



# Programare orientată pe obiecte

---

1. Clase și obiecte
2. Tablouri

TECHNICAL UNIVERSITY  
OF CLUJ-NAPOCA  
Computer Science



# Metode: cum funcționează un apel

```
// Autor : Fred Swartz
import javax.swing.*;
public class KmToMiles {
    private static double convertKmToMi(double kilometri) {
        return kilometri * 0.621; // sunt 0.621 mile intr-un kilometru.
    }
    public static void main(String[] args) {
        //... variabile locale
        String kmStr; // String km înainte de conversia la double.
        double km; // Număr of kilometri.
        double mi; // Număr of mile.
        //... Intrare
        kmStr = JOptionPane.showInputDialog(null, "Introduceti kilometri.");
        km = Double.parseDouble(kmStr);
        //... Calcule
        mi = convertKmToMi(km) ;
        //... Output
        JOptionPane.showMessageDialog(null, km + " kilometri sunt " + mi + "
        mile.");
    }
}
```



# Metode: cum funcționează un apel

- Considerați atribuirea `mi = convertKmToMi(km);`
- Pașii pentru procesarea acestei instrucțiuni sunt:
  1. Evaluează argumentele de-la-stânga-la-dreapta
  2. Depune un nou cadru de stivă (stack frame) pe stiva de apeluri. Spațiu pentru parametri și variabilele locale (parametrul kilometri doar, aici). Starea salvată a metodei apelante (include adresa de retur)
  3. Inițializează parametri. La evaluarea argumentelor, acestea sunt asignate parametrilor locali din metoda apelată.
  4. Execută metoda.
  5. Revino din metodă. Memoria folosită pentru cadrul de stivă pentru metoda apelată este extrasă de pe stivă.



# Crearea obiectelor

- Java are trei mecanisme dedicate asigurării inițializării corespunzătoare a obiectelor:
  - *inițializatori de instanță* (numiți și blocuri de inițializare de instanță),
  - *inițializatori de variabile instanță*, și
  - *constructori*.
- Toate cele trei mecanisme rezultă în cod executat automat la crearea unui obiect.
- La alocarea memoriei pentru un nou obiect folosind operatorul `new` sau metoda `newInstance()` a clasei `Class`, JVM asigură executarea codului de inițializare înainte de folosirea zonei alocate.



# Crearea obiectelor

- La invocarea operatorului **new** :
  - Se alocă memorie (se rezervă spațiu pentru obiect). Variabilele instanță sunt inițializate la valorilor implicite
  - Se execută inițializarea explicită. Variabilele inițializate la declararea atributelor primesc valorile declarate.
  - Se execută un constructor. Valorile variabilelor pot fi schimbate de constructor
    - Se atribuie variabilei o referință la obiect.
- Exemple:

```
Automobil beetle = new Automobil("Volkswagen  
Beetle", Color.orange, 80, 160, 10);
```



# Valori inițiale implicite pentru câmpuri

Tip	Valoare
boolean	false
byte	(byte) 0
short	(short) 0
int	0
long	0L
char	\u0000
float	0.0f
double	0.0d
referință la obiect	null



# Inițializarea câmpurilor

- Atribuire simplă
  - Dacă putem da o valoare inițială unui câmp la declararea sa.
  - D.e.
    - `public static int capacity = 10; //initialize to 10`
    - `private boolean full = false; //initialize to false`
- Blocuri de inițializare statice
  - Bloc normal de cod între acolade, { } și precedat de cuvântul cheie `static`
  - D.e.
    - `static {`  
`// codul necesar inițializării se scrie aici`  
`}`
  - Folosit la inițializarea variabilelor la nivel de clasă
  - Blocurile de inițializare statice sunt executate în ordinea în care apar în sursă



# Inițializarea câmpurilor

- Alternativă la blocurile statice: metodă statică privată:
- D.e.
  - ```
class Oricare {  
    public static varType myVar =  
        initializeClassVariable();  
    private static varType initializeClassVariable() {  
        //codul de inițializare se pune aici  
    }  
}
```





# Inițializarea câmpurilor

- Inițializarea membrilor instanțelor
  - Asemănătoare blocurilor statice, dar nu static
  - Compilatorul java copiază blocurile de inițializare în fiecare constructor
  - D.e.

```
{  
    // codul pentru inițializare, aici  
}
```

Computer Science



# Inițializarea câmpurilor

## ■ Inițializarea membrilor instanțelor

- Se folosește o metodă **final** pentru inițializarea unei variabile instanță:

```
class Oricare {  
    private varType myVar = initializeInstanceVariable();  
    protected final varType initializeInstanceVariable()  
    { //initialization code goes here }  
}
```

- Folositoare mai ales dacă subclasele doresc să refolosească metoda de inițializare.
- Metoda este **final** deoarece apelul metodelor non final la inițializarea instanțelor pot cauza probleme



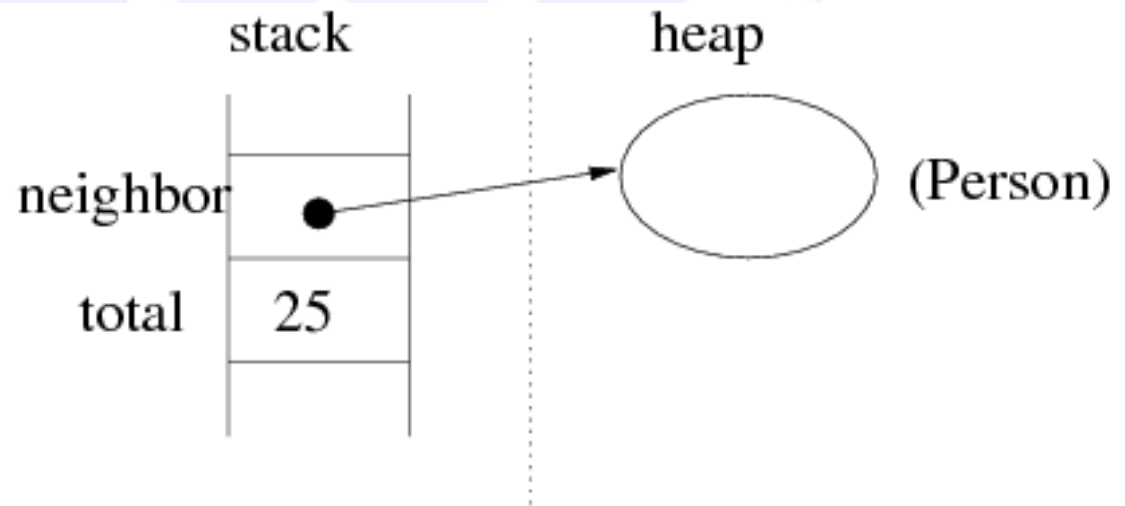
# Crearea obiectelor

- La apelul unui constructor:
  - Se alocă spațiu *pe heap* pentru obiect
  - *Fiecare obiect primește spațiu propriu (propria copie a variabilelor instanță)*
  - Starea obiectului este inițializată potrivit codului (definit de programator) clasei
- Observați că declararea unei variabile ca fiind de un tip de obiect produce o referință la obiect și nu obiectul în sine. Pentru a obține obiectul în sine folosiți **new** și un constructor pentru clasă



# Crearea obiectelor

- Puteți combina declararea și inițializarea  
`Person neighbor = new Person();`
- exact așa cum puteți face pentru tipurile primitive  
`int total = 25;`





# Asignarea obiectelor și alias-uri

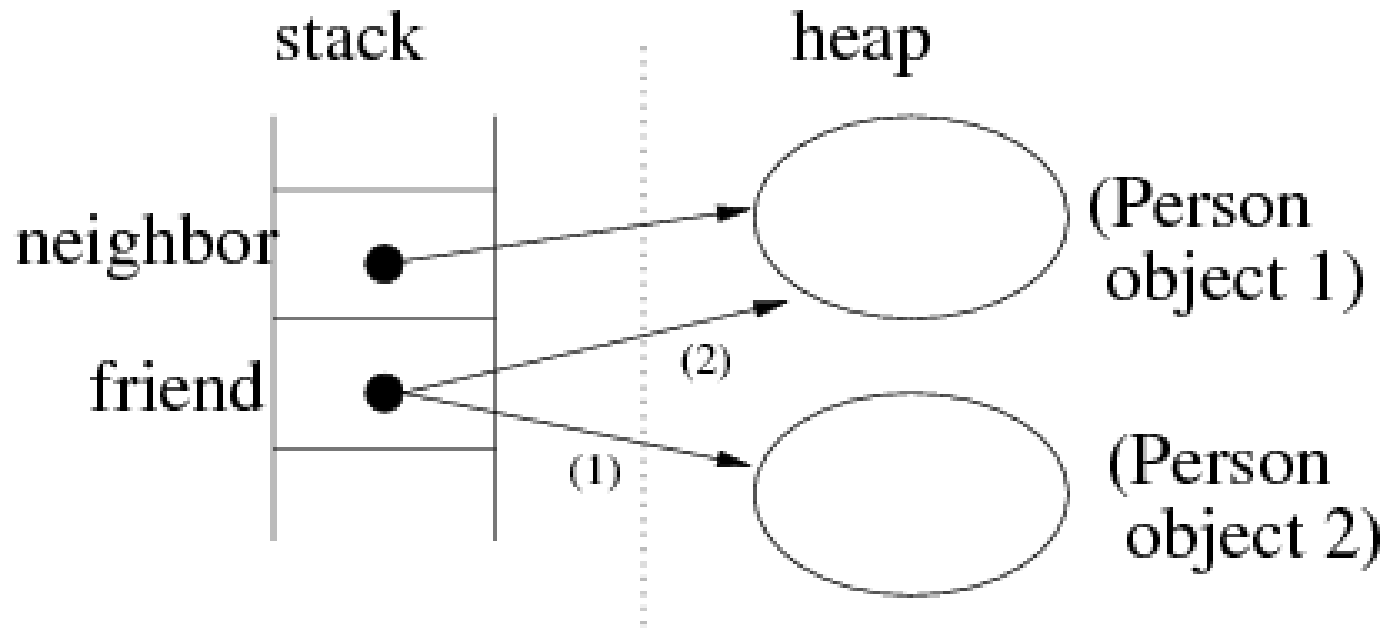
- Semnificația asignării este *diferită* pentru obiecte față de tipurile primitive

```
int num1 = 5;
int num2 = 12;
num2 = num1; // num2 conține acum 5
//-----
Person neighbor = new Person(); // creează object1
Person friend = new Person(); // creează object2
friend = neighbor;
```

- La sfârșit atât **friend** cât și **neighbor** se referă la object1 (ele sunt **alias** unul pentru celălalt) și nimic nu se mai referă la object2 (acesta este **inaccesibil**)



# Asignarea obiectelor și alias-uri



- Java va *colecta automat reziduul* (zona de memorie nefolosită) **object2**



# O clasă "Complex" foarte simplă

```
public class ComplexTrivial
{
    //Proprietăți
    private double real;
    private double img;
    // Constructor care
    // inițializează valorile
    public ComplexTrivial (double r, double i)
    { real = r; img = i; }
    // Definește o metodă pentru adunare
    public void aduna(ComplexTrivial cvalue) {
        real = real + cvalue.real;
        img = img + cvalue.img;
    }
    // Definește o metodă pentru scădere
    public void scade(ComplexTrivial cvalue){
        real = real - cvalue.real;
        img = img - cvalue.img;
    }
}
```

## ComplexTrivial

-real: double  
-img: double

<<constructor>>+ComplexTrivial(r: double, i: double)  
<<mutator>>+aduna(cvalue: ComplexTrivial)  
<<mutator>>+scade(cvalue: ComplexTrivial)

**Notă. Codul din prezentări nu respectă în totalitate stilul recomandat din lipsă de spațiu. Codul scris de Dvs. TREBUIE să respecte regulile de stil.**



# O clasă "comutator" simplă

```
// Un comutator on/off simplu (ne atașat de
// nimic).
class ComutatorSimplu {
    // Comută pe închis.
    public void inchideComutator()
    {
        System.out.println("Inchide
        comutatorul");
        seteazaInchis(true);
    }
    // Comută pe deschis.
    public void deschideComutator()
    {
        System.out.println(" Deschide
        comutatorul");
        seteazaInchis(false);
    }
    // Raportează dacă comutatorul este închis
    // sau nu.
    public boolean esteComutatorulInchis ()
    {
        return obtineStareInchis();
    }
}
```

```
// Returnează starea comutatorului
private boolean obtineStareInchis()
{    return inchis;    }
// Setează starea comutatorului.
private void seteazaInchis(boolean o)
{
    inchis = o;
}
// Dacă comutatorul este închis sau nu
// true înseamnă închis, fals înseamnă
// deschis.
private boolean inchis = false;
}
```

- Obs. **javadoc** NU va genera o documentație corespunzătoare pentru acest mod de comentare





# Un joc Tic Tac Toe

```
public class TicTacToe{
// Variabile instanță
/* tablou bidimensional de caractere pentru
tabla
*/
private char[][] tabla;

/** Constructor – crează o tablă în care
fiecare pătrat conține un caracter
subliniere '_' .
*/
public TicTacToe()
{
tabla = new char[3][3];
for (int rind = 0; rind < 3; rind ++ )
{
for (int col = 0; col < 3; col ++ )
{
tabla[rind][col] = '_';
} // sfârșit bucla interioară
} // sfârșit bucla exterioară
}
```

```
/** Pune caracterul c pe poziția [rind][col] a
* tablei de joc dacă rind, col și c sunt valide
* și pătratului de la [rind][col] nu i-a fost
* atribuită deja o valoare ( alta decât valoarea
* implicită, '_' ).
* @param rind rindul de pe tabla
* @param col coloana de pe tabla
* @param c caracter folosit la marcarea
* @return true dacă reușește, alfel false
*/
public boolean set(int rind, int col, char c)
{
if (rind >= 3 || rind < 0)
return false;
if (col >= 3 || col < 0)
return false;
if (tabla[rind][col] != '_')
return false;
if ( !(c == 'X' || c == 'O') )
return false;
// aserțiune: rind, col, c sunt valide
tabla[rind][col] = c;
return true;
}
```



# Un joc Tic Tac Toe (continuare)

```
/**
 * @return caracterul din poziția [rind][col] de pe tabla.
 * @param rind rindul de pe tabla
 * @param col coloana de pe tabla
 */
public char get(int rind, int col)
{
    return tabla[rind][col];
}
/** Tipărește starea tablei, d.e.
 *
 *  _ _ _
 *  _ X O
 *  O _ X
 */
public void print(){
    for (int rind = 0; rind < 3; rind ++){
        for (int col = 0; col < 3; col++){
            System.out.print(tabla[rind][col] + " ");
        } // sfârșit bucla interioară
        System.out.println();
    } // sfârșit bucla exterioară
}
}
```

## Exerciții:

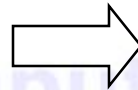
- completați jocul pentru al face jucabil (poate mai definiți clase?)
- modificați clasa astfel încât să permită dimensiuni mai mari ale tablei de joc



# Sugestii pentru proiectarea claselor

- Păstrați *private* datele întotdeauna
  - Schimbările în reprezentarea datelor nu vor afecta utilizatorii claselor; erorile sunt mai ușor de detectat
- *Inițializați* datele întotdeauna
  - Java nu va inițializa variabilele locale, dar va inițializa variabilele instanța ale obiectelor. Nu vă bazați pe valorile implicite, ci inițializați variabilele explicit
- Nu folosiți *prea multe tipuri* într-o clasă
  - Înlocuiți folosirile multiple *înrudite* ale tipurilor de bază cu alte clase. Spre exemplu:

```
private String strada;  
private String oras;  
private String stat;  
private String tara;  
private int codPostal;
```



```
class Adresa {  
    private String strada;  
    private String oras;  
    private String stat;  
    private String tara;  
    private int codPostal;  
    . . .  
}
```



# Sugestii pentru proiectarea claselor

---

- Nu toate câmpurile necesită accesorii și mutatori individuali
  - E.g. Angajat – obține și setează salariul, dar nu și data încadrării (nu se schimbă o dată creată)
- Folosiți o formă standard pentru definirea claselor, de exemplu
  - caracteristici publice (public)
  - caracteristici vizibile în pachet (package scope)
  - caracteristici private



# Sugestii pentru proiectarea claselor

- Folosiți o formă standard pentru definirea claselor și, pentru fiecare secțiune, scrieți în ordine
  - *constante*
  - *constructori*
  - *metode*
  - *metode statice*
  - variabile instanță
  - variabile statice
- De ce? Utilizatorii sunt mai interesați de interfața publică decât de datele private și mai mult de metode decât de date.



# Sugestii pentru proiectarea claselor

- Divizați clasele cu prea multe responsabilități, de exemplu

```
class PachetCarti { // nu e bine
    public void PachetCarti() { . . . }
    public void amesteca() { . . . }
    public void obtineValoareaMaxima() { . . . }
    public void obțineFelulMaxim() { . . . }
    public void rangulMaxim() { . . . }
    public void deseneaza() { . . . }

    private int[] valoare;
    private int[] fel;
    private int carti;
}
// creați clasa Carte!
```



# Sugestii pentru proiectarea claselor

- Faceți numele claselor și metodelor să reflecte responsabilitățile acestora
- O convenție bună este:
  - Numele clasei: substantiv (d.e. Comanda) sau substantiv+adjectiv (e.g. ComandaUrgenta)
  - Numele metodelor: verbe; accesorii încep cu "get"; mutatorii încep cu "set"



# Feluri de clase în Java

## Clase

De nivel maxim (Top Level)

Membre

Imbricate  
(Nested)

Interioare

locale

anonime

Tipuri membre  
nastate

Trebuie să  
aibă o clasă  
care le conține

Trebuie să  
aibă o clasă  
care le declară

Pot avea un  
executabil  
care le  
cuprinde





# Feluri de clase în Java

- O *clasă de nivel maxim* (top level class) nu apare în interiorul altei clase sau interfețe
- Dacă un tip nu este de nivel maxim atunci este *imbricat* (nested)
  - Un tip poate fi *membru* al altui tip; un tip membru este inclus direct într-o altă declarație de tip
    - Unii membrii sunt *clase interioare* (inner classes) și cuprind:
      - *Clase locale*: clase cu nume declarate înăuntrul unui bloc (corpul unui constructor sau al unei metode)
      - *Clase anonime*: clasă nenumite ale căror instanțe sunt create în expresii sau în instrucțiuni.



# Clase interioare (interne)

---

- *Clasă interioară*: clasă definită înăuntrul alteia
  - Permite *gruparea claselor* care țin logic una de alta pentru a *controla vizibilitatea* uneia în cealaltă
  - Clasele interne sunt diferite de compoziție
- Crearea unei instanțe de clasă internă de oriunde – excepție fac metodele **ne-stactice** ale clasei exterioare: se specifică tipul obiectului ca *NumeleClaseiExterioare.NumeleClaseiInterne*



# Exemplu de clasă internă

- Tipic, o clasă externă va avea o metodă care returnează o referință la clasa internă

```
public class Parcel {  
    class Contents {  
        private int val = 10;  
        public int value() { return val; }  
    }  
    class Destination {  
        private String label;  
        Destination(String dst) {  
            label = dst;  
        }  
        String readLabel() { return label; }  
    }  
    public Destination to(String s) {  
        return new Destination(s);  
    }  
}
```

```
public Contents cont() {  
    return new Contents();  
}  
public void ship(String dest) {  
    Contents c = cont();  
    Destination d = to(dest);  
    System.out.println(d.readLabel());  
}  
public static void main(String[] args) {  
    Parcel p = new Parcel();  
    p.ship("Romania");  
    Parcel q = new Parcel();  
    // Defining references to inner classes:  
    Parcel.Contents c = q.cont();  
    Parcel.Destination d = q.to("China");  
}
```



# Clase interne

- De patru feluri:

- Clase membru statice
- Clase membru
- Clase locale
- Clase anonime

- Clasă *membru statică* este un membru static al unei clase

- Are acces la toate metodele *statice* ale clasei exterioare

```
class MyOuter {  
    public static class MyInner {  
        //...  
        public void function1() {}  
        //more functions  
    }  
}
```



# Clase interne

- *Clasă membru* : definită și ea ca membru al clasei.
  - Este *specifică instanței* și
  - Are *acces* la *toate* metodele și membrii, chiar și la referința *this* a clasei exterioare.

```
class MyOuter {  
    private float variable = 0;  
    public void doSomething() { //do your stuff }  
    private class MyInner {  
        public void doSomething() { //do your stuff }  
        public void function() {  
            //To call a function with the same name from the enclosing class  
            MyOuter.this.doSomething();  
        }  
    }  
}
```



# Clase interne

- *Clasele locale*: declarate într-un bloc de cod; vizibile în acel bloc, ca orice altă metodă/variabilă

```
interface MyInterface {  
    public String getInfo();  
}  
class MyOuter {  
    MyInterface current_object;  
    public void setInterface(String info) {  
        class MyInner implements MyInterface {  
            private String info;  
            public MyInner(String inf) {info=inf;}  
            public String getInfo() {return info;}  
        }  
        current_object = new MyInner(info);  
    }  
}
```



# Clase interne

---

- Clasă *anonimă*: clasă locală fără nume
- Clasele interne sunt utile la implementarea *callback*.



# Elemente de bază despre tablouri

---

- Tablouri de **tipuri de date primitive**
- Tablouri de **Obiecte**





# Elemente de bază despre tablouri

- În Java, un *tablou* este o colecție indexată de date de același tip.
- Tablourile sunt utile la sortări și la manipularea unei colecții de valori.
- În Java, un tablou este un tip de dată referință.
- Se folosește operatorul **new** pentru a aloca memorie pentru stocarea valorilor într-un tablou.  

```
double[] precipitatii;  
precipitatii = new double [12];  
//creează un tablou de mărime 12.
```
- Folosim o *expresie indexată* pentru a ne referi la elemente individuale din colecție.
- Tablourile folosesc indexarea de la zero.



# Elemente de bază despre tablouri

- Un tablou are o *constantă publică* `length` care conține dimensiunea tabloului.
- Nu confundați *valoarea* `length` a unui tablou cu *metoda* `length()` a unui obiect `String`.
- `length()` este metodă pentru un obiect `String`, deci folosim sintaxa pentru metodă.

```
String str = "acesta este un sir";  
int size = str.length();
```

- Pe de altă parte, un tablou este un tip de dată referință, nu un obiect. De aceea nu folosim apel de metodă.

```
int size = precipitatie.length;
```



# Elemente de bază despre tablouri

---

- Folosirea constantelor pentru declararea dimensiunilor tablourilor nu duce întotdeauna la folosirea eficientă a spațiului.
- Declararea cu dimensiuni fixe poate pune două probleme:
  - Capacitatea poate fi insuficientă pentru sarcina de îndeplinit
  - Spațiu irosit.
- Java, nu este limitat la declararea cu dimensiune fixă.
- După creare însă, un tablou este o structură de lungime fixă.
- Indiferent de tipul de tablou cu care se lucrează, variabila tablou este o referință la un obiect creat pe heap.
- Accesul la datele din tablou se fac prin acest obiect tablou.
- Obiectul tablou are o variabilă *identificator* unică.



# Elemente de bază despre tablouri

- Codul următor cere utilizatorului dimensiunea unui tablou și declară un tablou de dimensiunea cerută:

```
int size;  
int[] number;  
size= Integer.parseInt(JOptionPane.showInputDialog(null,  
    "Marimea tabloului:"));  
number = new int[size];
```

- Orice expresie aritmetică validă este permisă la specificarea dimensiunii unui tablou:

```
size = Integer.parseInt(  
    JOptionPane.showInputDialog(null, ""));  
number = new int[size*size + 2* size + 5];
```

- Tablourile nu sunt limitate la tipurile de date primitive



# Tablourile sunt obiecte

|                                                     |                                                                                                                                                                                              |
|-----------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>int[] data;</pre>                              | <p><i>data este o variabilă referință al cărei tip este <code>int[]</code>, însemnând "tablou de int". În acest moment valoarea sa este null.</i></p>                                        |
| <pre>data = new int[5];</pre>                       | <p><i>Operatorul <code>new</code> face să se aloce pe heap o zonă de memorie destul de mare pentru 5 int. Aici, lui <code>data</code> i se asignează o referință la adresa din heap.</i></p> |
| <pre>data[0] = 6;<br/>data[2] = 12;</pre>           | <p><i>Inițial, toți cei cinci int sunt 0. Aici, la doi dintre ei li se atribuie alte valori.</i></p>                                                                                         |
| <pre>int[] info = {6, 10, 12, 0, 0};</pre>          |                                                                                                                                                                                              |
| <pre>int[] info = new int[]{6, 10, 12, 0, 0};</pre> |                                                                                                                                                                                              |



# Excepții de depășire a limitelor tablourilor

```
public class ArrayTool{
    public int sum(int[] data){
        int sum = 0;
        for (int i = 0; i < data.length; i++){
            sum += data[i];
        }
        return sum;
    }
    public int sum2(int[] data){
        int sum = 0;
        for (int i = 0; i <= data.length; i++){
            sum += data[i];
        }
        return sum;
    }
}
```

**Folosirea acestei comparații produce aruncarea unei excepții `ArrayIndexOutOfBoundsException`**



# Tablouri de tipuri primitive

```
int[] data;
```

data null

```
data = new int[3];
```

data 500 → 0 0 0

```
data[0] = 5;
```

```
data[1] = 10;
```

data 500 → 5 10 0



# Tablouri de alte tipuri primitive

```
double[] temps;  
temps = new double[24];  
temps[0] = 18.5;  
temps[1] = 24.2;
```

```
boolean[] raspunsuri = new boolean[6];  
.  
.  
.  
if (raspunsuri[0])  
    faCeva();
```

```
char[] tampon = new char[500];  
deschide un fișier pentru citire  
while (mai sunt caractere în fișier & tampon nu  
este plin)  
    tampon[i++] = caracter din fișier
```





# Tablouri bidimensionale

- Tablourile pot avea 2, 3, sau mai multe dimensiuni
- La declararea unei variabile pentru un astfel de tablou, folosiți câte o pereche de paranteze pătrate pentru fiecare dimensiune
- Pentru tablourile bidimensionale, elementele sunt indexate [rind][coloana]
- Exemplu:

```
char[][] tabla;  
tabla = new char[3][3];  
tabla[1][1] = 'X';  
tabla[0][0] = 'O';  
tabla[0][1] = 'X';
```



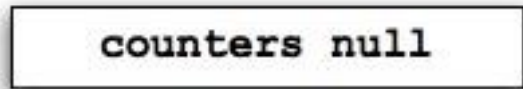
# O clasă contor

```
public class Counter
{
private int numar;
/**
 * Constructor. Initializeaza
 * contorul la zero.
 */
public Counter()
{
    numar = 0;
}
/**
 * @return valoarea curenta a
 * contorului
 */
public int obtineNumar()
{
    return count;
}
/**
 * Incrementeaza contorul
 * cu unu
 */
public void increment()
{
    numar++;
}
/**
 * Reseteaza contorul
 * la zero
 */
public void reset()
{
    numar = 0;
}
}
```



# Tablouri de obiecte

```
Counter[] counters;
```



STACK

HEAP

```
counters = new Counter[3];
```



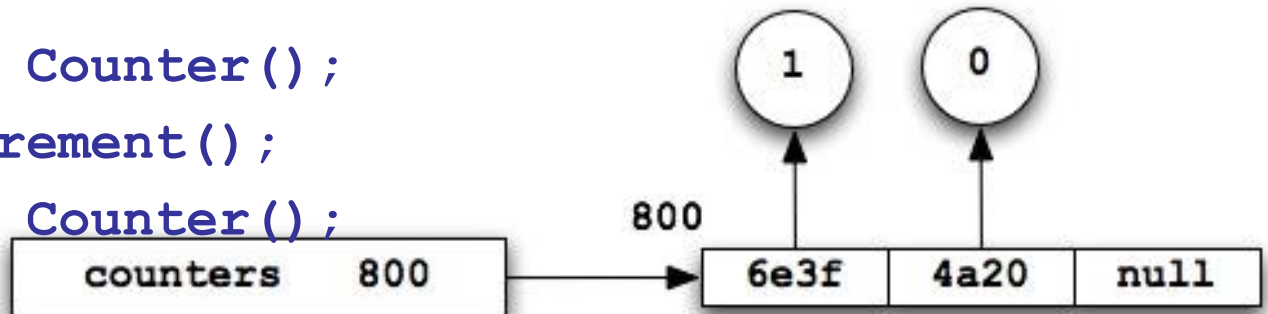
STACK

HEAP

```
counters[0]=new Counter();
```

```
counters[0].increment();
```

```
counters[1]=new Counter();
```



STACK

HEAP



# “Traversarea” tablourilor de obiecte

- Așa cum putem folosi cicluri pentru a traversa tablouri de tipuri primitive, putem face și cu tablourile de obiecte
- Exercițiu. Scrieți o metodă care:
  - ia un singur argument: un tablou de contoare
  - returnează suma numerelor conținute în contoare
  - mai întâi, presupuneți că fiecare element de tablou se referă la un contor valid
  - Apoi rescrieți metoda astfel ca să poată prelucra tablouri în care unele /toate elementele sunt **null**



# Capcană: un tablou de caractere nu este un String

- Totuși, un tablou de caractere nu este un obiect de clasa **String**

```
char[] a = {'A', 'B', 'C'};  
String s = a; //Illegal!
```

- Un tablou de caractere poate fi convertit la un obiect de clasa **String**
- Un tablou de caractere este conceptual o listă de caractere și de aceea conceptual ca un șir
- Clasa **String** are un constructor cu un singur parametru de tip **char[]**

```
String s = new String(a);
```

- Obiectul **s** va avea aceeași secvență de caractere ca întregul tablou **a** ("ABC"), dar este o copie *independentă*



# Capcană: un tablou de caractere nu este un String

- Un alt constructor al **String** folosește o subgamă a tabloului de caractere

```
String s2 = new String(a, 0, 2);
```

- Fiind dat **a** ca mai înainte, noul obiect șir este **"AB"**

- Un tablou de caractere are ceva în comun cu obiectele **String**

- D.e., un tablou de caractere poate fi tipărit folosind **println**

```
System.out.println(a);
```

- Dat fiind **a** ca mai înainte, se va tipări

**ABC**



# Prescurtări la inițializarea tablourilor

---

## Tablouri de tipuri primitive:

- `int[] info1 = { 2000, 100, 40, 60};`
- `int[] info2 = new int[]{ 2000, 100, 40, 60};`
- `char[] choices1 = { 'p', 's', 'q'};`
- `char[] choices2 = new char[]{ 'p', 's', 'q'};`
- `double[] temps1 = {75.6, 99.4, 86.7};`
- `double[] temps2 = new double[] {75.6, 99.4, 86.7};`

Computer Science



# Prescurtări la inițializarea tablourilor

---

Tablouri de obiecte:

- `Person[] people = {new Person("jo"), new Person("flo")};`
- `Person[] people = new Person[] {new Person("jo"),  
new Person("flo")};`
- `Point p1 = new Point(0,0);`
- `Point[] points1 = {p1, new Point(0, 10)};`
- `Point[] points2 = new Point[] {p1, new Point(0, 10)};`

***Notă: Construcția sintactică “new type[ ]” poate fi folosită la o asignare care nu este și o declarație de variabilă***





## Transmiterea tablourilor ca parametri

---

- Atunci când nu mai există nici o referință spre un obiect, sistemul va șterge obiectul și va disponibiliza memoria ocupată de acesta.
- Ștergerea unui obiect se numește *dealocarea* memoriei.
- Procesul de dealocare a memoriei se numește *colectarea reziduurilor*. Aceasta se face automat în Java.
- La transmiterea ca *parametru* a unui *tablou* spre o metodă, se transmite doar o *referință* spre tabloul respectiv
- Nu se creează o copie a tabloului în metodă.



# Exemplu: "baza de date" PersonDB

```
public class Person{
    private String nume;
    private int virsta;
    public Person(String nume, int virsta){
        this.nume = nume;
        this.virsta = virsta;
    }
    public Person(String nume){
        this(nume, 5);
    }
    public int obtineVirsta() { return virsta; }
    public String obtineNume() { return nume; }
}
```



# Tablouri de obiecte

```
Person[] people;
```

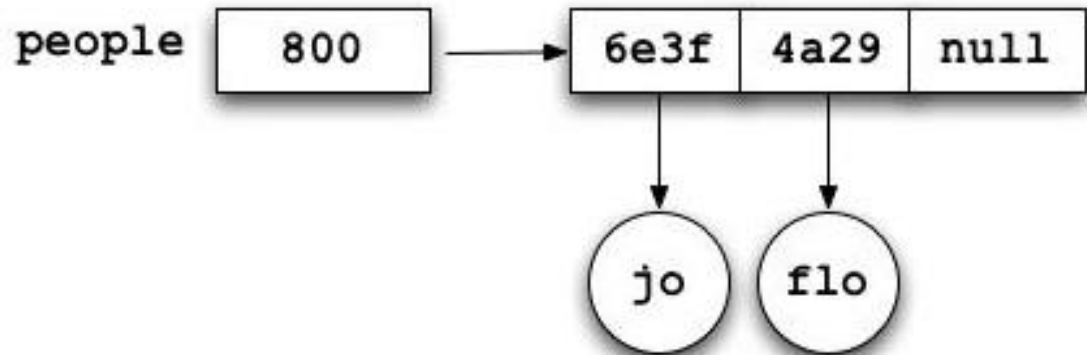


```
people = new Person[3];
```



```
people[0]=new Person("jo");
```

```
people[1]=new  
Person("flo");
```





# Exemplu: "baza de date" PersonDB

```
public class PersonDB{
    private Person[] people;
    public PersonDB(){
        people = new Person []{new Person("geta",25) ,
                                new Person("lili",18) ,
                                new Person("veta", 19)};
    }
    /** Calculează și returnează media de vârstă */
    public double getVirstaMedie(){
        return 0; // DIY
    }
    /** Returnează true dacă numele este în bază, altfel false */
    public boolean esteInBaza(String numeDeCautat){
        return false; // DIY
    }
}
```



# Copierea tablourilor

- Clasa **System** are o metoda numită **arraycopy**
  - Folosită la copierea eficientă a datelor între tablouri

```
public static void arraycopy(Object src,  
    int srcPos, Object dest, int destPos, int  
    length)
```



# De studiat

---

- Deitel: capitolele 6, 7
- Barnes: secțiunea 4.16, capitolul 6
- Eckel: capitolele 6, 7, 11, 17



# Rezumat

- Metode: cum funcționează apelul
- Crearea obiectelor
- Clase exemplu + ceva de făcut
- Sugestii pentru proiectarea claselor
- Feluri de clase
  - Clase interioare
- Tablouri:
  - Fundamente
  - Tablouri de tipuri primitive
  - Tablouri de obiecte
  - Traversarea
  - Inițializarea
  - Transmiterea ca parametri