



Programare orientată pe obiecte

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



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, creați un obiect de tipul corespunzător folosind valoarea primitivă ca argument
 - Noul obiect va conține o variabilă instanță 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**, și **Character** la valorile primitive corespunzătoare sunt (în ordine) **byteValue**, **shortValue**, **intValue**, **longValue**, **floatValue**, **doubleValue**, și **charValue**
 - Nici una dintre aceste metode nu necesită argumente

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



Boxing și 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;
```

- 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;
```



Constante și metode statice în clasele învelitoare

- Constante pentru valori maxime și 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` și `Boolean.FALSE` sunt obiectele booleene corespunzătoare valorilor `true` și `false` ale tipului primitiv `boolean`



Constante și metode statice în clasele învelitoare

- Metode statice pentru conversia unei reprezentări corect formate ca șir de caractere a unui număr la numărul de un tip dat
 - Metodele `Integer.parseInt`, `Long.parseLong`, `Float.parseFloat` și `Double.parseDouble` sunt pentru (în ordine) `int`, `long`, `float` și `double`
- Metode statice pentru conversia duală celei anterioare
 - Exemplu. Expresia
`Double.toString(123.99);`
returnează valoarea șir "`123.99`"
- Clasa `Character` conține un număr de metode utile la prelucrarea șirurilor



Referințe Java

- Toate obiectele sunt create în memorie
- Pentru a accesa datele dintr-un obiect sau a lucra cu un obiect, este nevoie de o variabilă pe stiva, variabila care poate stoca o referință la adresa obiectului.
- Despre variabilele care stochează referințe la adrese de obiecte se spune că păstrează tipul de date *referință*
- Exemplu:

```
PlatitorTaxe t = new  
    PlatitorTaxe (111111120633, 30000);
```



Clasa `String`

- Nu există un tip primitiv pentru șiruri în Java
- Clasa `String` este o clasă Java predefinită folosită la stocarea și prelucrarea șirurilor
- Obiectele de tipul `String` sunt compuse din șiruri de caractere scrise între ghilimele
 - Orice șir între ghilimele este de clasă `String`
`"Fiti fericiți cu Java."`
- Unei variabile de tipul `String` i se poate da valoarea unui obiect `String`

```
String binecuvintare = "Fiti fericiți cu  
Java.";
```




Concatenarea șirurilor

- *Concatenare*: folosind operatorul `+` operator aplicat asupra a două șiruri pentru a le conecta și a forma un șir mai lung
 - Dacă `salut` este egal cu `"Bună ziua, "`, și `javaClass` este egal cu `"viitori colegi"`, atunci `salut + javaClass` este egal cu `"Bună ziua, viitori colegi"`
- Orice număr de șiruri pot fi concatenate
- La combinarea unui șir cu aproape orice alt tip, rezultatul este un șir
 - `"Răspunsul este " + 42` se evaluează la `" Răspunsul este 42"`



Metode al clasei string String

- Clasa **String** conține multe metode utile la aplicațiile care prelucrează șiruri
 - O metodă a clasei **String** se apelează prin numele unui obiect **String**, un punct, numele metodei și 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());
```
 - *Poziția* sau *indexul* unui caracter într-un șir se numără întotdeauna de la zero



Metode ale clasei `String`

Lungimea șirului

<code>i =</code>	<code>s.length()</code>	lungimea șirului <code>s</code> .
------------------	-------------------------	-----------------------------------

Comparații (note: folosiți aceste metode în loc de `==` și `!=`)

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



Metode ale clasei String

Căutare - Notă : Toate metodele "indexOf" returnează -1 dacă șirul/caracterul nu este găsit

<code>i =</code>	<code>s.indexOf (t)</code>	indexul primei apariții lui String <code>t</code> în <code>s</code> .
<code>i =</code>	<code>s.indexOf (t, i)</code>	indexul lui String <code>t</code> la sau după poziția <code>i</code> în <code>s</code> .
<code>i =</code>	<code>s.indexOf (c)</code>	indexul primei apariții caracterului <code>c</code> în <code>s</code> .
<code>i =</code>	<code>s.indexOf (c, i)</code>	indexul caracterului <code>c</code> la sau după poziția <code>i</code> din <code>s</code> .
<code>i =</code>	<code>s.lastIndexOf (c)</code>	indexul ultimei apariții lui <code>c</code> în <code>s</code> .
<code>i =</code>	<code>s.lastIndexOf (c, i)</code>	indexul ultimei apariții lui <code>c</code> pe sau înainte de poziția <code>i</code> în <code>s</code> .
<code>i =</code>	<code>s.lastIndexOf (t)</code>	indexul ultimei apariții lui <code>t</code> în <code>s</code> .
<code>i =</code>	<code>s.lastIndexOf (t, i)</code>	indexul ultimei apariții a lui <code>t</code> pe sau înainte de poziția <code>i</code> în <code>s</code> .



Metode ale clasei `String`

Obținerea de părți

<code>c =</code>	<code>s.charAt(i)</code>	caracterul de la poziția <i>i</i> din <i>s</i> .
<code>s1 =</code>	<code>s.substring(i)</code>	subșirul de la indexul <i>i</i> până la sfârșitul lui <i>s</i> .
<code>s1 =</code>	<code>s.substring(i, j)</code>	subșirul de la indexul <i>i</i> până ÎNAINTE de indexul <i>j</i> din <i>s</i> .

Crearea unui nou șir din original

<code>s1 =</code>	<code>s.toLowerCase()</code>	nou <code>String</code> cu toate caracterele minuscule
<code>s1 =</code>	<code>s.toUpperCase()</code>	nou <code>String</code> cu toate caracterele majuscule
<code>s1 =</code>	<code>s.trim()</code>	nou <code>String</code> fără spații albe la început și sfârșit
<code>s1 =</code>	<code>s.replace(c1, c2)</code>	nou <code>String</code> cu toate <i>c2</i> -urile înlocuite prin <i>c1</i> .



Metode ale clasei `String`

Metode statice pentru conversia la `String`

<code>s =</code>	<code>String.valueOf(x)</code>	Converteste <code>x</code> la <code>String</code> , unde <code>x</code> este <i>orice</i> valoare de tip (primitiv sau obiect).
<code>s =</code>	<code>String.format(f, x...)</code>	[Java >=5] Folosește formatul <code>f</code> pentru a converti un număr variabil de parametri, <code>x</code> la un șir.

- Lista nu este exhaustivă.



Prelucrarea șirurilor

- Un obiect **String** nu se poate schimba (eng. immutable), caracterele pe care le conține nu pot fi schimbate
- Clasa **StringBuffer** are metode prin care se pot edita obiectele sale șir
- Se poate totuși modifica valoarea unei variabile **String** prin asignare

```
String nume = "Ionescu";  
nume = "Ion " + nume;
```



Seturi de caractere

- *ASCII*: Set de caractere folosit de multe limbaje de programare, care conține 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 limbajul Java care include tot setul ASCII plus multe dintre caracterele folosite cu alfabetul latin
 - Exemple (atenție, ș, ț sunt cu diacritice nu cu virgule în exemplu):
`char c='\u0103'; // litera 'ă'`
`String s="\u00eencoto\u015fm\u0103ni\u0163i"; // încotoșmăniți`



0 porțiune 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", declarați constante simbolice (cu nume) și folosiți-le numele

```
public static final double CM_PER_INCH =  
    2.54;
```

```
public static final int HOURS_PER_DAY = 24;
```

- Previne schimbarea nedorită a valorii
- Ușurează modificarea valorii
- Convenția de nume pentru constante: toate literele majuscule, limitele de cuvinte marcate prin liniuța de subliniere



Comentariile

- Comentariu de o *linie*: începe cu simbolurile `//` și provoacă ignorarea a ceea ce urmează până la sfârșitul liniei
 - Folosit de cel care scrie codul sau de cel care îl modifică
- Comentariul *bloc* este la fel ca în C (perechea `/*`, `*/`)
 - Furnizează documentație utilizatorilor programului



Documentarea programului

- Java include programul `javadoc` care extrage automat documentația 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 și modelul de indentare

Computer Science



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

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



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 la început cazurile comune (acolo unde se poate)
- Folosiți **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!
}
```



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, int.**

Computer Science



- Java 7, în plus:
 - Tipuri *enumerated* (discutate în [Enum Types](#)),
 - Clasa [String](#) și câteva clase speciale care împachetează anumite tipuri primitive, [Character](#), [Byte](#), [Short](#) și [Integer](#)
 - Cf. <http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html>
- 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> )  
    <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 întâlnește o valoare stabilită, 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

■ Bucla 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)

```
// Cod rău
```

```
stmts_A
```

```
while (boolExp) {
```

```
    stmts_B
```

```
    stmts_A
```

```
}
```

```
// Cod bun
```

```
while (true) {
```

```
    stmts_A
```

```
    if (!boolExp)
```

```
        break;
```

```
    stmts_B
```

```
}
```



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>)  
    <instrucțiuni>
```

- 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 *preceadă* 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 *preceadă* 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)

Constructori

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"



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 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): 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	<pre>public void turnLeft() { ... }</pre>
Ieșire	Pentru ieșire este nevoie de un tip returnat
Intrare	0 sau mai mulți parametri
Utilizare	<pre>> cab.turnLeft();</pre>
# 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 instanțibilă

- O clasă este *instanțibilă* dacă putem crea instanțe ale clasei respective.
- Exemple: clasele "învelitoare: ale tipurilor primitive, **String**, **Scanner**,... sunt toate instanțiable, 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 creați 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

TECHNICAL UNIVERSITY

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



## 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 **suprascrie**, 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 instanțiabilă, 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: 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 pentru 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 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
  - 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) ...
```



## De studiat

---

- Deitel – capitolele 3, 4, 5
  - Eckel – capitolele 4, 5
  - Barnes – capitolele 1, 2
- (cf. slide 8 din Prelegerea 1 pentru numele complet al cărților)





# Rezumat

- Controlul execuției
  - Clase învelitoare
  - Operatori aritmetici
  - Clasa String
- Controlul execuției
  - if, switch, while, for, do – while
  - break/continue cu etichetă
  - for each
- Metode:
  - tipuri (accesoare, mutatoare)
  - intrare, ieșire
- Clase
  - modificali de acces
  - supraîncărcare
  - constructori
  - clase instanțiable
  - creare, supraîncărcarea constructorilor
  - referința `this`
  - Variabile la nivel de clasă
- Cuvântul cheie **static** la
  - metode
  - variabile