare angajat al unei instituţii lucrează pentru aceeaşi instituţie

```
private static String COMPANY_ID =
                              "T.U. Cluj-Napoca";
```

  - Se poate dar nu este recomandat să se declare o variabilă vizibilă în afara clasei

Folosirea constantelor şi metodelor importate automat, de bibliotecă

- Creatorii limbajului Java au creat cca. 500 clase
  - Exemple: System, Scanner, Character, Boolean, Byte, Short, Integer, Long, Float şi Double sunt clase
- Aceste clase sunt stocate în pachete (biblioteci) de clase
- Un pachet este un instrument de grupare convenabilă a claselor cu funcţionalitate înrudită
- `java.lang` – pachetul este importat implicit în fiecare program Java care conţine clase fundamentale (de bază) – d.e. System, Character, Boolean, Byte, Short, Integer, Long, Float, Double, String

Folosirea metodelor importate, de bibliotecă

- Pentru a folosi oricare dintre clasele de bibliotecă (altele decât java.lang):
  - Folosiţi întreaga cale împreună cu numele clasei

```
area = Math.PI * radius * radius;
```

  sau
  - Importaţi clasa

  sau
  - Importaţi pachetul care conţine clasa pe care o folosiţi
- Pentru a importa un întreg pachet folosiţi simbolul de nume global, *
- Exemplu
  - import java.util.*;
  - Reprezintă toate clasele dintr-un pachet

Metode statice

- În Java se pot declara metode şi variabile care să aparţină *clasei*, nu obiectului. Aceasta se face prin declararea lor ca **static**
- Metodele statice: declarate inserând cuvântul cheie `static` imediat după specificatorul de vizibilitate

```
public class ArrayWorks {
   public static double medie(int[] arr)  {
            double total = 0.0;
            for (int k=0; k<arr.length; k++)  {
                  total = total + arr[k];
            }
      return  total / arr.length;
   }
}
```

Metode statice

- Metodele statice sunt apelate folosind numele clasei în locul unei referinţe la un obiect

```
int[] myArray = {10, 21, 1};
...
double media = ArrayWorks.medie(myArray);
```

- Exemple metode statice utile: în clasa Java standard, numită **Math**

```
public class Math {
  public static double abs(double d) {...}
  public static int abs(int k) {...}
  public static double pow(double b, double exp){...}
  public static double random() {...}
  public static int round(float f) {...}
  public static long round(double d) {...}
  …
}
```

Restricţii pentru metodele statice

- Corpul unei metode statice *nu poate* referi nici o variabilă instanţă (ne-statică)
- Corpul unei metode statice *nu poate* invoca nici o metodă ne-statică
- Dar, corpul unei metode statice *poate* instanţia obiecte

```
public class Go {
    public static void main(String[] args)   {
        Greeting greeting = new Greeting();
    }
}
```

- Aplicaţiile Java de sine stătătoare, trebuie să-şi iniţieze execuţia dintr-o metodă statică numită întotdeauna *main* şi care are un singur tablou de String-uri ca parametru

Variabile statice

- Orice *variabilă instanţă* poate fi declarată **static** prin includerea cuvântului-cheie **`static`** imediat înainte de specificarea tipului său

```
public class StaticStuff {
    public static double staticDouble;
    public static String staticString;
    . . .
}
```

- O variabilă statică
  - Poate fi referită fie de clasă fie de obiect
  - Instanţierea unui nou obiect de acelaşi tip nu sporeşte numărul de variabile statice

Exemple cu variabile statice

```
StaticStuff s1, s2;
s1 = new StaticStuff();
s2 = new StaticStuff();
s1.staticDouble = 3.7;
System.out.println( s1.staticDouble );
System.out.println( s2.staticDouble );
s1.staticString = "abc";
s2.staticString = "xyz";
System.out.println( s1.staticString );
System.out.println( s2.staticString );
```

Exemple cu constante

- Exemplu de constante în clasa standard Java numită **Color**

```
public class Color {
  public static final Color BLACK = new Color(0,0,0);
  public static final Color BLUE = new Color(0,0,255);
  public static final Color GREEN= new Color(0,255,0);
    . . .
}
```

- Constantele Color
  - Ajută la compararea culorilor folosind numele asignat lor

```
Color myColor;
...
if (myColor == Color.GREEN) ...
```