



Programare orientată pe obiecte

1. Câteva observații despre operatori
2. Structuri de control în Java
3. Clase și Obiecte



Operatori

- Sunt tratați în detaliu la laborator
- Câteva diferențe față de C:
 - Operatorul pe biți >>>
 - D.e. `n >>> p;` // deplasează biții lui `n` spre dreapta cu `p` poziții. În pozițiile de rang superior se inserează zerouri.
 - Concatenarea pentru String: operatorul `+`
 - Operatori pentru lucrul cu obiecte – îi vom trata în detaliu mai târziu



Despre precedența operatorilor

Precedența operatorilor	
<code>. [] (args) post ++ --</code>	Tineți minte precedența
<code>! ~ unary + - pre ++ --</code>	pentru operatorii unari,
<code>(type) new</code>	<code>*/%</code>
<code>* / %</code>	<code>+ -</code>
<code>+ -</code>	<i>comparații</i>
<code><< >> >>></code>	<code>&& </code>
<code>< <= > >= instanceof</code>	<code>=</code> <i>asignări</i>
<code>== !=</code>	Folosiți paranteze ()
<code>&</code>	pentru ceilalți
<code>^</code>	
<code> </code>	
<code>&&</code>	
<code> </code>	
<code>? :</code>	
<code>= += -= etc</code>	



Instrucțiunea if

- Instrucțiunea **if** specifică ce bloc de cod să se execute în funcție de rezultatul evaluării unei condiții numite *expresie booleană*.


```
if (<expresie booleană>
    <then bloc>
else
    <else bloc>
```
- **<expresie booleană>** este o expresie condițională care se evaluează la true sau false.
 - Sintaxă similară limbajului C, dar atenție la ce este o expresie booleană în Java



Compararea obiectelor

- La compararea a două **variabile**, se compară *conținutul* lor.
- În cazul **obiectelor**, *conținutul* este *adresa* unde este stocat obiectul (adică se vor compara referințele).
 - Șirurile în Java sunt obiecte ale clasei String
 - Clasa String furnizează metode de comparare (am văzut deja)
- Cea mai bună metodă în ceea ce privește compararea obiectelor este să *definim metode de comparare* pentru clasa respectivă.



Sugestii pentru if

- Începeți testul cu cazul cel mai relevant
 - Face codul mai ușor de citit
- Nu uitați de clauza else!
- Evitați condițiile complicate
- Divizați condițiile în variabile/funcții booleene
- Încercați să folosiți condiții 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 lanțuri de if

- Toate condițiile ar trebui să fie înrudite apropiat
- Puneți cazurile comune mai întâi (acolo unde se poate)
- Folosiți **switch** dacă se poate

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



Instrucțiunea switch

- Sintaxa pentru instrucțiunea **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 >**.



Instrucțiunea **switch**

- Dacă există o valoare care se potrivește cu valoarea expresiei, atunci se execută corpul acesteia. Altfel execuția continuă cu instrucțiunea care urmează instrucțiunii **switch**
- Instrucțiunea **break** face să nu se execute porțiunea care urmează din **switch**, ci să se continue cu ceea ce vine după **switch**.
- **break** este necesar pentru a executa instrucțiunile dintr-un caz, și numai unul.
 - iarăși, ca în C



Sugestii pentru **switch**

- Ordonăți cazurile (logic sau alfabetic)
 - Prevedeți întotdeauna cazul implicit (**default**)
 - Întotdeauna folosiți **break** între cazuri
 - Încercați să păstrați mică dimensiunea instrucțiunii **switch**
 - Divizați cazurile de mari dimensiuni în funcții

```
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;
}
```



Instrucțiuni repetitive

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



Instrucțiunea **while**

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


```
while ( <expresie booleana> )
    <instrucțiuni>
```
- 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 întâlnește o valoare stabilă, numită *santinelă*.



Capcane în scrierea instrucțiunilor repetitive

- La instrucțiunile repetitive este important să se asigure terminarea ciclului la un moment dat.
- Tipurile de probleme potențiale de ținut minte:
- Bucula infinită**

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

- Deoarece **item** este inițializat 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 instrucțiunilor repetitive

Bucula infinită

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

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



Capcane în scrierea instrucțiunilor repetitive

Depășirea de capacitate

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



Sugestii pentru ciclurile **while**

- Se folosesc pentru ciclări mai complicate
 - Evitați să aveți mai mult de o ieșire din buclă
 - Sunt permise ieșirile forțate (pentru a evita duplicarea codului)

<pre>// Cod rău stmts_A while (boolExp) { stmts_B stmts_A }</pre>	<pre>// Cod bun while (true) { stmts_A if (!boolExp) break; stmts_B }</pre>
---	---



Instrucțiunea **do-while**

- Instrucțiunea **while** este o *buclă cu pre-testare* (testul se face la intrarea în buclă). De aceea, corpul buclei poate să nu fie executat niciodată.
- Instrucțiunea **do-while** este o *buclă cu post-testare*. Corpul ciclului este executat cel puțin o dată.
- Formatul instrucțiunii **do-while** este:


```
do
  <instructiuni>
while (<expresie booleană>);
```
- `<instructiuni>` se execută până când `<expresie booleană>` devine false.



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

- Atenție la două lucruri la folosirea buclelor întrerupte:
 - **Pericolul ciclării infinite.** Expresia booleană a instrucțiunii `while` este `true` întotdeauna. Dacă nu există o instrucțiune `if` care să forțeze ieșirea din ciclu, rezultatul va fi o buclă infinită.
 - **Puncte de ieșire multiple.** Se poate totuși, chiar dacă e mai complicat să se scrie un control buclă cu mai multe ieșiri (`break`). Practica recomandă pe cât posibil curgerea *o-intrare o-ieșire* a controlului.



Instrucțiunea **for**

- Formatul instrucțiunii **for** este:


```
for (<initializare>; <expresie booleană>; <increment>)
  <instructiuni>
```
- Exemplu:


```
int sum = 0;
for (int i = 1; i <=100; i++){
    sum += i;
}
```
- Variabila `i` din exemplu se numește *variabilă de control*. ea contorizează numărul de repetiții.
- `<increment>` poate fi orice cantitate.
- Din nou, ca în C



Sugestii pentru ciclurile **for**

- Sunt ideale atunci când numărul de iterații este cunoscut
 - Folosiți o instrucțiune pentru fiecare parte
 - Declarați variabilele de ciclu în antet (reduce vizibilitatea și interferența)
 - Nu se recomandă modificarea variabilei de control în corpul ciclului



break cu etichetă

- **break** se folosește în **bucle** și **switch**
 - semnificațiile sunt diferite
- **break** poate fi și urmat de o etichetă, L
 - Încearcă să transfere controlul la o instrucțiune etichetată cu L
 - Un **break** cu etichetă se termină întotdeauna anormal; dacă nu există instrucțiuni etichetate cu L, apare o eroare de compilare
 - Un **break** cu etichetă permite ieșirea din mai multe bucle imbricate
 - Eticheta trebuie să precedă bucla cea mai exterioară din care se dorește ieșirea
 - 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 instrucțiunii continue sare peste iterația curentă a unei bucle exterioare marcate cu o etichetă dată.
- Eticheta trebuie să precedă bucla cea mai exterioară din care se dorește ieșirea

```
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");
    }
}
```

continue cu etichetă – exemplu de la Sun



for pentru iterații peste colecții și tablouri

- Creat special pentru iterații peste colecții și tablouri (vom reveni asupra instrucțiunii) – Java 5
- Nu funcționează 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

- Rulați bucla în minte (verificați cazurile din capete)
- Folosiți nume cu semnificație
 - Folosiți nume ca *i*, *j*, *n* doar în buclele scurte unde variabila de ciclu este doar un index
- Evitați mai mult de trei cicluri imbricate (lucru valabil și pentru imbricările de if)
 - Restructurați sau divizați în funcții ajutătoare
- Nu folosiți variabila ciclului *după sfârșitul* buclei



Instrucțiuni de tratare a excepțiilor

- Deocamdată doar semnificația (în detaliu, ulterior)
 - **throw** – aruncă o excepție
 - **try-catch**, and **finally** – folosite la tratarea excepțiilor
 - **try** – identifică blocul de instrucțiuni unde poate fi aruncată o excepție
 - **catch** – asociată cu **try**; identifică un bloc de instrucțiuni care pot trata un anumit tip de excepție; se execută dacă apare o excepție în blocul **try**
 - **finally** – asociată cu **try**; identifică un bloc de instrucțiuni care se execută indiferent de apariția sau nu a unei erori în blocul **try**.
- Excepțiile 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 instanță (câmpuri)

Constructorii

Metode



Constructor:

Scop	Inițializează starea unui obiect
Nume	La fel cu numele clasei. Prima literă mare, stil "cămilă"
Cod	<pre>public Taxi(){ ... }</pre>
Ieșire	Nu este tip de retur în antet
Intrare	0 sau mai mulți parametri
Utilizare	<pre>> Taxi cab; > cab = new Taxi();</pre>
# de apelări	Cel mult o dată pe obiect; invocat de operatorul "new"



Constructorii 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 operație "new" încununată de succes creează un obiect pe heap și 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** și 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	<code>public void turnLeft(){</code> ... <code>}</code>
Ieșire	Pentru ieșire este nevoie de un tip returnat
Intrare	0 sau mai mulți parametri
Utilizare	<code>> cab.turnLeft();</code>
# 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 fișierelor, graficii, paginilor web, hardware, ...
 - Crea alte obiecte
 - Apela o metodă a unui alt obiect:
`obiect.metoda(args)`
 - Apela o metodă din aceeași 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"
- **Ieșire:** e nevoie de un tip returnat.
- **Intrare:** 0 sau mai mulți parametri
- **Corp:**
 - Între acolade
 - Conține un # arbitrar de instrucțiuni (asignare, "if", return, etc.).
 - Poate conține declarații de "variabile locale"
- **Cum se apelează:** operatorul "punct":
`numeObiect.numeMetoda(argumente)`
`cab1.increaseSpeed(5, 50)`



Metode accesoare și 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 obținere)/
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 așteaptă o metodă sunt specificate via lista de "parametri formali" (`tip nume1, tip nume2, ...`)
- La apelul unei metode, numărul, ordinea și tipul argumentelor trebuie să se potrivească cu parametrii corespunzători.

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



Ieșirea unei metode

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


```
public void setKm(int km){..}
```
- Dacă are ieșire:
 - Tipul returnat este non-void (d.e. `int`, `boolean`, `Taxi`)


```
public int getkm(){..}
```
 - Trebuie să conțină o instrucțiune `return` cu o valoare


```
// valoarea 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 declarația metodei conține modificatorul **static**, este vorba de o metodă la nivelul clasei.
 - Metodele la nivelul clasei pot accesa doar constantele și variabilele clasei



Clasă instanțiabilă

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



Utilitatea conceptului de clasă

- Conceptul este util deoarece:
 - Obiectele moștenesc atribute din clase
 - Toate obiectele au atribute predictibile deoarece obiectele sunt membre ale anumitor clase
- Trebuie:
 - Să creai clase din care să se instanțieze obiectele (d.e. clasa Taxi)
 - Să scrieți alte clase care să folosească obiectele (se scrie un program/o clasă pentru a conduce taxiul la aeroport; folosește d.e. clasa Taxi pentru a crea un obiect automobil de condus)



Crearea unei clase

- Trebuie:
 - Să dați un nume clasei
 - Să determinați ce date și 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 circumstanțe speciale (clasa este complet definită și nu sunt necesare definiții de subclase)
 - **abstract** – folosit doar în circumstanțe speciale (clasa este incomplet definită – conține o metodă neimplementată)
 - Ultimii doi îi vom trata mai târziu



Șablon de program pentru codul unei clase

<input type="text"/>	Clauze import
<input type="text"/>	Comentariu pentru clasă Descriere a clasei în formatul pentru javadoc
class <input type="text"/> {	Numele clasei
<input type="text"/>	Declarații Datele partajate de mai multe metode se declară aici
<input type="text"/>	Metodă
...	
<input type="text"/>	Metodă
}	



Blocuri și vizibilitate (scop)

- **Bloc:** în orice clasă sau metodă, codul inclus între o pereche de acolade
- Porțiunea 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 sfârșitul blocului în care a fost declarată
- Dacă declarați o variabilă într-o clasă și folosiți același nume pentru a declara o variabilă într-o metodă a clasei, variabila declarată în metodă are precedența sau **suprascris**, prima variabilă



Supraîncărcarea unei metode

Supraîncărcare:

- Implică folosirea unui termen pentru a indica semnificații diverse
- Scrierea unei metode cu același nume, dar cu *argumente diferite*
- Supraîncărcarea unei metode Java înseamnă că scrieți mai multe metode cu același 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 **ambiguității**
- O situație 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 inițializare 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 instanțiable, Java furnizează automat un constructor
- Dar, la crearea unui constructor propriu, constructorul furnizat automat *încetează să existe*
- Ca și la alte metode, constructorii pot fi *supraîncărcați*
 - Supraîncărcarea constructorilor oferă o cale de a crea obiecte cu sau fără argumente inițiale, după necesități.



Referința `this`

- Clasele pot crește foarte repede
 - Fiecare clasa poate avea multe câmpuri de date și multe metode
- La instanțierea multor obiecte ale unei clase, cerințele de memorie pot fi substanțiale
 - Nu este nevoie să se stocheze o copie separată a fiecărei variabile și metode la fiecare instanțiere a unei clase



Referința `this`

- Compilatorul accesează câmpurile de date corespunzătoare ale obiectului, deoarece i se trimite implicit o referință `this` la metodele claselor
- Metodele statice (metodele la nivel de clasă) nu au o referință `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 instanță a unei clase
- Numele instituției = "T.U. Cluj-Napoca"
- Fiecare angajat al unei instituții lucrează pentru aceeași instituție


```
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 și metodelor importate automat, de bibliotecă

- Creatorii limbajului Java au creat cca. 500 clase
 - Exemple: System, Scanner, Character, Boolean, Byte, Short, Integer, Long, Float și Double sunt clase
- Aceste clase sunt stocate în pachete (biblioteci) de clase.
- Un pachet este un instrument de grupare convenabilă a claselor cu funcționalitate înrudită
- `java.lang` – pachetul este importat implicit în fiecare program Java care conține 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`):
 - Folosiți întreaga cale împreună cu numele clasei
`area = Math.PI * radius * radius;`
 sau
 - Importați clasa
 sau
 - Importați pachetul care conține clasa pe care o folosiți
- Pentru a importa un întreg pachet folosiți simbolul de nume global, *
- Exemplu:
 - `import java.util.*;`
 - Reprezintă toate clasele dintr-un pachet



Metode statice

- În Java se pot declara metode și variabile care să aparțină *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 referințe 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) {...}
    ...
}
```



Restricții asupra metodelor statice

- Corpul unei metode statice *nu poate* referi nici o variabilă instanță (nestică).
- Corpul unei metode statice *nu poate* invoca nici o metodă ne-statică.
- Dar, corpul unei metode statice *poate* instanția obiecte.

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

- Aplicațiile Java de sine stătătoare, trebuie să-și inițieze execuția dintr-o metodă statică numită întotdeauna *main* și care are un singur tablou de String-uri ca parametru.



Variabile statice

- Orice *variabilă instanță* 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
 - Instanțierea unui nou obiect de același tip nu sporește 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 );
```



De ce metode și variabile statice?

- Metodele statice sunt utile în situații în care datele trebuie partajate între mai multe obiecte de același 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 și static și final => le putem compara
- ```
Color myColor;
...
if (myColor == Color.green) ...
```



## Rezumat

- Controlul execuției
  - if, switch, while, for, do -while
  - break/continue cu etichetă
  - for each
- Metode:
  - tipuri (accesoare, mutatoare)
  - intrare, ieșire
  - modificatori de acces
  - supraîncărcare
- Clase
  - constructori
  - clase instanțiabile
  - creare, supraîncărcarea constructorilor
  - referința **this**
  - Variabile la nivel de clasă
- Static
  - metode
  - variabile