Programare orientată pe obiecte

1. Câteva observaţii despre operatori
2. Structuri de control în Java
3. Clase şi Obiecte

Operatori

- Sunt trataţi în detaliu la laborator
- Câteva diferenţe faţă de C:
  - Operatorul pe biţi >>>
    - D.e. n >>> p; // deplasează biţii lui *n* spre dreapta cu *p* poziţii. În poziţiile de rang superior se inserează zerouri.
  - Concatenarea pentru String: operatorul +
  - Operatori pentru lucrul cu obiecte – îi vom trata 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 numite *expresie booleană.*

```
if (<expresie booleană>)
  <then bloc>
else
  <else bloc>
```

- **`<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 în Java sunt obiecte ale clasei String
  - Clasa String furnizează metode de comparare (am văzut deja)
- 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
```

Sugestii pentru lanţuri de **if**

- Toate condiţiile ar trebui să fie înrudite apropiat
- Puneţi cazurile comune mai întâi (acolo unde se poate)
- Folosiţi **`switch`** dacă se poate

```
// Cod bun
if (rnd < 0) {
// Error!
} else if (rnd < 0.1) {
// ...
} else if (rnd < 0.5) {
// ...
} else if (rnd < 1.0) {
// ...
} else
// Error!
```

```
// Cod rau
if (screen.needsRepaint()) {
   // redesenează ecranul
} else if (p1.canMove()) {
   // ia mutarea de la p1
} else if (p2.canMove()) {
   // ia mutarea de la p2
} else {
   // blocaj!
}
```


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,** sau **int**.
- Valoarea lui **`<expresie aritmetică>`** se compară cu constanta **`eticheta_i`** din **`<case eticheta_i>`**.

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

```
switch (file.getType()) {

   //caz cu tratare comuna
   // Non-breaking case
   case IMAGE_PNG:
   case IMAGE_JPG:
   openWithPaint(file);
   break;

case IMAGE_WMF:
   displayWMF(file);
   break;
default:
   // Tip necunoscut
   break;
}
```


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 alta capcană frecventă.

Sugestii pentru ciclurile **while**

- Se folosesc pentru ciclări mai complicate
  - Evitaţi să aveţi mai mult de o ieşire din buclă
  - Sunt permise ieşirile forţate (pentru e evita duplicarea codului)

```
// Cod rău
stmts_A
while (boolExp) {
  stmts_B
  stmts_A
}
```

```
// Cod bun
while (true) {
  stmts_A
  if (!boolExp)
    break;
  stmts_B
}
```


Instrucţiunea **do-while**

- Instrucţiunea **while** este o *buclă cu pre-testare* (testul se face la intrarea în buclă). De aceea, 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 false.

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*. ea 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
  - Un **break** cu etichetă se termină întotdeauna anormal; 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

continue cu etichetă – exemplu de la Sun

```
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 de la Sun:

```
public class ForEachDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12,
                              1076, 2000, 8, 622, 127 };

        for (int element : arrayOfInts) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```


Sugestii generale pentru bucle

- Rulaţi bucla în minte (verificaţi cazurile din capete)
- Folosiţi nume cu semnificaţie
  - Folosiţi nume ca `i, j, n` doar în buclele scurte unde variabila de ciclu este doar un index
- Evitaţi mai mult de trei cicluri imbricate (lucru valabil şi pentru imbricările de if)
  - Restructuraţi sau divizaţi în funcţii ajutătoare
- Nu folosiţi variabila ciclului *după sfârşitul* buclei

Instrucţiuni de tratare a excepţiilor

- Deocamdată doar semnificaţia (în detaliu, ulterior)
  - **`throw`** – aruncă o excepţie
  - **`try-catch`**, and **`finally`** – folosite la tratarea excepţiilor
  - **`try`** – identifică blocul de instrucţiuni unde poate fi aruncată o excepţie
  - **`catch`** – asociată cu **`try`**; identifică un bloc de instrucţiuni care pot trata un anumit tip de excepţie; se execută dacă apare o excepţie în blocul **`try`**
  - **`finally`** – asociată cu **`try`**; identifică un bloc de instrucţiuni care se execută indiferent de apariţia sau nu a unei erori în blocul **`try`**.
- Excepţiile nu trebuie folosite pentru a simula un goto!

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
Nume	La fel cu numele clasei. Prima literă mare, stil "cămilă"

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 lista 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) este o 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

```
public static Person create(int age) { //exemplu: metodă de fabricare
   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, stil "cămilă"

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

Ieşire	Pentru ieşire este nevoie de un tip returnat
Intrare	0 sau mai mulţi parametri
Uutilizare	`> 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){..}
public int getKM() {..}
public void increaseSpeed(int accel, int limit){..}
```

- **Nume:** Verb care începe cu literă mică, stil "cămilă"
- **Ieşire:** e nevoie de un tip returnat.
- **Intrare:** 0 sau mai mulţi parametri
- **Corp:**
  - Între acolade
  - Conţine un # arbitrar de instrucţiuni (asignare, "if", return, etc.).
  - Poate conţine declaraţii de " variabile locale"
- **Cum se apelează:** operatorul "punct":

```
numeObiect.numeMetoda(argumente)
cab1.increaseSpeed(5, 50)
```


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();
> cab.getKm()
0
> cab.setKm(500);
> cab.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

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


Modificatori de acces pentru metode

- **public**- cel mai frecvent folosit; metoda este vizibilă tuturor
- **private** – nu poate fi folosită de toate clasele; metoda este vizibilă doar în cadrul clasei
- **protected** – nu poate fi folosită de toate clasele; metoda este vizibilă doar în pachetul din care face parte
- **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

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**

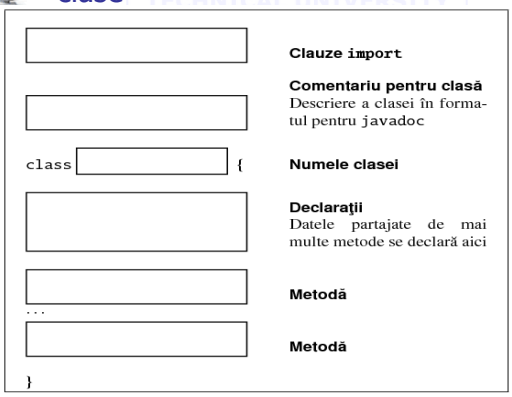
Utilitatea conceptului de clasă

- Conceptul este util deoarece:
  - Obiectele moştenesc atribute din clase
  - Toate obiectele au atribute predictibile deoarece obiectele sunt membre ale anumitor clase
- Trebuie:
  - Să creaţi clase din care să se instanţieze obiectele (d.e. clasa Taxi)
  - Să scrieţi alte clase care să folosească obiectele (se scrie un program/o clasă pentru a conduce taxiul la aeroport; foloseşte d.e. 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)
  - Ultimii doi îi vom trata mai târziu

Şablon de program pentru codul unei clase



Blocuri şi vizibilitate (scop)

- **Bloc**: în orice clasă sau metodă, 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 variabila 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

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 că scrieţi mai multe 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);
}
```


Argumente pentru constructori

- 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
- Dar, 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

- Clasele pot creşte foarte repede
  - Fiecare clasa poate avea multe câmpuri de date şi multe metode
- La instanţierea multor obiecte ale unei clase, cerinţele de memorie pot fi substanţiale
  - Nu este nevoie să se stocheze o copie separată a fiecărei variabile şi metode la fiecare instanţiere a unei clase

Referinţa this

- Compilatorul accesează câmpurile de date corespunzătoare ale obiectului, deoarece i se trimite implicit o referinţă `this` la metodele claselor
- Metodele statice (metodele la nivel de clasă) nu au o referinţă `this` deoarece nu au un obiect asociat

```
public getStudentID()
{
  return studentID;
}
public getStudentID()
{
  return this.studentID;
}
```


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 sunt declarate inserând cuvântul cheie "static" imediat după specificatorul de vizibilitate (*public*, *private* sau *protected*).

```
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;
   }
}
```

Metodele statice

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

```
double myArray = {1.1, 2.2, 3.3};
...
double media = ArrayStuff.medie(myArray);
```

- Exemple metode statice utile: in 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 asupra metodelor statice

- Corpul unei metode statice *nu poate* referi nici o variabilă instanţă (nestatică).
- 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 declarata **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.

Example 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 );
```


De ce metode şi variabile statice?

- Metodele statice sunt utile în situatii în care datele trebuie partajate între mai multe obiecte de acelaşi tip.
- Un exemplu bun de utilitate a unei metode statice este 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 darkGray = new
                          Color(64,64,64);
  . . .
}
```

- Constantele Color sunt şi static şi final => le putem compara

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


Rezumat

- Controlul execuţiei
  - if, switch, while, for, do –while
  - break/continue cu etichetă
  - for each
- Metode:
  - tipuri (accessoare, mutatoare)
  - intrare, ieşire
  - modificatori de acces
  - supraîncărcare
- Clase
  - constructori
  - clase instanţiabile
  - creare, supraîncărcarea constructorilor
  - referinţa **this**
  - Variabile la nivel de clasă
- Static
  - metode
  - variabile