Programare orientată pe obiecte

1. Clase şi obiecte
2. Tablouri

Metode: cum funcţionează 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 funcţionează un apel

- Consideraţi atribuirea **`mi = convertKmToMi(km);`**
- Paşii pentru procesarea acestei instrucţiuni sunt:
  1. Evaluează argumentele de-la-stânga-la-dreapta
  2. Depune un nou cadru de stivă (stack frame) pe stiva de apeluri. Spaţiu pentru parametri şi variabilele locale (parametrul kilometri doar, aici). Starea salvată a metodei apelante (include adresa de retur)
  3. Iniţializează 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 apelata este extrasă de pe stivă.

Crearea obiectelor

- Java are trei mecanisme dedicate asigurării iniţializării corespunzătoare a obiectelor:
  - *iniţializatori de instanţă* (numiţi şi blocuri de iniţializare de instanţă),
  - *iniţializatori de variabile instanţă*, şi
  - *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 iniţializare înainte de folosirea zonei alocate.

Crearea obiectelor

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

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


Valori iniţiale 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
referinţă la obiect	null

Iniţializarea câmpurilor

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

Iniţializarea câmpurilor

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

Iniţializarea câmpurilor

- Iniţializarea membrilor instanţelor
  - Asemănătoare blocurilor statice, dar nu static
  - Compilatorul java copiază blocurile de iniţializare în fiecare constructor
  - D.e.

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


Iniţializarea câmpurilor

- Iniţializarea membrilor instanţelor
  - Se foloseşte o metodă **final** pentru iniţializarea unei variabile instanţă:

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

  - Folositoare mai ales dacă subclasele doresc să refolosească metoda de iniţializare.
  - Metoda este **final** deoarece apelul metodelor non final lainiţializarea instanţelor pot cauza probleme

Crearea obiectelor

- La apelul unui constructor:
  - Se alocă spaţiu *pe heap* pentru obiect
  - *Fiecare obiect primeşte spaţiu propriu (propria copie a variabilelor instanţă)*
  - Starea obiectului este iniţializată potrivit codului (definit de programator) clasei
- Observaţi că declararea unei variabile ca fiind de un tip de obiect produce o referinţă la obiect şi nu obiectul în sine. Pentru a obţine obiectul în sine folosiţi **new** şi un constructor pentru clasă

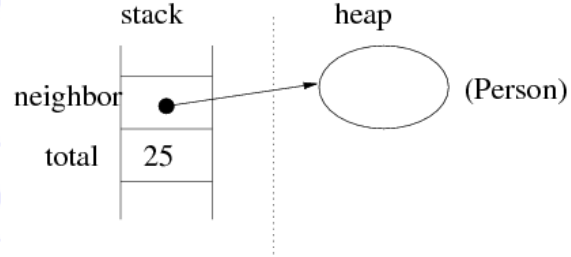
Crearea obiectelor

- Puteţi combina declararea şi iniţializarea

```
Person neighbor = new Person();
```

- exact aşa cum puteţi face pentru tipurile primitive

```
int total = 25;
```



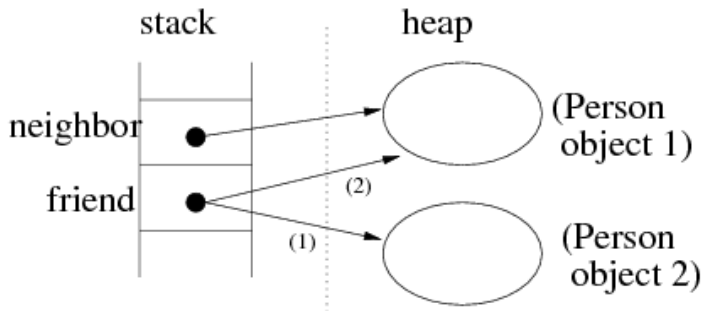
Asignarea obiectelor şi alias-uri

- Semnificaţia asignării este *diferită* pentru obiecte faţă de tipurile primitive

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

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

Asignarea obiectelor şi alias-uri



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

O clasă "Complex" foarte simplă

```
public class ComplexTrivial
{
  //Proprietăţi
  private double real;
  private double img;
  // Constructor care
  // iniţializează valorile
  public ComplexTrivial (double r, double i)
  { real = r; img = i; }
  // Defineşte o metodă pentru adunare
  public void aduna(ComplexTrivial cvalue) {
    real = real + cvalue.real;
    img  = img  + cvalue.img;
  }
   // Defineşte 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 spaţiu. Codul scris de Dvs. TREBUIE să respecte regulile de stil.

O clasă "comutator" simplă

```
// Un comutator on/off simplu (ne ataşat 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;   }
  // Seteaza starea comutatorului.
  private void setează Inchis(boolean o)
  {
    inchis = o;
  }
  // Daca comutatorul este inchis sau nu
  // true inseamna inchis, fals inseamnă
  // deschis.
  private boolean inchis = false;
}
```

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

Un joc Tic Tac Toe

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

 /** Constructor – crează o tablă în care
    fiecare pătrat conţine 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] = '_';
     } // sfârşit bucla interioară
   } // sfârşit bucla exterioară
 }
```

```
 /** Pune caracterul  c pe poziţia [rind][col]  a
  * tablei de joc dacă rind, col şi c  sunt valide
  * şi 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 marcare
  * @return  true dacă reuşeşte, 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;
   // asserţiune: rind, col, c sunt valide
   tabla[rind][col] = c;
   return true;
 }
```


Un joc Tic Tac Toe (continuare)

```
 /**
  * @return caracterul din poziţia [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];
 }
 /** Tipăreşte 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] + " ");
     } // sfârşit bucla interioară
     System.out.println();
   } // sfârşit bucla exterioară
 }
}
```

Exerciţii:

- completaţi jocul pentru al face jucabil (poate mai definiţi clase?)
- modificaţi clasa astfel încât să permită dimensiuni mai mari ale tablei de joc

Exemplu de program: Jocul Hammurabi

- Un joc care simulează funcţionarea unei societăţi bazate pe agricultură, primitive.
- **Scop.** Utilizatorul, care tocmai a devenit conducătorul unui regat, vrea să-şi rezerve locul în istorie prin faptul că are cea mai mare populaţie de ţărani. Simularea ţine timp de cinci ani sau până când toţi au murit de foame.
- **Grânele** sunt resursa de bază. În fiecare an, conducătorul este întrebat cum să folosească stocul de grâne.
  - Câte chintale să fie folosite pentru a hrăni populaţia.
  - Câte chintale să se folosească ca sămânţă pentru recolta anului viitor.
  - Grâul care mai rămâne este păstrat ca rezervă în caz că anul viitor aduce o recoltă proastă.

Exemplu de program: Jocul Hammurabi

- **Condiţii iniţiale.**
  - Programul trebuie să creeze un Regat cu următoarele valori: o zonă de 600 ha, o populaţie de 100 şi cu 1400 chintale de grâu din recolta precedentă.
- **Reguli pentru alimente şi populaţie**
  - Fiecare persoană necesită un minimum de 6 chintale de grâu pe an pentru a supravieţui.
  - **Înfometarea**. Dacă conducătorul nu alocă destulă mâncare pentru toţi, unii vor muri de foame. Populaţia se va reduce cu numărul celor decedaţi.
  - **Apar imigranţi dacă e belşug**. Dacă oamenii primesc mai mult de 6 chintale pe persoană, vor fi atraşi imigranţi din regatele învecinate ceea ce cauzează o creştere a populaţiei.
  - **Formula**. Această idee simplificată asupra mărimii populaţiei se poate calcula simplu prin împărţirea cantităţii totale de alimente la cantitatea necesara pe persoană.

```
populatiaMea = griuConsum/GRIU_NECESAR_UNEI_PERSOANE
```

Exemplu de program: Jocul Hammurabi

- **Agricultura**
  - **Sămânţa pentru plantare**. Nu se poate folosi tot grâul pentru consum. O parte trebuie folosită pentru însămânţarea recoltei anului viitor. E nevoie 0.9 chintale pentru fiecare hectar. Pentru tot e nevoie de 0.9 * suprafaţa.
  - **Recolta**. Exista variaţii ale vremii de la un an la altul. Recolta variază între 0.9 şi 2.4 chintale pe hectar cultivat. Acest număr se generează în fiecare an.
- **Principiu de proiectare**: ***Separarea responsabilităţilor*** – *scopul fiecărei clase este diferit.*
  - Unul dintre principalele scopuri ale separării unui program în clase diferite este să se grupeze datele şi metodele care ţin una de alta şi să nu se amestece diferitele responsabilităţi ale claselor.


```
// Author : Fred Swartz
import javax.swing.*;
public class Hammurabi {
    public static void main(String[] args) {
        //... Iniţializare
        Regat samaria = new Regat();  // Create a new Regat
        //... Execută simularea timp de 5 ani sau până când mor toţi de foame.
        while (samaria.ceAn() <= 5) {
            //... Afişează starea regatului la începutul fiecărui an.
            JOptionPane.showMessageDialog(null, samaria.toString());
            //DEFACUT: Întreabă conducătorul cât să se dea oamenilor.
            int griuConsum = 0;  // Substitut temporar pentru intrebare.
            //.. Întreabă conducătorul cât să se reţină pentru sămânţă.
            String plantStr = JOptionPane.showInputDialog(null,
               "Gloriosule Conductor, câte chintale din griul ramas "
               + (samaria.citGriu()-griuConsum) + " sa fie plantate?");
            int saminta = Integer.parseInt(plantStr);
            //... DEFACUT: Verifică dacă este destul grâu pentru această cerere
            // si dacă este necesar intreabă din nou
            //... Actualizează grâul şi populatia acestui Regat.
            samaria.simuleazaUnAn(griuConsum, saminta);
        }
        //... Arată starea finală.
        JOptionPane.showMessageDialog(null, "În final " + samaria.toString());
    }
}
```

Exemplu de program: Jocul Hammurabi. Clasa Regat

```
// Author : Fred Swartz
class Regat {
   private final static float MIN_GRIU_PENTRU_SUPRAVIETUIRE = (float)0.8;
   private final static float MAX_TEREN_CULTIVABIL_PER_PERSOANA = (float)6.0;
   private final static float SAMINTA_NECESARA_PER_HA = (float)0.9;
   private float griulMeu = 1400; // Chintale de grâu  în stoc.
   private int suprafataMea  = 600; // Suprafaţa Regatului în hectare.  Obs.
                        // suprafataMea nu este folosita deocamdată
                        // dar va fi dacă adăugaţi verificarea cantităţii totale
                        // de pământ care poate fi cultivat.
   private int anulMeu  = 0;    // Ani de la fondarea Regatului.
   private float recoltaMea = 0;  // Ultima recoltă în chintale.
   public float citGriu() {  return griulMeu;   }
   public int ceAn() {  return anulMeu;    }
   public String toString()
   {
      // DEFACUT: Nu uitaţi să adăugaţi şi aici populaţia
      return "Starea Regatului în anul " + anulMeu  + ", ultima recolta = " +
            recoltaMea + ", total grine = " + griulMeu;
   }
```


Exemplu de program: Jocul Hammurabi. Clasa Regat (continuare)

```
   public void simuleazaUnAn(float griuConsum, float saminta)
   {
        //DEFACUT: Trebuie calculata populaţia pe baza grânelor.
        //... Redu stocul de grâu cu cantitatea folosită ca aliment şi ca sămânţă
        griulMeu = griulMeu - griuConsum - saminta;
        //... Calculaţi noua recoltă
        //    1. Câte hectare pot fi plantate cu saminta.
        //    2. Recolta la hectar este variabilă (0.9-2.4)
        //    3. Recolta este rezultat pe hectar * suprafata plantată.
        float hectarePlantate = saminta / SAMINTA_NECESARA_PER_HA;
        // DEFACUT: Verificaţi că sunt destui oameni şi este destul pământ
        //      pentru a cultiva acel număr de hectare.
        float recoltaLaHectar = (float)(0.9 + 1.5 * Math.random());
        recoltaMea       = recoltaLaHectar * hectarePlantate;
        //... Calculează noua cantitate de grâu din stoc.
        griulMeu += recoltaMea;  //  Noua cantitate de grâu din stoc
        anulMeu++;           // A mai trecut un an.
   }
    // DEFACUT: de verificat potrivirea tipurilor de variabile (float, int, etc)
    // cu modul în care sunt folosite
}
```

Exemplu de program: Jocul Hamurabi. Temă în continuare

- Extensii suplimentare:
  - **Vânzarea şi cumpărarea de teren**. Se poate cumpăra pământ de la regatele învecinate (şi li se poate vinde) pentru 9 chintale pe hectar. Planul anual ar trebui să cuprindă şi cât pământ să se cumpere/vândă pe lângA alocarea mâncării şi a seminţei. Permiterea adăugării de pământ este singura cale ca populaţia să crească cu adevărat.
  - **Limitări la cultivare**. Cantitatea cultivabilă de un ţăran trebuie limitată la 6 hectare. dacă scade populaţia, s-ar putea să nu se mai cultive toată suprafaţa.
  - **Revolta**. Dacă mai mult de 45% din populaţie moare de foame, apare revolta şi conducător este alungat – jocul se termină
  - **Soldaţi.** Pot juca un rol important, dar Dvs. decideţi ce anume (d.e. înăbuşe revolte, ocupă teren etc.). Dar poate trebuie alimentaţi mai bine decât ţăranii.
  - **Fluctuaţii aleatoare suplimentare**
  - **Şobolani**. În fiecare an, şobolanii mănâncă între 0-10% din grâul pe care îl aveţi.
  - **Ciuma**. Există o şansă de 15% pentru izbucnirea unei ciume în fiecare an. Dacă se întâmplă, moare o jumătate din populaţie.

Sugestii pentru proiectarea claselor

- Păstraţi *private* datele întotdeauna
  - Schimbările în reprezentarea datelor nu vor afecta utilizatorii claselor; erorile sunt mai uşor de detectat
- *Iniţializaţi* datele întotdeauna
  - Java nu va iniţializa variabilele locale, dar va iniţializa variabilele instanţa ale obiectelor. Nu vă bazaţi pe valorile implicite, ci iniţializaţi variabilele explicit
- Nu folosiţi *prea multe tipuri* într-o clasă
  - Înlocuiţi 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 accesori şi mutatori individuali
  - E.g. Angajat – obţine şi setează salariul, dar nu şi data încadrării (nu se schimbă o dată creată)
- Folosiţi o formă standard pentru definirea claselor, de exemplu
  - caracteristici publice (public)
  - caracteristici vizibile în pachet (package scope)
  - caracteristici private

Sugestii pentru proiectarea claselor

- Folosiţi o formă standard pentru definirea claselor şi, pentru fiecare secţiune, scrieţi în ordine
  - constante
  - constructori
  - metode
  - metode statice
  - variabile instanţă
  - variabile statice
- De ce? Utilizatorii sunt mai interesaţi de interfaţa publică decât de datele private şi mai mult de metode decât de date.

Sugestii pentru proiectarea claselor

- Divizaţi clasele cu prea multe responsabilităţi, de exemplu

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

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


Sugestii pentru proiectarea claselor

- Faceţi numele claselor şi metodelor să reflecte responsabilităţile acestora
- O convenţie 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"

Elemente de bază despre tablouri

- În Java, un *tablou* este o colecţie indexată de date de acelaşi tip.
- Tablourile sunt utile la sortări şi la manipularea unei colecţii de valori.
- În Java, un tablou este un tip de dată referinţă.
- Se foloseşte operatorul **new** pentru a aloca memorie pentru stocarea valorilor într-un tablou.

```
precipitatii = new double [12];
 //creează un tablou de mărime 12.
```

- Folosim o *expresie indexată* pentru a ne referi la elemente individuale din colecţie.
- Tablourile folosesc indexarea de la zero.

Elemente de bază despre tablouri

- Un tablou are o *constantă publică* **length** care conţine dimensiunea tabloului.
- Nu confundaţi *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 referinţă, nu un obiect. De aceea nu folosim apel de metodă.

```
int size = precipitatii.length;
```

Elemente de bază despre tablouri

- Folosirea constantelor pentru declararea dimensiunilor tablourilor nu duce întotdeauna la folosirea eficientă a spaţiului.
- Declararea cu dimensiuni fixe poate pune două probleme:
  - Insuficientă capacitate pentru sarcina de îndeplinit
  - Spaţiu 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 referinţă 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 şi 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

`int[] data;`	*data este o variabilă referinţă al cărei tip este int[], însemnând "tablou de int". În acest moment valoarea sa este null.*
`data = new int[5];`	*Operatorul new face să se aloce pe heap o zonă de memorie destul de mare pentru 5 int. Aici, lui data i se asignează o referinţă la adresa din heap.*
`data[0] = 6;` `data[2] = 12;`	*Iniţial, toţi cei cinci int sunt 0. Aici, la doi dintre ei li se atribuie alte valori.*
`int[] info = {6, 10, 12, 0, 0};`	
`int[] info = new int[]{6, 10, 12, 0, 0};`	

Excepţii de depăşire 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 comparaţii produce aruncarea unei excepţii ArrayIndexOutOfBoundsException

Tablouri de tipuri primitive

```
int[] data;
```


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


```
data[0] = 5;
data[1] = 10;
```

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 fişier pentru citire
while (mai sunt caractere în fişier & tampon nu este plin)
    tampon[i++] = caracter din fişier
```


Tablouri bidimensionale

- Tablourile pot avea 2, 3, sau mai multe dimensiuni
- La declararea unei variabile pentru un astfel de tablou, folosiţi 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 Contor
{
private int numar;
/**
*  Constructor. Initializeaza
   contorul la zero.
*/
public Contor()
{
    numar = 0;
}
/**
* @return valoarea curenta a
* contorului
*/
public int obtineNumar()
{
    return count;
}
```

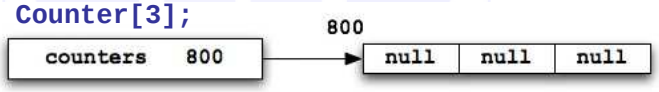
```
/**
* Incrementeaza contorul
* curenta cu unu
*/
public void increment()
{
  numar++;
}
/**
* Reseteaza contorul
* la zero
*/
public void reset()
{
 numar = 0;
}
}
```

Tablouri de obiecte

```
Counter[] counters;
```

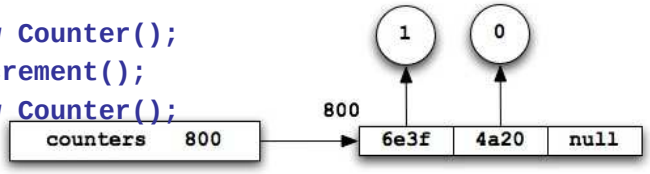
STACK HEAP

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



STACK HEAP

```
counters[0]=new Counter();
counters[0].increment();
counters[1]=new Counter();
```



STACK HEAP

"Traversarea" tablourilor de obiecte

- Aşa cum putem folosi cicluri pentru a traversa tablouri de tipuri primitive, putem face şi cu tablourile de obiecte
- Exerciţiu. Scrieţi o metodă care:
  - ia un singur argument: un tablou de contoare
  - returnează suma numerelor conţinute în contoare
  - mai întâi, presupuneţi că fiecare element de tablou se referă la un contor valid
  - Apoi rescrieţi metoda astfel ca să poată prelucra tablouri în care unele /toate elementele sunt **`null`**

Capcană: un tablou de caractere nu este un String

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

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

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

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

  - Obiectul **`s`** va avea aceeaşi secvenţă 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`** foloseşte o subgamă a tabloului de caractere

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

  - Fiind dat **`a`** ca mai înainte, noul obiect şir 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 iniţializarea 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};
```


Prescurtări la iniţializarea 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 si o declarație de variabilă

Transmiterea tablourilor ca parametri

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

Exemplu: "baza de date" PersonDB

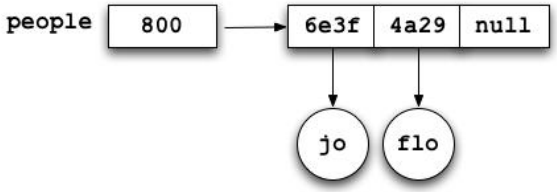
```
public class Person{
  private String nume;
  private int virsta;
  public Person(String nume, int virsta){
    this.nume = nume;
    this. virsta = virsta;
  }
  public Person(String nume){
    this(name, 5);
  }
  public int obtineVirsta() { return virsta; }
  public String obtineNume() { return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


Rezumat

- Metode: cum funcţionează apelul
- Crearea obiectelor
- Clase exemplu + ceva de făcut
- Jocul Hamurabi cu temă pentru laborator
- Sugestii pentru proiectarea claselor
- Tablouri:
  - Fundamente
  - Tablouri de tipuri primitive
  - Tablouri de obiecte
  - Traversarea
  - Iniţializarea
  - Transmiterea ca parametri