Programare orientată pe obiecte

1. Clase şi obiecte (continuare)
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 parametrii. La evaluarea argumentelor, acestea sunt asignate parametrilor locali din metoda apelată.
  4. Execută metoda.
  5. Revine din metodă. Memoria folosită pentru cadrul de stivă pentru metoda apelată este scoasă de pe stivă.

Transmiterea parametrilor

- In Java transmiterea parametrilor la apelul metodelor se face **numai prin valoare**
  - Modificarile aduse parametrilor în interiorul unei metode nu se păstrează la revenirea din metoda respectivă
- Dacă unul dintre parametrii unei metode are drept tip o clasă, aceasta înseamna că la apel metoda va primi referința unui obiect al clasei
  - Ceea ce se transmite prin valoare este chiar referința, NU obiectul indicat de ea
  - Metoda **nu va putea modifica referința** respectivă, dar **va putea modifica obiectul** indicat de ea

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*
  - *constructori*
- Toate cele trei mecanisme presupun 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 valorile lor 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

- **Exemplu:**

```
class Ex{

 int nr = 1;

 public Ex() {
        nr = 20;
 }

 public static void main(){
        Ex e  = new Ex();
        System.out.println(
                "Nr = "+ e.nr);
        }
}
```

Valori iniţiale implicite pentru câmpuri (variable instanță)

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
  - O valoare iniţială unui câmp la declararea sa
  - Exemplu
    - `public static int capacity = 10; //iniţializat la 10`
    - `private boolean full = false; //iniţializat la false`
- Blocuri de iniţializare statice
  - Bloc normal de cod între acolade, { } şi precedat de cuvântul cheie static
  - Exemplu
    - ```
      static {
            // codul necesar iniţializării se scrie aici
      }
      ```
  - Folosit la iniţializarea variabilelor la nivel de clasă
  - Blocurile de iniţializare statice sunt executate în ordinea în care apar în codul sursă
      ```

Iniţializarea câmpurilor

- Alternativă la blocurile statice: metodă statică privată
  - Exemplu

```
class Oricare {
        public static varType myVar = initializeClassVariable();
        private static varType initializeClassVariable() {
                //codul de iniţializare se pune aici
        }
}
```

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

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

Iniţializarea câmpurilor

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

```
class Oricare {
    private varType myVar = initializeInstanceVariable();
    protected final varType initializeInstanceVariable(){
        //cod de iniţializare
    }
}
```

  - Folositoare mai ales dacă subclasele doresc să refolosească metoda de iniţializare
  - Metoda este **final** deoarece apelul metodelor non-final la iniţializarea 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
- 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

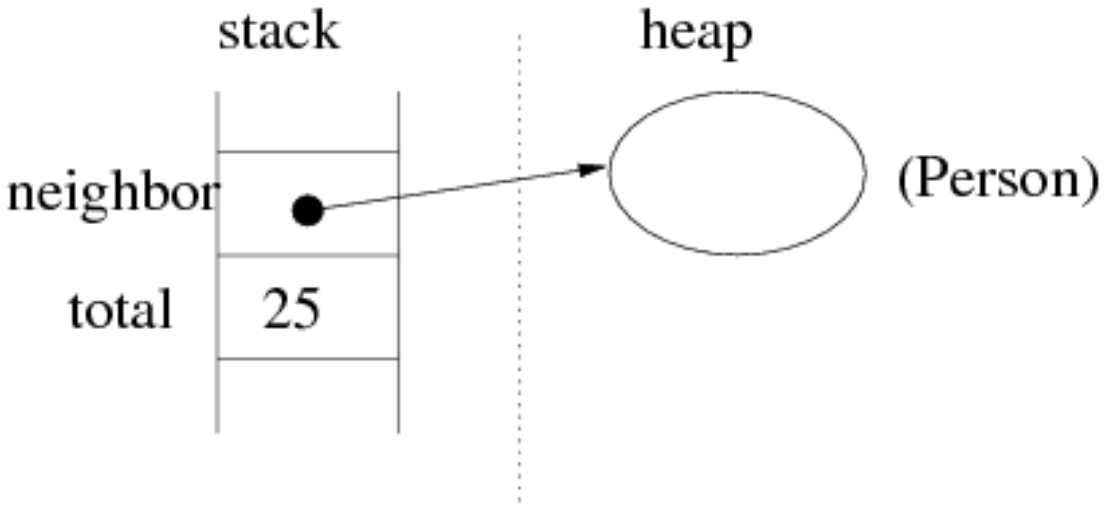
Crearea obiectelor

- Se poate combina declararea şi iniţializarea

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

- La fel ca și pentru tipurile primitive

```
int total = 25;
```



Constructori

- Orice clasă (inclusiv cele abstracte) trebuie să aibă cel puțin un constructor
  - Nu înseamnă că trebuie neapărat implementat unul
  - În lipsa codului explicit, Java generează implicit un constructor, caz în care variabilele instanță vor fi inițializate cu valorile implicite
- Exemplu de cod pentru implementarea uni constructor

```
class Person{
    String nume;
    int varsta;

    public Person(String n, int v){
     nume = n;
     varsta = v;
    }

    public Person() {
    }

    public static void main() {
       Person p = new Person("Ion", 28);
       // alt cod…
    }
}
```

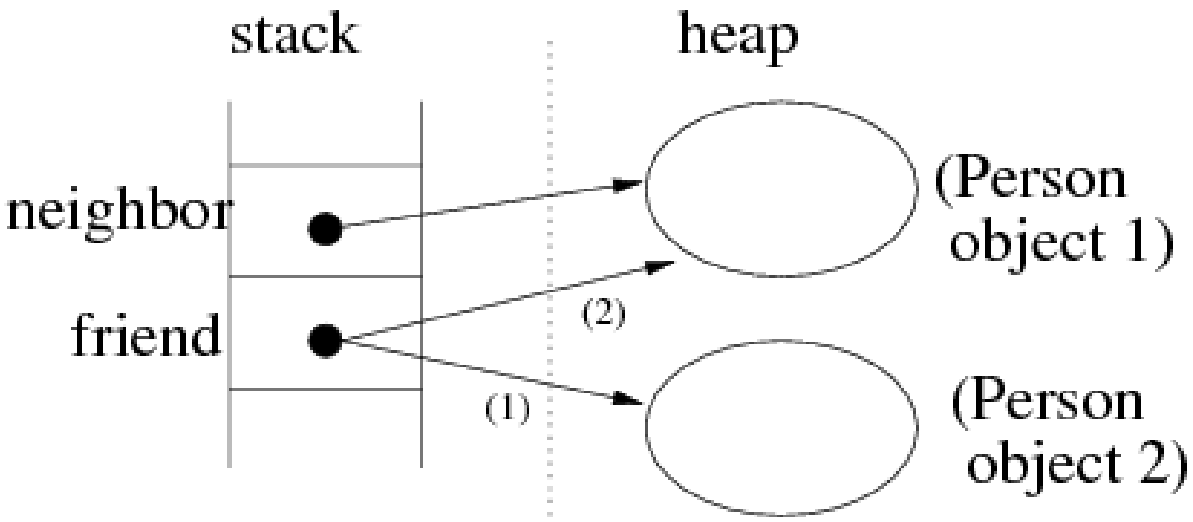
Asignarea obiectelor şi alias-uri

- 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`

Alocare memorie pe **stack** și **heap**

- De obicei metodele, variabilele și obiectele din programele Java sunt alocate în una din cele două locuri de memorie: **stack** sau **heap**

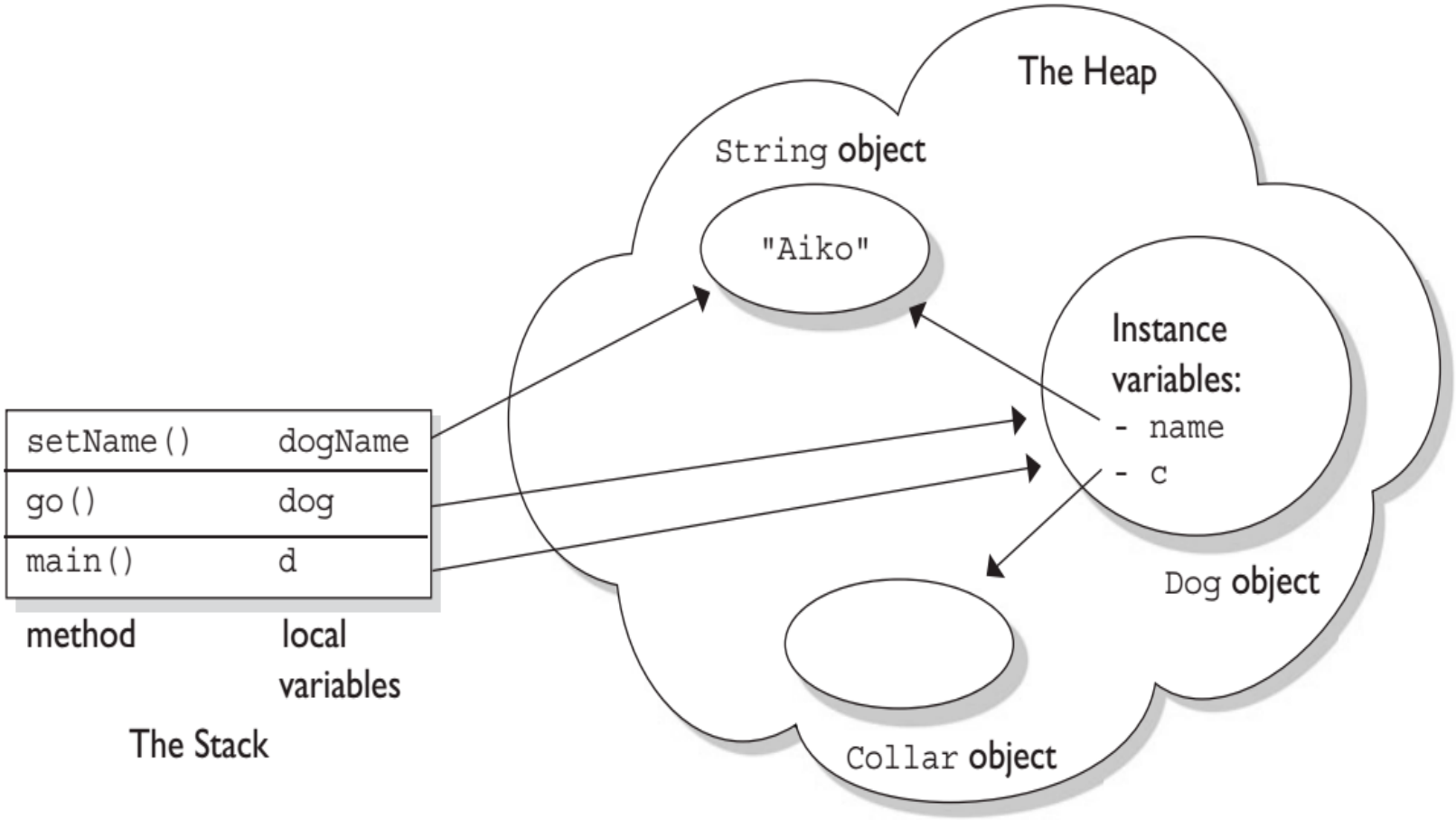
- Regula de bază este:
  - Variabilele instanță și obiectele se alocă pe **heap**
  - Variabilele locale se alocă pe **stack**

- Exemplu concret de cum se alocă variabilele dintr-un program Java:

Alocare memorie pe **stack** și **heap**

```
1. class Collar { }
2.
3. class Dog {
4.          Collar c; // variabilă instanță
5.          String name; // variabilă instanță
6.
7.          public static void main(String [] args) {
8.
9.                    Dog d; // variabilă locală
10.                   d = new Dog();
11.                   d.go(d);
12.         }
13.         void go(Dog dog) { // variabilă locală: dog
14.                   c = new Collar();
15.                   dog.setName("Aiko");
16.         }
17.         void setName(String dogName) { // variabilă locală: dogName
18.                   name = dogName;
19.                   // alte lucruri
20.         }
21. }
```

Alocare memorie pe **stack** și **heap**



Alocare memorie pe **stack** și **heap**

- Linia 7: metoda main() este plasată pe stivă
- Linia 9: variabila referință d este creată pe stivă (dar încă nu este nici un obiect de tip Dog)
- Linia 10: un obiect de tipul Dog este creat pe în memoria heap; variabila d este o referință către acest obiect
- Linia 11: o copie a variabilei referință d este transmisă ca argument la apelul metodei go()
- Linia 13: metoda go() este plasată pe stivă cu parametrul dog ca variabilă locală
- Linia 14: un nou obiect Collar este creat în memoria heap
- Linia 17: setName() este adăugată pe stivă cu parametrul dogName ca variabilă locală

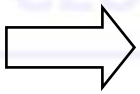
Alocare memorie pe **stack** și **heap**

- Linia 18: variabila instanță name referă acum către obiectul dogName
- După terminarea liniei 19, setName() este finalizată și este scoasă de pe stivă împreună cu toate variabilele sale locale (dogName). Doar obiectul String "Aiko" rămâne deoarece mai există o referință către el
- După terminarea liniei 15, metoda go() este scoasă de pe stivă. Obiectul String "Aiko" nu mai este referit
- După terminarea liniei 11, metoda main este scoasă de pe stivă. Obiectul d nu mai este referit. Execuția programului se termină

Reguli pentru proiectarea claselor

- *Iniţializaţi* întotdeauna datele
  - Java nu va iniţializa variabilele locale, dar va iniţializa variabilele instanţă 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;
```



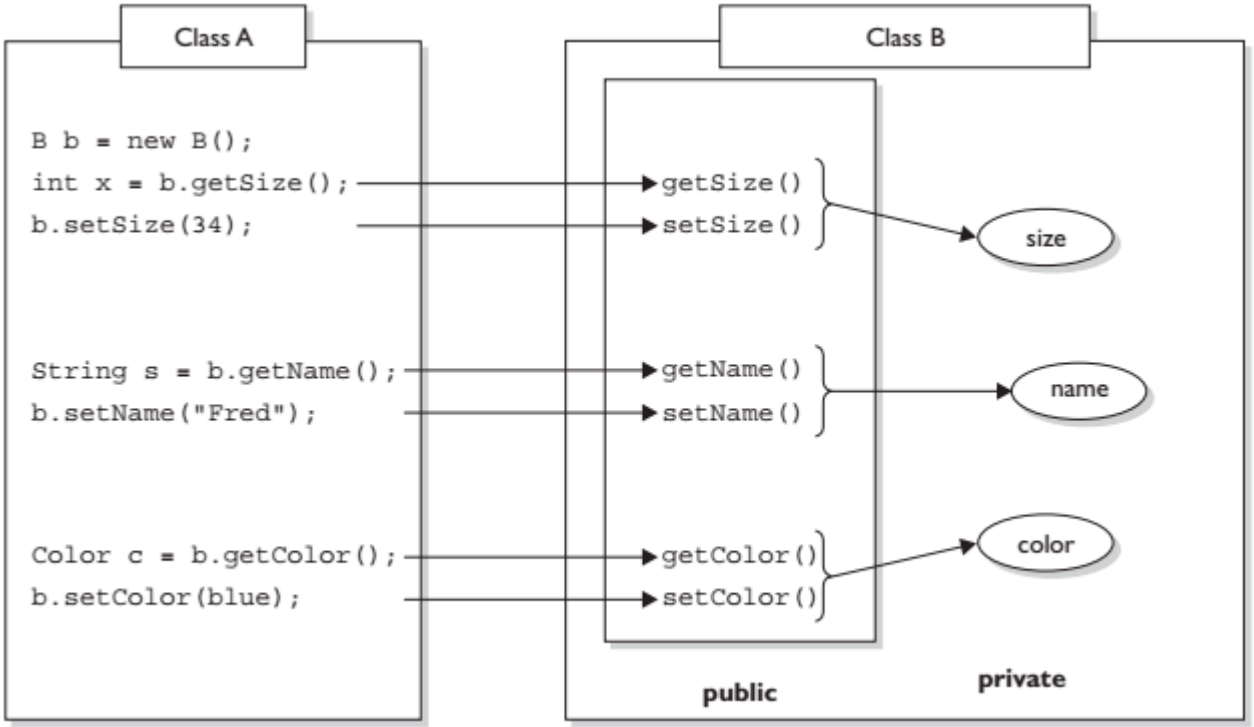
```
class Adresa {
    private String strada;
    private String oras;
    private String stat;
    private String tara;
    private int codPostal;
}
```

Reguli pentru proiectarea claselor

- Folosiți **încapsularea**
  - Păstraţi datele întotdeauna private
    - Schimbările în reprezentarea datelor nu vor afecta utilizatorii claselor; erorile sunt mai uşor de detectat
  - Pentru accesul sau modificarea valorilor variabilelor instanță se vor implementa metode *accesori* şi *mutatori* (*getter* și *setter)*
  - Nu toate câmpurile necesită accesori şi mutatori individuali
    - Exemplu: Angajat – obţine şi setează salariul, dar nu şi data angajării (nu se schimbă o dată creată)

Reguli pentru proiectarea claselor

- Exemplu de cod pentru încapsulare:



  - Clasa A nu poate accesa variabilele instanță ale clasei B decât prin intermediul metodelor getter și setter

Reguli pentru proiectarea claselor

- Contra-exemplu de cod pentru încapsulare:

```
class Foo {
        public int left = 9;
        public int right = 3;
        public void setLeft(int leftNum) {
                left = leftNum;
                right = leftNum/3;
        }
}
```

  - Întrebare: Variabila right va fi totdeauna 1/3 din left?
  - Răspuns: Nu, deoarece utilizatorii clasei Foo pot seta direct valoarea variabilelor left și right, fără să apeleze metoda setLeft()

Reguli 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

Reguli pentru proiectarea claselor

- Divizaţi clasele cu prea multe responsabilităţi, de exemplu

```
class PachetCarti { // nu este recomandat
  public 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!!!
```

Reguli pentru proiectarea claselor

- Faceţi numele claselor şi metodelor să reflecte responsabilităţile acestora
- O convenţie bună este:
  - Numele clasei: substantiv (ex. Comanda) sau substantiv+adjectiv (ex. ComandaUrgenta)
  - Numele metodelor: verbe; accesorii încep cu "get"; mutatorii încep cu "set"

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)

O clasă "Comutator" simplă

```
// Un comutator on/off simplu

class ComutatorSimplu {

    // Comută pe închis.
    public void inchide() {
        System.out.println("Inchide
                           comutatorul");
        seteazaInchis(true);
    }

    // Comută pe deschis.
    public void deschide() {
        System.out.println(" Deschide
                 comutatorul");
        seteazaInchis(false);
    }

    // Raportează dacă comutatorul
    //este închis sau nu.
    public boolean esteInchis () {
       return obtineStareInchis();
    }

// Returnează starea comutatorului
 private boolean obtineStareInchis()
    {  return inchis;  }

// Seteaza starea comutatorului.
 private void seteazăInchis(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

Tipuri 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*: clase 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 se face obișnuit de oriunde
  - Excepție fac metodele statice ale clasei exterioare când 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();
    // Definire de referințe către clase
    //interne:
    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 metoda1() {}
        //alte metode
    }
}
```

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() { ...}
   private class MyInner  {
      public void doSomething() {...}
      public void method() {
          //Apelul unei metode cu același nume a clasei exterioare
         MyOutter.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*: este o clasă locală fără nume
  - Exemplu:

```
MyAnonimousClass frenchGreeting = new MyAnonimousClass () {
                    private String info;
                    public MyAnonimousClass(String inf) {info=inf;}
                    public String getInfo() {return info;}
                };
```

- Sunt foarte des folosite la implementarea interfețelor grafice când este nesesar să se adauge ascultători anumitor anumitor componente

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

```
//creează un tablou de mărime 12
double[] precipitatii;
precipitatii = new double [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:
  - Capacitatea poate fi insuficientă 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 variabile referință

`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

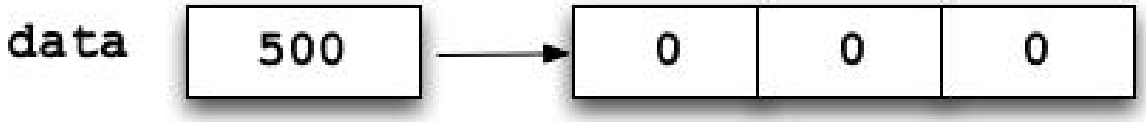
```
public class ArrayTool{
  public int sum(int[] data){
    int sum = 0;
    for (int i = 0; i < data.length; i++){
      sum += data[i];
    }
    return sum;
  }
  public int sum2(int[] data){
    int sum = 0;
    for (int i = 0; i <= data.length; i++){
      sum += data[i];
    }
    return sum;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

Folosirea acestei comparaţii produce aruncarea unei excepţii ArrayIndexOutOfBoundsException

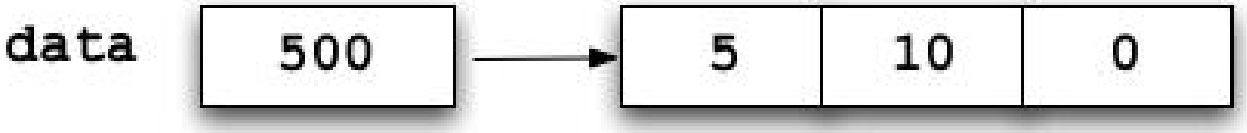
Tablouri de tipuri primitive

```
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[] pfile = new char[500];
deschide un fişier pentru citire
while (mai sunt caractere în fişier & pFile nu
                                        este plin)
    pfile[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';
```

Tablouri de obiecte

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 numar;
    }

    /**
    * Incrementeaza contorul
    * cu unu
    */
    public void increment() {
      numar++;
    }

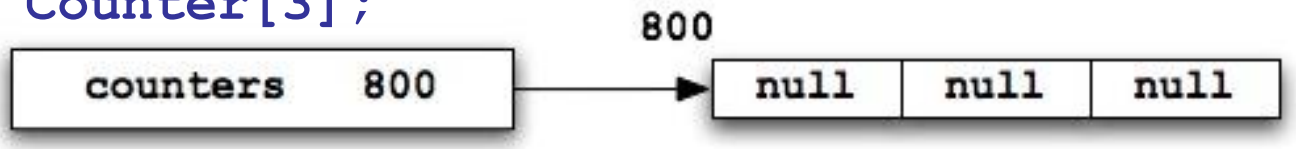
    /**
    * Reseteaza contorul
    * la zero
    */
    public void reset() {
     numar = 0;
    }
}
```

Tablouri de obiecte

```
Counter[] counters;
```

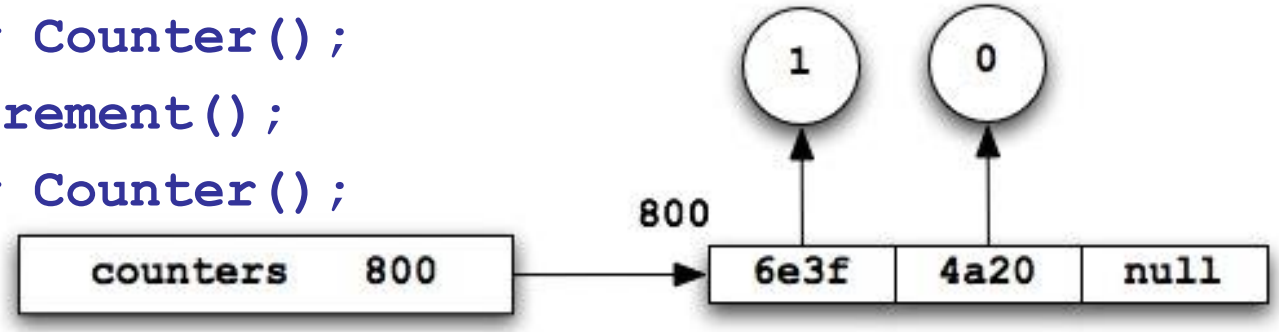
STACK HEAP

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



STACK HEAP

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



STACK HEAP

Exemplu: 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;
  }
```

Exemplu: Un joc Tic-Tac-Toe

```
/**
 * @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 a-l face jucabil (poate mai definiţi și alte clase?)
- Modificaţi clasa astfel încât să permită dimensiuni mai mari ale tablei de joc

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)};
```

Observaţie: Construcţia sintactică "new type[ ]" poate fi folosită la o asignare care nu este şi o declaraţie de variabilă

Transmiterea tablourilor ca parametri

- Atunci când nu mai există nici o referinţă spre un obiect, sistemul va şterge obiectul şi va elibera memoria ocupată de acesta
  - Ştergerea unui obiect se numeşte *dealocarea* memoriei
  - Procesul de dealocare a memoriei se numeşte *colectarea reziduurilor* și este realizat 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

Capcană: un tablou de caractere nu este un String

- Un tablou de caractere nu este un obiect de clasă **`String`**

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

- Un tablou de caractere poate fi convertit la un obiect de clasă **`String`**
- Un tablou de caractere este conceptual o listă de caractere şi de aceea este 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 String este `"AB"`
- Un tablou de caractere are ceva în comun cu obiectele `String`
  - Un tablou de caractere poate fi tipărit folosind `println`

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

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

```
ABC
```

Copierea tablourilor

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

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

Exemplu:

```
import java.lang.*;
public class SystemDemo {
   public static void main(String[] args) {
    // Declară două tablouri:
    int[] arr1 = {1,2,3,4,5,6};
    int[] arr2 = {0,2,4,6,8,10};

    /* Copiază două elemente din arr1 începând
        cu cel de-al doilea element în arr2
        începând cu cel de-al patrulea
        element: */
    System.arraycopy(arr1, 1, arr2, 3, 2);

    // Afișează conținutul lui arr2:
    for (int i=0; i<arr2.length; i++)
      System.out.println("Elementul #" + i +
                         " = " + arr2[i]);
   }
}
```