

# Cuprins detaliat Curs LPAI

(2005-2006)

<b>1. Introducere .....</b>	<b>4</b>
<b>1.1. Introducere in sisteme programabile (calculatoare) si programare .....</b>	<b>4</b>
<b>1.2. Dezvoltarea programelor in limbajul Java. Masina virtuala Java (JVM).....</b>	<b>6</b>
<b>2. Introducere in limbajul Java .....</b>	<b>8</b>
<b>2.1. Etapele dezvoltarii programelor Java si instrumentele folosite .....</b>	<b>8</b>
<b>2.2. Exemplu introductiv.....</b>	<b>9</b>
2.2.1. Cod si cuvinte cheie.....	9
2.2.2. Membri si operatori .....	9
2.2.3. Analiza programului, linie cu linie .....	10
2.2.4. Exemplificarea etapelor dezvoltarii programelor Java .....	11
<b>2.3. Elementele de baza ale limbajului Java.....</b>	<b>12</b>
2.3.1. Comentarea codului .....	12
2.3.2. Cuvintele cheie Java .....	12
2.3.3. Tipurile primitive Java.....	13
2.3.4. Variabile de tip primitiv.....	17
2.3.5. Valorile literale Java .....	18
2.3.6. Operatori Java.....	20
2.3.7. Instructiuni pentru controlul executiei programului Java .....	23
<b>2.4. Tipuri referinta.....</b>	<b>30</b>
2.4.1. Introducere in tipuri referinta Java.....	30
2.4.2. Tablouri cu elemente de tip primitiv.....	30
2.4.3. Variabile obiect (de tip clasa) in Java .....	36
2.4.4. Tablouri cu elemente de tip referinta .....	38
<b>3. Elemente de programare orientata spre obiecte .....</b>	<b>42</b>
<b>3.1. Conceptul de obiect .....</b>	<b>42</b>
<b>3.2. Conceptele de bază ale abordării OO .....</b>	<b>42</b>
3.2.1. Obiectele .....	43
3.2.2. Mesajele si comunicatia între obiecte .....	47
3.2.3. Clasa (tipul de date al obiectelor) .....	47
3.2.4. Relații între clase .....	52
3.2.5. Ierarhii de clase.....	54
<b>3.3. Crearea claselor Java.....</b>	<b>76</b>
3.3.1. Structura codului unei clase Java.....	76
3.3.2. Elementele declarației de clasa în Java .....	77
3.3.3. Corpul clasei în Java .....	78
3.3.4. Atributele (variabilele membru) Java .....	78
3.3.5. Constructorii (initializatorii) obiectelor Java .....	80
3.3.6. Metodele (functiile membru) Java .....	82
3.3.7. Controlul accesului la membri Java - nivelurile de încapsulare.....	85
3.3.8. Diferențele între membri de instanță și membri de clasă în Java.....	85
<b>4. Exemple de clase de biblioteca Java.....</b>	<b>86</b>
<b>4.1. Clase Java pentru lucrul cu (siruri de) caractere.....</b>	<b>86</b>
4.1.1. Încapsularea caracterelor si a sirurilor de caractere .....	86
4.1.2. Clasa care încapsuleaza caractere Unicode (Character) – interfața publica.....	86

4.1.3. Clasa care incapsuleaza siruri de caractere nemodificabile (String) – interfata publica .....	88
4.1.4. Clasa care incapsuleaza siruri de caractere modificabile (StringBuffer) – interfata publica .....	95
<b>4.2. Clase predefinite pentru incapsularea tipurilor primitive. Conversii .....</b>	<b>100</b>
4.2.1. Incapsularea tipurilor primitive .....	100
4.2.2. Clasa care incapsuleaza intregi de tip int (Integer) – interfata publica .....	100
<b>4.3. Clase Java pentru operatii de intrare-iesire (IO) .....</b>	<b>104</b>
4.3.1. Clasificarea fluxurilor IO in functie de tipul de data transferate .....	104
4.3.2. Clasificarea fluxurilor IO in functie de specializare .....	108
4.3.3. Fluxuri terminale ( <i>data sink</i> ) .....	108
4.3.4. Fluxuri de prelucrare .....	111
<b>5. Elemente de programare Java pentru retele bazate pe IP .....</b>	<b>118</b>
<b>5.1. Introducere in Protocolul Internet (IP) si stiva de protocoale IP .....</b>	<b>118</b>
5.1.1. Elemente de terminologie a retelor de comunicatie .....	118
5.1.2. Modele de comunicatie stratificata .....	118
5.1.3. Modelul de interconectare a sistemelor deschise (modelul OSI) .....	120
5.1.4. Modelul de comunicatie si protocoalele Internet .....	120
5.1.5. Detalii utile in programare privind protocoalele Internet .....	122
5.1.6. Introducere in <i>socket-uri</i> .....	124
<b>5.2. Incapsularea adreselor IP in limbajul Java .....</b>	<b>125</b>
5.2.1. Incapsularea adreselor IP .....	125
5.2.2. Clasa <code>java.net.InetAddress</code> – interfata publica .....	125
<b>5.3. Socket-uri flux (TCP) .....</b>	<b>129</b>
5.3.1. Lucrul cu socket-uri flux (TCP) .....	129
5.3.2. Clasa socket flux (TCP) pentru conexiuni (Socket) – interfata publica .....	130
5.3.3. Clasa socket TCP pentru server (ServerSocket) – interfata publica .....	134
5.3.4. Clienti pentru servere flux (TCP) .....	136
5.3.5. Servere flux (TCP) non-concurente .....	139
5.3.6. Fire de executie (threads) .....	144
5.3.7. Clasa Thread – interfata publica .....	146
5.3.8. Servere flux concurente .....	150
<b>5.4. Socketuri datagrama (UDP) .....</b>	<b>153</b>
5.4.1. Lucrul cu socketuri datagrama (UDP) .....	153
5.4.2. Clasa pachet (datagrama) UDP (DatagramPacket) – interfata publica .....	154
5.4.3. Clasa socket UDP (DatagramSocket) – interfata publica .....	157
5.4.4. Programe ilustrative pentru lucrul cu <i>socket-uri</i> datagrama (UDP) .....	159
<b>6. Elemente de programare Java pentru Web .....</b>	<b>166</b>
<b>6.1. Introducere in arhitectura si tehnologiile Web .....</b>	<b>166</b>
<b>6.2. Identificarea si accesul la resursele retelelor IP .....</b>	<b>166</b>
6.2.1. Caracteristicile URL-urilor .....	166
6.2.2. Crearea unui URL .....	167
6.2.3. Analiza lexicala a unui URL .....	168
6.2.4. Citirea direct dintr-un URL .....	169
6.2.5. Conectarea la un URL .....	170
<b>6.3. Applet-uri (miniaplicatii) Java .....</b>	<b>171</b>
6.3.1. Caracteristicile <i>applet-urilor</i> Java .....	171
6.3.2. Ciclul de viata al <i>applet-urilor</i> Java .....	171
6.3.3. Dezvoltarea unui <i>applet</i> Java .....	172
<b>6.4. Interfete grafice Java. Biblioteci grafice Java. Java Swing. Java Beans .....</b>	<b>175</b>
6.4.1. Elementele unei aplicatii grafice Swing .....	175
6.4.2. Modalitati de a crea containerul de nivel maxim .....	179
6.4.3. Crearea interactivitatii aplicatiilor si miniaplicatiilor grafice Swing .....	182
6.4.4. Utilizarea componentelor grafice Swing pentru lucrul cu text .....	185
<b>6.5. Lucrul cu programe CGI in Java .....</b>	<b>190</b>
<b>6.6. Servleturi (miniservere) Java. Tehnologia Java Server Pages (JSP) .....</b>	<b>191</b>
<b>7. Alte limbaje de programare pentru Web .....</b>	<b>192</b>

<b>7.1. Meta-limbajul XML si limbajele derivate din XML .....</b>	<b>192</b>
<b>7.2. Limbajul JavaScript.....</b>	<b>193</b>
<b>7.3. Limbajul PhP .....</b>	<b>194</b>
<b>7.4. Limbajul ASP .....</b>	<b>195</b>
<b>7.5. Limbajul C# (C sharp) si tehnologia .NET .....</b>	<b>196</b>
<b>7.6. Alte limbaje .....</b>	<b>197</b>

---

# 1. Introducere

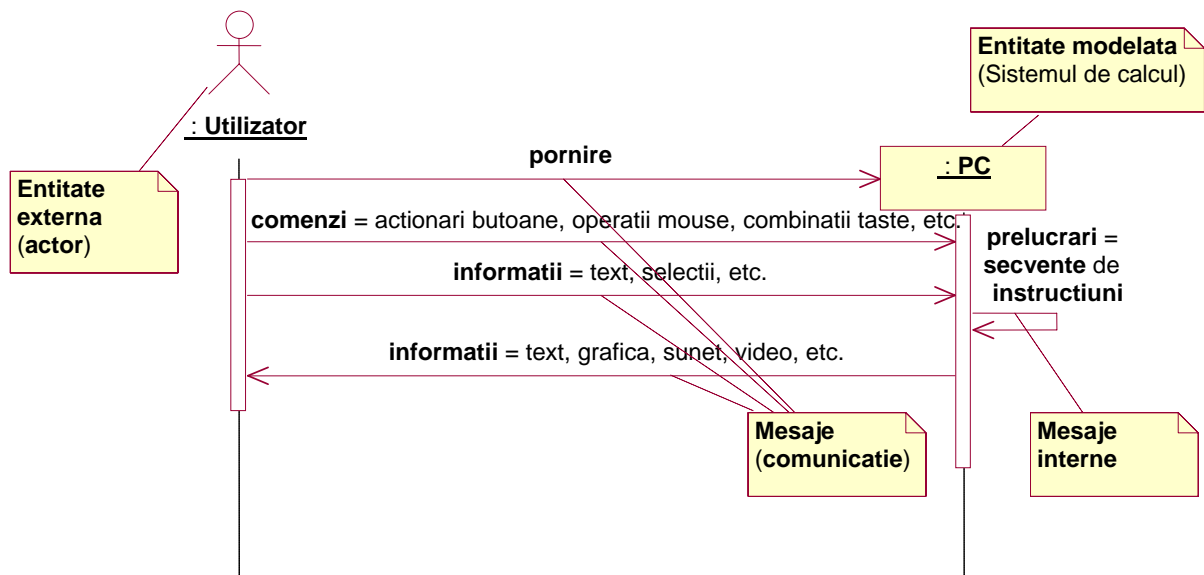
## 1.1. Introducere in sisteme programabile (calculatoare) si programare

**Calculatorul** este un **sistem electronic** (o masina electronica) care permite realizarea unor sarcini (*tasks*) **in mod repetat**. Aceste sarcini poarta numele de **programe**, iar calculatorul mai poate fi denumit si **sistem programat** sau **sistem de calcul**. Infrastructura electronica a sistemului de calcul este numita *hardware*, iar programele sunt numite *software*.

**Hardware-ul** cuprinde in primul rand **procesorul** (care efectueaza calculele, prelucrarile) si **memoria** (in care se stocheaza programe si datele necesare acestora pentru a realiza sarcinile date), care sunt direct conectate. In plus mai contine **elemente periferice** (memorie externa, tastatura, mouse, placa video, monitor, imprimanta, placa de retea, placa de sunet), **conectate prin intermediul unor dispozitive numite porturi** (de intrare-iesire).

Pentru descrierea vizuala a interactiunilor intre diferite entitati putem folosi o **diagrama numita MSC** (*Message Sequence Chart*). In **limbajul de modelare unificat** (UML = *Unified Modeling Language*) diagramele MSC aplicate entitatilor numite obiecte se numesc **diagrame de secventa** (a mesajelor schimbate intre obiecte). In continuare vor fi folosite diagrame MSC bazate pe notatiile UML.

**Interactiunea dintre utilizator si sistemul de calcul** (numit **PC**, de la *Personal Computer*) poate fi descrisa prin urmatorul MSC:



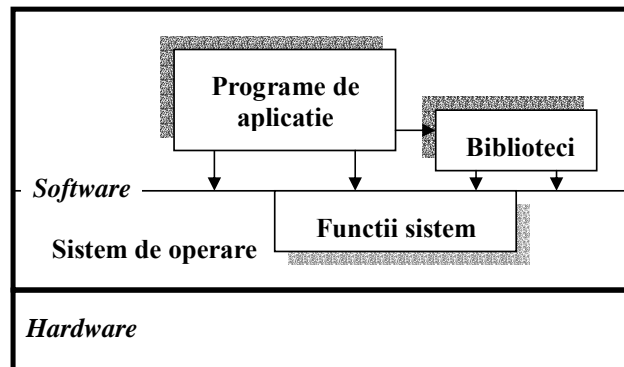
Pentru comentarea elementelor diagramei s-a folosit simbolul UML:

Comentariu

Se observa ca **sistemul de calcul primeste comenzi si informatii de la utilizator** (care lanseaza si configureaza programele) si ii furnizeaza acestuia informatii (rezultatele executiei programelor).

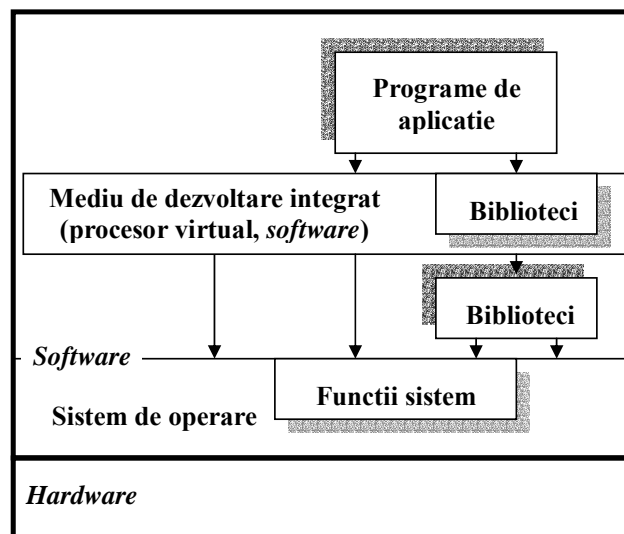
**Programele** sunt **secvente** de operatii de baza (numite **instructiuni**) efectuate de procesor. *Software-ul* cuprinde un **nivel de baza**, numit **sistem de operare**, care face posibil lucrul in mod transparent cu elementele *hardware*. Sistemul de operare ofera programatorilor un set de functii (numite functii sistem). **Programele realizate de programatori** sunt executate prin intermediul sistemului de operare (*peste* sistemul de operare). Apelul la functiile oferite de sistemul de operare se numesc apeluri sistem. Pentru a usura programarea, programatorii au la dispozitie, in general, **colectii de programe** numite **biblioteci** (*libraries*), care utilizeaza fie apeluri sistem, fie lucrul direct cu elementele *hardware*.

Un **sistem programabil traditional** cuprinde, asadar, un nivel *hardware* gestionat de un nivel *software* (sistemul de operare) si completat de alt nivel *software* (bibliotecile de programe). Programatorii creeaza noi programe, care fac apel la biblioteci, fac apeluri sistem, si eventual lucreaza direct cu elementele *hardware*. Aceasta este stiva de niveluri din modelul traditional.



In anumite situatii, cum este cea a utilizarii de catre programatori a unui instrument de dezvoltare a programelor numit **mediu de dezvoltare integrat**, **IDE** (*Integrated Development Environment*), intre programele create de programatori si sistemul de operare intervine un nou nivel, cel al unei **masini virtuale** (procesor virtual, *software*). IDE-ul **poate oferi biblioteci** si **preia rolul de sistem de executie de la sistemul de operare**, pentru programele dezvoltate, **pe durata dezvoltarii acestor programe**.

Stiva modelului traditional cu IDE este urmatoarea:

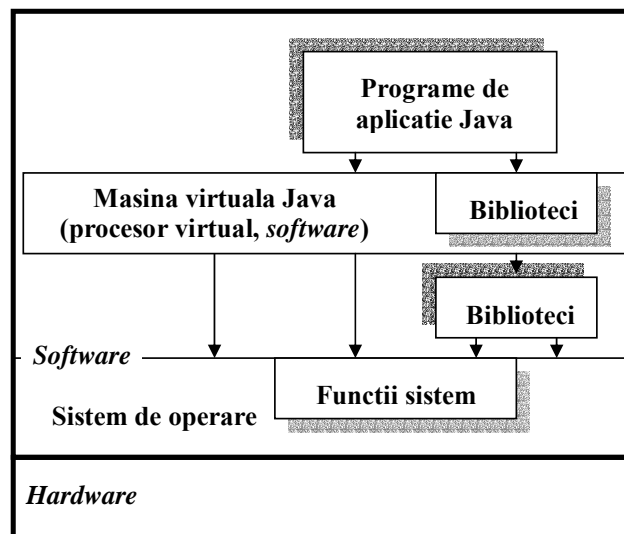


O **posibila problema** este in acest caz **dependenta introdusa de bibliotecile pe care le ofera IDE-ul**. Totusi, un IDE bun ofera o multitudine de **unelte de dezvoltare** bine organizate si exercita un **control mai strict** (decat cel exercitat de sistemul de operare) **al resurselor utilizate** de programele dezvoltate.

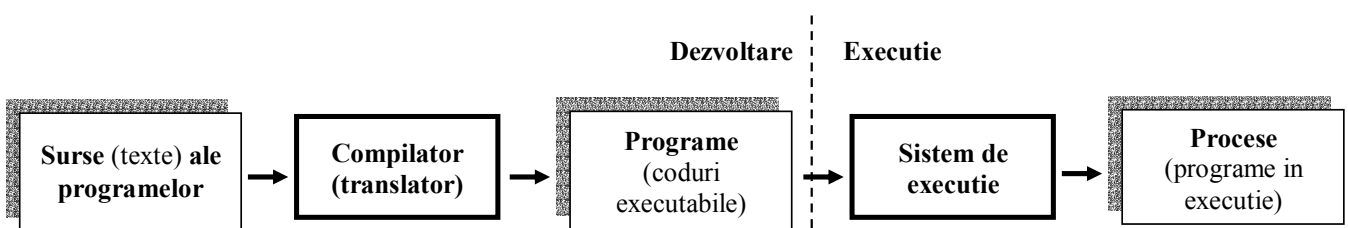
## 1.2. Dezvoltarea programelor in limbajul Java. Masina virtuala Java (JVM)

Limbajul Java utilizeaza o abordare asemanatoare IDE-urilor, bazandu-se pe o **masina virtuala Java**, numita **JVM (Java Virtual Machine)**, care este un **procesor virtual (software)** prin intermediul caruia sunt executate programele Java.

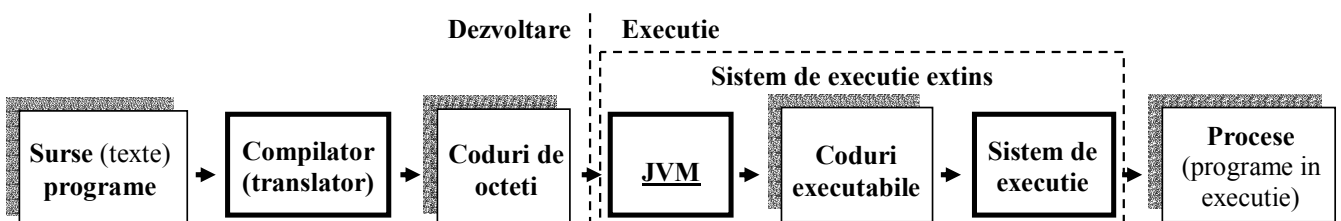
Executia programelor de catre masina virtuala Java se numeste **interpretare**, deoarece codul de octeti Java (obtinut in urma compilarii) este interpretat de JVM, adica este convertit in timp real in cod direct executabil pe sistemul de operare curent utilizat, pentru ca apoi sa fie executat. Stiva **modelului Java** este urmatoarea:



Pe un sistem programabil traditional programele sunt mai intai **convertite** (traduse, translatate, *compile*) de la limbajul de programare utilizat de programator la coduri executabile de procesorul hardware, pentru ca apoi sa fie **executate sub controlul sistemului de operare**.

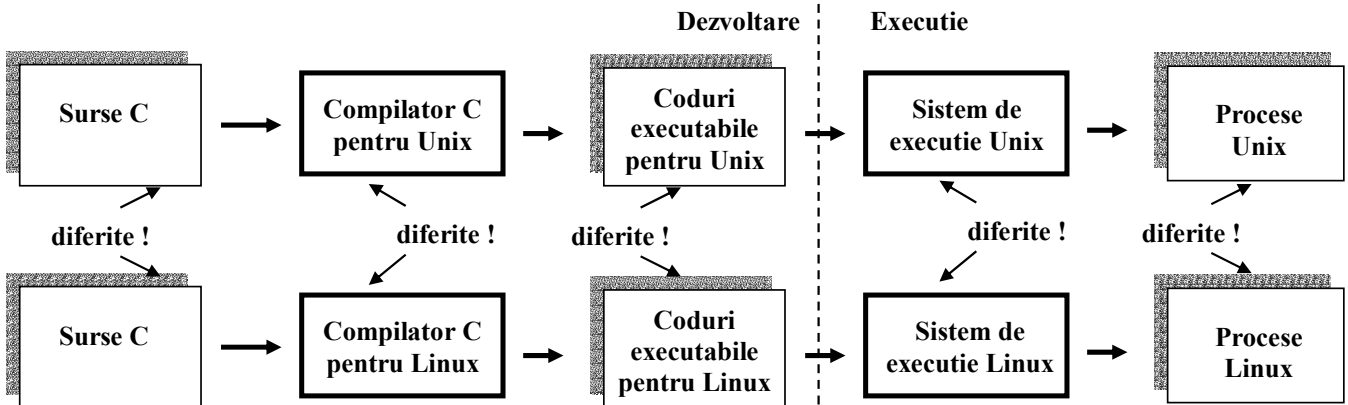


In **sistemul de programare Java** programele sunt *compile* de la limbajul de programare Java la coduri executabile de procesorul software (JVM), numite coduri de octeti (*bytecodes*), pentru ca apoi *codurile de octeti* sa fie **executate de JVM** (iar JVM este executata sub controlul sistemului de operare).

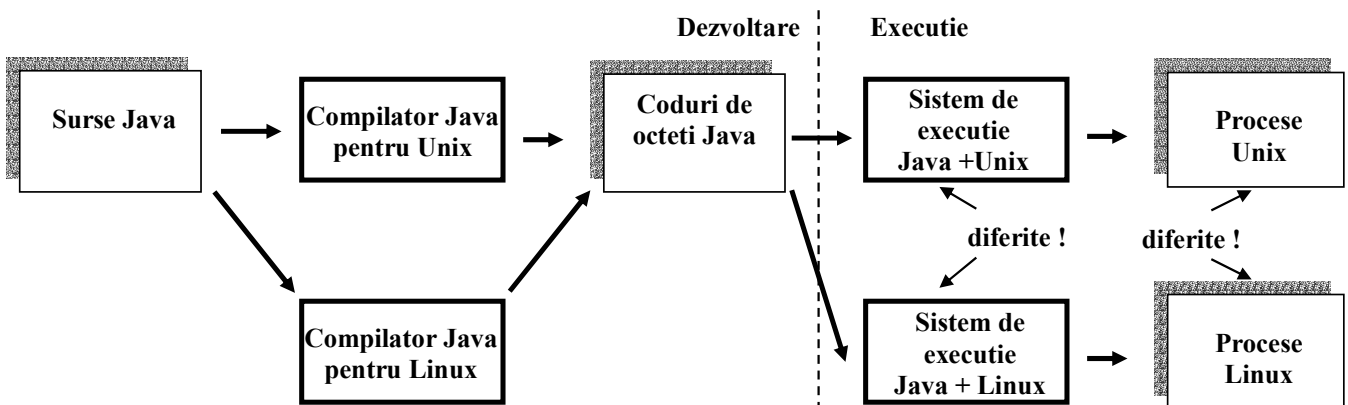


**Avantajul obtinut de Java** este acela ca, daca se creaza cate un procesor *software* Java (masina virtuala Java – JVM) pentru fiecare tip de sistem de operare existent, atunci pe orice calculator poate fi instalata o JVM, pe care apoi pot fi executate *coduri de octeti* Java care au fost compilate pe orice alt calculator. *Codurile de octeti* Java sunt **portabile**, iar limbajul Java este **neutru** din punct de vedere al arhitecturii (sistemului de operare al) calculatorului.

**Dezvoltarea traditionala, in C de exemplu, trebuie realizata separat pentru fiecare tip de sistem de executie.**



**Dezvoltarea in Java realizata pentru oricare tip de sistem de executie conduce la acelasi cod de octeti, interpretabil de toate masinile virtuale Java, indiferent de tipul de sistem de executie.**



**Principalul dezavantaj al programelor Java** este viteza de executie redusa, comparativ cu cea a limbajelor traditionale. Acesta este efectul interpretarii programelor Java (dubla procesare a lor, la nivelul interpretorului si la nivelul sistemului de operare si al *hardware*-ului).

O **solutie de imbunatatire a vitezei de executie** este utilizarea unui **compiler Java** de tipul **Just-In-Time (JIT)**, care genereaza coduri direct executabile. Dezavantajele unei astfel de solutii sunt cele ale modelului traditional, adica dependenta compilatoarelor JIT de platforma de executie, lipsa portabilitatii executabilelor.

## 2. Introducere in limbajul Java

### 2.1. Etapele dezvoltarii programelor Java si instrumentele folosite

Programele sunt **dezvoltate** (concepute, editate, compilate, verificate) de catre **programatori**, si **executate (folosite)** de catre **utilizatori**. Utilizatorii fac apel la programele create de programatori pentru a atinge diferite obiective (sarcini de serviciu asistate de calculator, educatie electronica, divertisment, etc.).

**Conceperea (proiectarea) programului** inseamna **cel putin scrierea pe hartie a unei schite a codului sau a unui pseudocod** (schita a programului scrisa in limbaj natural), **dar poate include si aplicarea unor instrumente** (diagrame, limbaje cum ar fi UML – Limbajul de Modelare Unificat) **si metodologii** mai complicate (cum ar fi programarea agila/extrema sau ROSE – Ingineria Software Orientata-spre-obiecte Rational).

**Odata proiectat, codul Java trebuie editat.** In Java tot codul este organizat in clase. Dupa editarea unei clase cu numele `<NumeClasa>` intr-un editor un editor de text, continutul trebuie salvat intr-un fisier cu numele `<NumeClasa>.java`.

**Limbajul Java este case-sensitive** (face deosebirea intre litere mici si mari), **inclusiv in ceea ce priveste numele claselor si fisierelor**, iar numele fisierelor (urmate de extensia `.java`) trebuie sa fie identice cu numele claselor.

**Pentru obtinerea codului de octeti Java**, trebuie compilat codul din acest fisier. Daca se presupune utilizarea compilatorului Java (`javac`) din linia de comanda (consola standard de intrare), atunci trebuie executata urmatoarea comanda, in directorul `directorcurent`, in care se afla fisierul `<NumeClasa>.java`:

```
directorcurent> javac <NumeClasa>.java
```

In urma acestei comenzi, compilatorul Java va crea genera codul de octeti (neutru din punct de vedere architectural, adica acelasi pentru orice pereche {sistem de operare, sistem de calcul} pe care e compilat) corespunzator intr-un fisier cu numele `<NumeClasa>.class`, in directorul curent (acelasi director in care se afla si fisierul `<NumeClasa>.java`). Compilatorul Java genereaza cate un fisier pentru fiecare clasa compilata.

**Pentru executia programului**, acesta trebuie lansat in interpretor (`java`), folosind comanda:

```
directorcurent> java <NumeClasa>
```

(numele clasei Java este argument pentru programul interpretor Java, numit `java`).

**Daca in urma compilarii apar erori**, ele trebuie corectate (urmarind si **indicatiile din mesajele de eroare**), revenind la etapa conceperii si editarii. **O alternativa** este folosirea **utilitarului de depanare a programelor Java**.

**Daca in urma executiei apar erori de conceptie** (comportamentul programului difera de cel dorit), ele trebuie corectate revenind la etapa conceperii si editarii.

O alternativa la dezvoltarea programelor utilizand direct compilatorul si interpretorul Java este utilizarea unuia dintre mediile integrate (IDE-urile) Java.

---



## 2.2. Exemplu introductiv

### 2.2.1. Cod si cuvinte cheie

In urmatorul program simplu Java sunt evidentiata cu format intensificat (*bold*) cuvintele cheie Java folosite.

```
1  public class Salut { // declaratia clasei
2      public static void main(String[] args) { // declaratia unei metode
3          System.out.println("Buna ziua!"); // corpul metodei
4      } // incheierea corpului metodei
5  } // incheierea corpului clasei
```

**Cuvintele cheie** (*keywords*) sunt cuvinte pe care limbajul Java le foloseste in scopuri precise, si care nu pot fi utilizate de programator in alte scopuri (sunt cuvinte rezervate).

**Cuvintele cheie de mai sus au, in general, urmatoarele semnificatii:**

- **public**: specificator (calificator, modificador) al *modului de acces* la clase (tipuri complexe), metode (functii) si atribute (variabile avand drept scop clasele)
- **class**: declara o *clasa Java* (tip de date complex)
- **static**: specificator (calificator, modificador) al caracterului *de clasa* al unei metode sau al unui atribut (in lipsa lui, caracterul implicit al unei metode sau al unui atribut este *de obiect*)
- **void**: specifica faptul ca metoda nu returneaza nimic

**In particular**, cuvintele cheie de mai sus au urmatoarele semnificatii:

- **public** din linia 1: codul clasei `Salut` poate fi accesat de orice cod exterior ei
- **class**: declara clasa Java `Salut`
- **public** din linia 2: codul metodei `main()` poate fi accesat de orice cod exterior ei
- **static**: metoda `main()` este o metoda cu caracter *de clasa* (nu cu caracter *de obiect*)
- **void**: metoda `main()` nu returneaza nimic

### 2.2.2. Membri si operatori

**Metodele** Java (numite si functii membru sau operatii) **si atributele** Java (numite si proprietati sau campuri) **poarta denumirea colectiva de membri** ai claselor Java.

**Caracterul global (de clasa)** dat de cuvantul cheie **static** precizeaza faptul ca membrul pe care il precede este parte a intregii clase (global, unic la nivel de clasa), si nu parte a obiectelor (variabilelor de tip clasa) particulare. Lipsa acestui cuvantul cheie indica un membru cu **caracter individual (de obiect)**, care e distinct pentru fiecare obiect.

**Operatorii** utilizati in programul de mai sus sunt:

- operatorul de declarare a **blocurilor** (acolade: "{ " si " }"),
  - operatorul **listei de parametri ai metodelor** (paranteze rotunde: "( " si " )"),
  - operatorul de **indexare a tablourilor** (paranteze drepte: "[ " si " ]"),
  - operatorul de **calificare a numelor** (punct: "."),
  - operatorul de **declarare a sirurilor de caractere** (ghilimele: "\"" si "\""),
  - operatorul de **sfarsit de instructiune** (punct si virgula: ";").
-

### 2.2.3. Analiza programului, linie cu linie

Sa analizam acum programul, linie cu linie, si element cu element.

```
1 public class Salut {
```

**Linia 1 declara o clasa Java** (cf. `class`), numita `salut`, al carei cod poate fi accesat de orice cod exterior ei (cf. `public`). Altfel spus, **codul clasei `salut` este neprotejat si deschis tuturor celorlalte clase**. Dupa declaratia clasei, urmeaza corpul ei, aflat intre elementele operatorului de declarare a blocurilor.

**Clasa este o constructie orientata spre obiecte (OO, *object-oriented*)**. **Declaratia de clasa defineste un tip complex care combina date si functiile care actioneaza asupra acelor date**.

**Variabilele de tip clasa se numesc obiecte**. Cum Java e un limbaj OO pur, tot codul Java trebuie declarat in interiorul unor clase.

```
2 public static void main(String[] args) {
```

**Linia 2 declara o metoda `main()`**, al carei cod este neprotejat si deschis tuturor celorlalte clase (cf. `public`), o metoda **cu caracter global clasei** (cf. `static`) si **care nu returneaza nici o valoare** (cf. `void`). Metoda `main()` este numita ***punct de intrare in program***, si reprezinta metoda care va fi executata prima, atunci cand va fi lansata interpretarea clasei `Salut`.

Metoda `main()` **primeste ca parametru**, in interiorul operatorului listei de parametri ai metodelor, **un tablou de obiecte de tip `string`**. Operatorul de indexare este folosit pentru a declara tabloul. Prin intermediul acestui tablou, **interpretorul Java paseaza argumentele adaugate de utilizator dupa numele interpretorului (*java*) si al programului** (clasei, in cazul nostru, `Salut`). Programul poate utiliza sau nu aceste argumente, pe care le acceseaza prin intermediul referintei `args` (**referinta la tablou de obiecte de tip `string`**).

`string` este numele unei clase din **biblioteca standard Java**, numita `java`, din **pachetul de clase care sunt implicit importate**, numit `java.lang`. **Numele sau complet (calificat de numele pachetului) este `java.lang.String`**. Rolul clasei `String` este de a reprezenta (incapsula) siruri de caractere Java (in Java caracterele sunt reprezentate in **format UNICODE**, in care **fiecare caracter necesita 2 octeti pentru codificare**). Clasa `String` permite reprezentarea sirurilor de caractere **nemodificabile (*immutable*)**.

Dupa declaratia metodei, urmeaza corpul ei, aflat intre acolade.

```
3 System.out.println("Buna ziua!");
```

**Linia 3 reprezinta corpul metodei `main()`**. Ea reprezinta o **instructiune Java de tip invocare de metoda** (apel de functie). Este invocata metoda `println()` pentru a se trimite pe ecran (in consola standard de iesire) un sir de caractere. Metoda `println()` **apartine obiectului `out`** (de tip `java.io.PrintStream`), care este **atribut cu caracter de clasa al clasei `System`** (de fapt, `java.lang.System`). Obiectul `out` **corespunde consolei standard de iesire**, la fel cum obiectul `in` corespunde consolei standard de intrare, iar obiectul `err` corespunde consolei standard pentru mesaje de eroare (vezi si figura urmatoare). Operatorul de calificare a numelor este folosit pentru a se specifica numele calificat al metodei `println()`. Sirul de caractere care ii este pasat ca parametru (**Buna ziua!**) este plasat in operatorul de declarare a sirurilor de caractere. Metoda `println()` nu returneaza nici o valoare. Instructiunea se incheie cu operatorul de sfarsit de instructiune.

**Linia 4 din program precizeaza incheierea declaratiei metodei `main()`**.

**Linia 5 din program precizeaza incheierea declaratiei clasei `salut` (liniile 3, 4 si 5 din formeaza corpul clasei `salut`) si prin urmare incheierea programului**.

---

## 2.2.4. Exemplificarea etapelor dezvoltarii programelor Java

Sa ilustram acum utilizarea sistemului de dezvoltare Java pentru programul dat.

Prima operatie este **editarea** programului intr-un **editor de text**, de exemplu, sub Windows, *Notepad.exe*. **Fisierul**, care va contine cele 5 linii de text ale programului, **trebuie** salvat cu numele *Salut.java*.

Pentru **obtinerea codului de octeti (bytecode) Java**, codul din acest fisier trebuie **compilat**. Daca se presupune utilizarea compilatorului Java din linia de comanda (consola standard de intrare), atunci trebuie executata urmatoarea **comanda**, in directorul in care se afla fisierul *Salut.java*:

```
directorcurent> javac Salut.java
```

In urma acestei comenzi, **compilatorul** Java va crea **genera codul de octeti** corespunzator intr-un fisier cu numele *Salut.class*, in directorul curent (acelasi director in care se afla si fisierul *Salut.java*).

Pentru **executia programului**, acesta trebuie **lansat in interpretor** (de fapt, interpretorul e lansat in executie, iar programul Java e interpretat), folosind comanda:

```
directorcurent> java Salut
```

**Rezultatul va fi** aparitia mesajului **Buna ziua!** pe ecranul monitorului, in fereastra linie de comanda (consola standard de iesire), o linie sub comanda de executie, urmata de aparitia cursorului (in cazul nostrum: *directorcurent>* ) pe linia urmatoare.

In final, pe ecran poate fi urmatoarea secventa de informatii:

```
directorcurent> javac Salut.java
directorcurent> java Salut
Buna ziua!
directorcurent>
```

---

## 2.3. Elementele de baza ale limbajului Java

### 2.3.1. Comentarea codului

Java suporta **trei tipuri de delimitatori pentru comentarii** - traditionalul `/*` si `*/` din C, `//` din C++, si o noua varianta, care incepe cu `/**` si se termina cu `*/`.

Delimitatorii `/*` si `*/` sunt utilizati pentru a separa textul care trebuie tratat ca un comentariu de catre compilator. Acesti delimitatori sunt folositori cand vreti sa comentati o portiune mare (mai multe linii) de cod, ca mai jos:

```
/* Acesta este un comentariu care va separa
   mai multe linii de cod. */
```

Delimitatorul de comentariu `//` este imprumutat din C++ si este folosit pentru a indica ca restul liniei trebuie tratat ca un comentariu de catre compilatorul Java. Acest tip de delimitator de comentariu este folositor mai ales pentru a adauga comentarii adiacente liniilor de cod, cum se arata mai jos:

```
Date astazi = new Date(); // creaza un obiect data initializat cu data de azi
System.out.println(astazi); // afiseaza data
```

Delimitatorii `/**` si `*/` sunt noi, apar pentru prima data in Java, si sunt folositi pentru a arata ca textul trebuie tratat ca un comentariu de catre compilator, dar de asemenea ca textul este parte din documentatia clasei care poate fi generata folosind JavaDoc. Acesti delimitatori pot fi folositi pentru a incadra linii multiple de text, in acelasi mod in care s-a aratat ca se face cu `/*` si `*/`, dupa cum urmeaza:

```
/** Clasa NeuralNetwork implementeaza o retea back-propagation
    si ... */
```

Delimitatorii de comentariu din Java sunt enumerati in tabelul 2.3.1.

**Tabelul 2.3.1. Delimitatorii de comentariu din Java**

<i>Inceput</i>	<i>Sfarsit</i>	<i>Scop</i>
<code>/*</code>	<code>*/</code>	Textul continut este tratat ca un comentariu.
<code>//</code>	(nimic)	Restul liniei este tratata ca un comentariu.
<code>/**</code>	<code>*/</code>	Textul continut este tratat ca un comentariu de catre compilator, si <i>poate folosit de catre JavaDoc pentru a genera automat documentatie.</i>

### 2.3.2. Cuvintele cheie Java

In tabelul 2.3.2 este prezentata **lista cuvintelor cheie** din Java. In plus, specificatia limbajului Java **rezerva cuvinte cheie** aditionale **care vor fi folosite in viitor**. Cuvintele cheie Java nefolosite sunt prezentate in tabelul 2.3.3.

Tabelul 2.3.2. Cuvinte cheie Java

abstract (OO)	finally ( <b>exceptii</b> )	public (OO)
boolean	<b>float</b>	<b>Return</b>
<b>break</b>	<b>for</b>	<b>Short</b>
byte	<b>if</b>	<b>Static</b>
<b>case</b>	implements (OO)	super (OO)
catch ( <b>exceptii</b> )	import	<b>Switch</b>
<b>char</b>	instanceof (OO)	Synchronized
class (OO)	<b>int</b>	this (OO)
<b>continue</b>	interface (OO)	throw ( <b>exceptii</b> )
<b>default</b>	<b>long</b>	throws ( <b>exceptii</b> )
<b>do</b>	native	transient
<b>double</b>	new (OO)	try ( <b>exceptii</b> )
<b>else</b>	package	<b>void</b>
extends (OO)	private (OO)	<b>volatile</b>
final (OO)	protected (OO)	<b>while</b>
strictfp ( <i>din Java2</i> )		

OO = tine de orientarea spre obiecte, **exceptii** = tine de tratarea exceptiilor, **bold** = existent si in limbajul C

Tabelul 2.3.3. Cuvinte cheie Java rezervate pentru utilizare viitoare

const	goto	
-------	------	--

Tabelul 2.3.4. Alte cuvinte Java rezervate

false	null	true
-------	------	------

### 2.3.3. Tipurile primitive Java

Tipurile primitive sunt **caramizile cu care se construiesc partea de date a unui limbaj**. Asa cum materia se compune din atomi legati impreuna, tipurile de date complexe se construiesc prin combinarea tipurilor primitive ale limbajului. Java contine un set mic de tipuri de date primitive: intregi, in virgula mobila, caracter si booleane.

In Java, ca si in C sau C++, **o variabila se declara prin tipul ei urmat de nume**, ca in urmatoarele exemple:

```
int x;
float LifeRaft;
short people;
long TimeNoSee;
double amountDue, amountPaid;
```

In codul de mai sus, x este declarat ca intreg, LifeRaft este declarat ca o variabila cu valori in virgula mobila, people este declarat ca intreg scurt, TimeNoSee este declarat ca intreg lung, iar amountDue si amountPaid sunt declarate ca variabile in dubla precizie, cu valori in virgula mobila.

#### 2.3.3.1. Tipuri intregi

Java ofera **patru tipuri de intregi**: byte, short, int, si long, care sunt definite ca **valori cu semn reprezentate pe 8, 16, 32, si 64 biti** cum se arata in tabelul 2.3.5. **Operatiile care se pot aplica primitivelor intregi** sunt enumerate in tabelul 2.3.6.

Tabelul 2.3.5. Tipurile întregi primitive din Java.

Tip	Dimensiune in biti	Dimensiune in octeti	Valoare minima	Valoare maxima
<b>byte</b>	8	1	-256	255
<b>short</b>	16	2	-32,768	32,767
<b>int</b>	32	4	-2,147,483,648	2,147,483,647
<b>long</b>	64	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Tabelul 2.3.6. Operatori pentru tipuri întregi primitive.

Operator	Operatie
=	Egalitate
!=	Inegalitate
>	Mai mare decat
<	Mai mic decat
>=	Mai mare sau egal cu
<=	Mai mic sau egal cu
+	Adunare
-	Scadere
*	Inmultire
/	Impartire
%	Modul
++	Incrementare
--	Decrementare
~	Negare logica pe biti
&	SI logic
	SAU logic
^	XOR logic
<<	Deplasare la stanga
>>	Deplasare la dreapta
>>>	Deplasare la dreapta cu completare cu zero

Daca amandoi operanzii sunt de tipul `long`, atunci rezultatul va fi un `long` pe 64 de biti. Daca unul din operanzi nu este `long`, el va fi transformat automat intr-un `long` inaintea operatiei. Daca nici un operand nu este `long`, atunci operatia se va face cu precizia pe 32 de biti a unui `int`. Orice operand `byte` sau `short` va fi transformat intr-un `int` inaintea operatiei.

### 2.3.3.2. Tipuri in virgula mobila

Suportul pentru numere in virgula mobila in Java este asigurat prin intermediul a  **doua tipuri primitive**: `float` si `double`, care sunt valori pe 32 si 64 de biti. Operatorii disponibili pentru folosirea cu aceste primitive sunt prezentati in Tabelul 2.3.7.

Numerele in virgula mobila din Java respecta specificatia IEEE Standard 754. Variabilele Java de tipul `float` si `double` pot fi transformate in alte tipuri numerice, dar nu pot fi transformate in tipul `boolean`.

Tabelul 2.3.7. Operatori pentru tipuri in virgula mobila primitive.

<i>Operator</i>	<i>Operatie</i>
=	Egalitate
!=	Inegalitate
>	Mai mare decat
<	Mai mic decat
>=	Mai mare sau egal cu
<=	Mai mic sau egal cu
+	Adunare
-	Scadere
*	Inmultire
/	Impartire
%	Modul
++	Incrementare
--	Decrementare

Daca amandoi operanzi sunt de un tip in virgula mobila, operatia se considera a fi o operatie in virgula mobila. Daca unul din operanzi este `double`, toti vor fi tratati ca `double` si se fac automat transformarile necesare. Daca nici un operand nu este `double`, fiecare va fi tratat ca `float` si transformat dupa necesitati.

**Numerele in virgula mobila pot avea urmatoarele valori speciale:** minus infinit, valori finite negative, zero negativ, zero pozitiv, valori finite pozitive, plus infinit, NaN ("*not a number* = nu este numar").

Aceasta valoare, NaN, este folosita pentru a indica valori care nu se incadreaza in scala de la minus infinit la plus infinit. De exemplu operatia de mai jos va produce NaN:

```
0.0f / 0.0f
```

Faptul ca NaN este gandit ca o valoare in virgula mobila poate provoca efecte neobisnuite cand valori in virgula mobila sunt comparate cu operatori relationali. Pentru ca NaN nu se incadreaza in scala de la minus infinit la plus infinit, rezultatul compararii cu el va fi `false`. De exemplu, `5.3f > NaN` si `5.3f < NaN` sunt `false`. De fapt, cand NaN este comparat cu el insusi cu `==`, rezultatul este `false`.

### 2.3.3.3. Alte tipuri primitive

In plus fata de tipurile intregi si tipurile in virgula mobila, Java include **inca doua tipuri primitive** boolean si caracter. Variabilele de tipul `boolean` pot lua valorile `true` sau `false`, in timp ce variabilele de tipul `char` pot lua valoarea unui caracter Unicode.

### 2.3.3.4. Valori predefinite

Una dintre sursele obisnuite ale erorilor de programare este folosirea unei variabile neinitializate. In mod obisnuit, acest *bug* apare in programele care se comporta aleator.

Cateodata aceste programe fac ce se presupune ca ar trebui sa faca; in alte dati produc efecte nedorite. Astfel de lucruri se petrec pentru ca o variabila neinitializata poate lua valoarea vreunui obiect uitat alocat aflat in locatia de memorie in care programul ruleaza. Java previne acest tip de probleme prin **desemnarea unei valori predefinite fiecărei variabile neinitializate**. Valorile predefinite sunt date in functie de tipul variabilei, cum se arata in Tabelul 2.3.8.

**Tabelul 2.3.8. Valori predefinite standard pentru tipurile primitive din Java.**

<i>Primitiva</i>	<i>Valoare predefinita</i>
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	null
boolean	false
toate referintele	null

### 2.3.3.5. Conversii intre tipurile primitive

In Java, se poate *converti explicit* o variabila de un tip in alt tip ca mai jos:

```
float fRunsScored = 3.2f;
int iRunsScored = (int) fRunsScored;
```

In acest caz, valoarea in virgula mobila 3.2, care este pastrata in `fRunsScored` va fi transformata intr-un intreg si pusa in `iRunsScored`. Cand se transforma in intreg, partea fractionara a lui `fRunsScored` va fi trunchiata asa incat `iRunsScored` va fi egal cu 3.

Acesta este un exemplu a ceea ce se numeste *conversie prin trunchiere*. O conversie prin trunchiere poate pierde informatii despre amplitudinea sau precizia unei valori, cum s-a vazut in acest caz. Trebuie intotdeauna avuta grija cand se scriu conversii prin trunchiere din cauza potentialului ridicat de risc de a pierde date.

Celalalt tip de conversie se numeste *conversie prin extindere*. O conversie realizata prin extindere poate duce la pierdere de precizie, dar nu va pierde informatii despre amplitudinea valorii. In general, conversiile realizate prin extindere sunt mai sigure. Tabelul 2.3.9 prezinta conversiile realizate prin extindere care sunt posibile intre tipurile primitive din Java.

**Tabelul 2.3.9. Conversiile realizate prin extindere intre tipurile primitive disponibile in Java.**

<i>Din</i>	<i>In</i>
byte	short, int, long, float, <b>or</b> double
short	int, long, float, <b>or</b> double
char	int, long, float, <b>or</b> double
int	long, float, <b>or</b> double
long	float <b>or</b> double
float	double



## 2.3.4. Variabile de tip primitiv

### 1. Declararea variabilelor de tip primitiv

Formatul pentru **declararea variabilelor de tip primitiv** este urmatorul:

```
<tipPrimitiv> <numeVariabilaTipPrimitiv>;
```

unde **tipPrimitiv** poate fi `byte`, `short`, `int`, `long`, `float`, `double`, `char`, sau `boolean`.

### 2. Alocarea si initializarea variabilelor de tip primitiv

Odata cu declararea, variabilelor de tip primitiv Java li se alocă spațiul de memorie necesar (1B pentru `byte`, 2B pentru `short`, 4B pentru `int`, 8B pentru `long`, 4B pentru `float`, 8B pentru `double`, 2B pentru `char`, și 1b pentru `boolean`).

Tot odata cu declararea, variabilele de tip primitiv Java sunt initializate in mod implicit (`byte`, `short`, `int`, `long`, `float` și `double` cu 0, `char` cu `null`, și `boolean` cu `false`).

Este posibilă și **initializarea explicită a variabilelor de tip primitiv**, folosind urmatorul format:

```
<tipPrimitiv> <numeVariabilaTipPrimitiv> = <valoareInitiala>;
```

Dupa declarare (care include alocare și initializare, implicită sau explicită), spațiul alocat variabilei arată astfel:

```
numeVariabilaTipPrimitiv valoareInitiala  
                        (variabila de tip primitiv)
```

### 3. Accesul la variabile de tip primitiv

Dupa declarare, variabilelor de tip primitiv li se pot da valori, operație numită scriere sau **atribuire a unei valori**:

```
<numeVariabilaTipPrimitiv> = <expresie>;
```

De asemenea, după declarare, valorile variabilelor de tip primitiv pot fi citite, **valorile lor pot fi folosite**, ca parametri în apelul unor funcții, ca termeni ai unor expresii, s.a.m.d.

**Scrierea** (atribuirea unei valori) și **citirea** (folosirea valorii pe care o conține) unei variabile se numesc colectiv **acces** la acea variabilă.

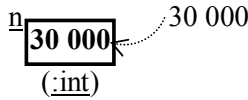
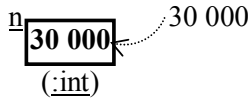
Pentru exemplificare vom folosi următoarele linii de cod Java:

```
1    int n;           // declarare (implicit alocare si initializare cu 0)
2    n = 30000;       // atribuire
3    int m = 10;      // declarare (implicit alocare) si initializare explicita
4    m = n;           // atribuire (m) si folosire a unei valori (n)
```

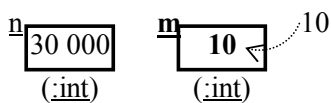
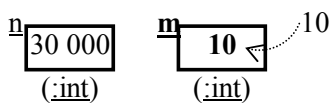
In linia 1 se declara o variabila, **n** (al carui nume este scris subliniat in reprezentarea grafica), de tip **int** (al carui nume este precedat de simbolul “:” si este scris subliniat in reprezentarea grafica). In mod implicit variabila este initializata cu valoarea **0**.

Dupa: `1 int n;` rezulta: 
  

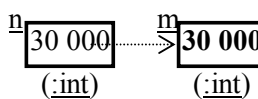
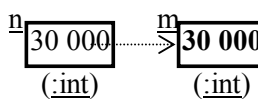

In linia 2 se atribuie variabilei **n** valoarea **30000**.

Dupa: `2 n = 30000;` rezulta: 
  


In linia 3 se declara o variabila, **m**, de tip **int**, initializata explicit cu valoarea **10**.

Dupa: `3 int m = 10;` rezulta: 
  


In linia 4 se foloseste valoarea variabilei **n** (**30000**) pentru a se atribui variabilei **m**.

Dupa: `4 m = n;` rezulta: 
  


### 2.3.5. Valorile literale Java

O valoare literala este o **valoare explicita care este folosita intr-un program**. De exemplu, programul poate include valoarea literala 3.1415 care se foloseste mereu cand este necesara valoarea lui *pi*, sau poate folosi 65 ca varsta legala de pensionare. Aceste valori, 3.1415 si 65, sunt amandoua valori literale.

#### 2.3.5.1. Valori literale intregi

Valorile literale intregi pot fi specificate in **notatie zecimala, hexazecimala sau octala**. Pentru a specifica o valoare zecimala, se foloseste numarul in mod normal. Pentru a arata ca o valoare literala este un **long**, se poate adauga "L" sau "l" la sfarsitul numarului. Valorile **hexazecimale** sunt date in baza 16 care include digitii 0-9 si literele A-F. Pentru a specifica o valoare hexazecimala, se foloseste **0x** urmat de digitii sau literele care formeaza valoarea. Similar, o valoare **octala** este identificata de un simbol **0**. Pentru **exemple in privinta folosirii valorilor intregi literale**, consultati Tabelul 2.3.10.

Tabelul 2.3.10. Exemple de valori intregi literale.

<i>Integer</i>	<i>Long</i>	<i>Octal</i>	<i>Hexadecimal</i>
0	0L	0	0x0
1	1L	01	0x1
10	10L	012	0xA
15	15L	017	0xF
16	16L	020	0x10
100	100L	0144	0x64

### 2.3.5.2. Valori literale in virgula mobila

Valorile literale in virgula mobila din Java sunt similare cu valorile literale intregi. Valorile literale in virgula mobila pot fi specificati atat in **notatia zecimala familiara** (de exemplu, 3.1415) sau in **notatia exponentiala** (de exemplu, 6.02e23). Pentru a arata ca o valoare literala trebuie tratata ca un *float* cu simpla precizie, i se ataseaza un "f" sau "F". Pentru a arata ca trebuie tratat in dubla precizie (*double*), i se adauga un "d" sau "D".

Java include *constantele predefinite*, POSITIVE\_INFINITY, NEGATIVE\_INFINITY, si NaN, pentru a reprezenta *infinitul si valorile care nu sunt numere*.

Lista care urmeaza arata **cateva valori literale in virgula mobila valide in Java**:

```
43.3F
3.1415d
-12.123f
6.02e+23f
6.02e23d
6.02e-23f
6.02e23d
```

### 2.3.5.3. Valori literale booleene

Java suporta doua valori literale *booleene* true si false.

### 2.3.5.4. Valori literale caracter

O valoare literala caracter este *un singur caracter sau o secventa escape inchisa in apostroafe*, de exemplu, 'b'. **Secventele escape** sunt folosite *pentru a inlocui caractere speciale sau actiuni*, cum ar fi linie noua, forma noua, sau retur de car. Secventele escape disponibile sunt prezentate in Tabelul 2.3.11. Iata cateva exemple de secvente *escape*:

```
'\b'
'\n'
'\u15e'
'\t'
```

Tabelul 2.3.11. Secvente *escape*.

Secventa	Utilizare
\b	Backspace
\t	Tab orizontal
\n	Line feed
\f	Form feed
\r	Carriage return
\"	Ghilimele
\'	Apostrof
\\	Backslash
\uxxxx	Caracter Unicode numarul xxxx

### 2.3.5.5. Valori literale siruri de caractere

Deși **nu exista tipul primitiv sir de caractere in Java (!)**, in programe se pot include valori literale de tipul sir de caractere. Majoritatea aplicatiilor si appleturilor folosesc o forma de valori literale sir de caractere, cel puțin pentru stocarea mesajelor de eroare. O **valoare literală sir de caractere** este format din **zero sau mai multe caractere** (incluzand secventele escape din Tabelul 10) **inchise între ghilimele**. Ca exemple de valori literale sir de caractere fie urmatoarele:

```
"Un sir"
"Coloana 1\tColoana 2"
"Prima linie\r\nA doua linie"
"Prima pagina\fA doua pagina"
""
```

Deoarece Java nu are tipul primitiv sir de caractere, **fiecare folosire a unei valori literale sir de caractere determina crearea automata a unui obiect din clasa string (!)**. Oricum, din cauza managementului automat al memoriei din Java, programul nu trebuie sa faca nimic anume pentru a elibera memoria utilizata de valoarea literală sau de sir, odata terminat lucrul cu el.

### 2.3.6. Operatori Java

Operatorii unui limbaj **se folosesc pentru a combina sau a schimba valorile din program in cadrul unei expresii**. Java contine un set foarte bogat de operatori. Iata lista completa a operatorilor din Java:

Tabelul 2.3.12. Lista completa a operatorilor din Java

=	>	<	!	~
?	:	==	<=	>=
!=	&&		++	--
+	-	*	/	&
	^	%	<<	>>
>>>	+=	-=	*=	/=
&=	=	^=	%=	<<=
>>=	>>>=			

#### 2.3.6.1. Operatori pe valori intregi

Majoritatea operatorilor din Java lucreaza cu valori intregi. Operatorii **binari** (cei care **necesita doi operanzi**) sunt prezentati in Tabelul 2.3.13. Operatorii **unari** (care **necesita un singur operand**) in Tabelul 2.3.14. Fiecare tabel ofera cate un exemplu de utilizare pentru fiecare operator.

Tabelul 2.3.13. Operatori binari pe intregi.

Operator	Operatie	Exemplu
=	Atribuire	a = b
==	Egalitate	a == b
!=	Inegalitate	a != b
<	Mai mic decat	a < b
<=	Mai mic sau egal cu	a <= b
>=	Mai mare sau egal cu	a >= b
>	Mai mare decat	a > b
+	Adunare	a + b
-	Scadere	a - b
*	Inmultire	a * b
/	Impartire	a / b
%	Modul	a % b
<<	Deplasare la stanga	a << b
>>	Deplasare la dreapta	a >> b
>>>	Deplasare la dreapta cu umplere cu zero	a >>> b
&	SI pe biti	a & b
	SAU pe biti	a   b
^	XOR pe biti	a ^ b

Tabelul 2.3.14. Operatori unari pe intregi.

Operator	Operatie	Exemplu
-	Negare unara	-a
~	Negare logica pe biti	~a
++	Incrementare	a++ sau ++a
--	Decrementare	a-- sau --a

In plus fata de operatorii din tabelele 2.3.13 si 2.3.14, Java include si un tip de operatori de atribuire bazati pe alti operatori. Acestia vor opera pe un operand si vor stoca rezultatul in acelasi operand. De exemplu, pentru a mari valoarea unei variabile x, puteti face urmatoarele:

```
x += 3;
```

Aceasta este *identic cu* formula mai explicita  $x = x + 3$ . Fiecare operator specializat de atribuire din Java aplica functia sa normala pe un operand si pastreaza rezultatul in acelasi operand. Urmatorii operatori de atribuire sunt disponibili:

Tabelul 2.3.15. Operatori de atribuire pentru intregi

+=	-=	*=
/=	&=	=
^=	%=	<<=
>>=	>>>=	

### 2.3.6.2. Operatori pe valori in virgula mobila

Operatorii Java pe valori in virgula mobila sunt un subset al celor disponibili pentru intregi. Operatorii care *pot opera pe operanzi de tipul float sau double* sunt prezentati in Tabelul 2.3.16, unde sunt date si exemple de utilizare.

Tabelul 2.3.16. Operatori binari pe intregi.

<i>Operator</i>	<i>Operatie</i>	<i>Exemplu</i>
=	Atribuire	a = b
==	Egalitate	a == b
!=	Inegalitate	a != b
<	Mai mic decat	a < b
<=	Mai mic sau egal cu	a <= b
>=	Mai mare sau egal cu	a >= b
>	Mai mare decat	a > b
+	Adunare	a + b
-	Scadere	a - b
*	Inmultire	a * b
/	Impartire	a / b
%	Modul	a % b
-	Negare unara	-a
++	Incrementare	a++ sau ++a
--	Decrementare	a-- sau --a

### 2.3.6.3. Operatori pe valori booleane

Operatorii din Java pentru valori booleene sunt cuprinsi in Tabelul 2.3.17. Daca ati programat in C sau C++ inainte de a descoperii Java, sunteti probabil deja familiar cu ei. Daca nu operatorii conditionali vor probabil o experienta noua.

Tabelul 2.3.17. Operatori pe valori booleene.

<i>Operator</i>	<i>Operatie</i>	<i>Exemplu</i>
!	Negare	!a
&&	SI conditional	a && b
	SAU conditional	a    b
==	Egalitate	a == b
!=	Inegalitate	a != b
?:	Conditional	a ? expr1 : expr2

**Operatorul conditional** este singurul operator **ternar** (*opereaza asupra a trei operanzi*) din Java si are urmatoarea forma sintactica:

```
<expresieBooleana> ? <expresie1> : <expresie2>
```

Valoarea lui `booleanExpr` este evaluata si daca este `true`, expresia `expr1` este executata; daca este `false`, este executata expresia `expr2`. Aceasta face din operatorul conditional o *scurtatura pentru*:

```
if (<expresieBooleana>)
    <expresie1>
else
    <expresie2>
```

## 2.3.7. Instructiuni pentru controlul executiei programului Java

Cuvintele cheie pentru controlul executiei programului sunt aproape identice cu cele din C si C++. Aceasta este una dintre cele mai evidente cai prin care Java isi demonstreaza mostenirea dobandita de la cele doua limbaje. In aceasta sectiune, veti invata sa folositi instructiunile din Java pentru a scrie metode.

Limbajul Java ofera  *doua structuri alternative*  - instructiunea `if` si instructiunea `switch` - pentru a selecta dintre mai multe alternative. Fiecare are avantajele sale.

### 2.3.7.1. Instructiunea `if`

Instructiunea `if` din Java *testeaza o expresie booleana*.

- *Daca* expresia booleana *este evaluata ca fiind true*, instructiunile care urmeaza dupa `if` sunt executate.
- *Daca* expresia booleana *este false*, instructiunile de dupa `if` nu se executa.

De exemplu, urmariti urmatorul fragment de cod:

```
1 import java.util.Date;
2 // ...
3 Date today = new Date();
4 if (today.getDay() == 0)
5     System.out.println("Este duminica.");
```

Acest cod foloseste pachetul `java.util.Date` si creaza o variabila numita `today` in care se pastreaza data curenta. Functia membru `getDay()` este apoi aplicata lui `today` si rezultatul este comparat cu 0. O valoare rezultata 0 pentru `getDay` arata ca este duminica, adica expresia booleana `today.getDay() == 0` este `true`, si un mesaj este afisat. Daca astazi nu e duminica nu se intampla nimic.

Daca aveti experienta in C sau C++, **veti fi tentati sa rescrieti exemplul precedent astfel:**

```
1 Date today = new Date();
2 if (!today.getDay())
3     System.out.println("Este duminica.");
```

In C and C++, expresia `!today.getDay()` va fi evaluata 1 daca expresia `today.getDay` este 0 (indicand duminica). In Java, expresiile folosite intr-o instructiune `if` trebuie sa fie evaluate la un boolean. Din aceasta cauza, acest cod nu va functiona, deoarece `!today.getDay` va fi evaluata la 0 sau 1, in functie de ce zi a saptamanii este. Dar valorile intregi nu pot fi transformate explicit in valori booleane.

In Java exista si o instructiune `else` care poate fi executata cand expresia testata de `if` este evaluata ca fiind `false`, ca in urmatorul exemplu:

```
1 Date today = new Date();
2 if (today.getDay() == 0)
3     System.out.println("Este duminica.");
4 else
5     System.out.println("Nu este duminica.");
```

In acest caz, un mesaj va fi afisat daca este duminica, iar daca nu este duminica va fi afisat celalalt mesaj. In cele doua exemple de pana acum se executa o singura instructiune dupa `if` sau in `else`. Inchizand instructiunile intre acolade, se pot executa oricate linii de cod. Acest lucru este demonstrat in urmatorul exemplu:

```
1   Date today = new Date();
2   if (today.getDay() == 0) {
3       System.out.println("Este duminica.");
4       System.out.println("O zi buna pentru tenis.");
5   }
6   else {
7       System.out.println("Nu este duminica.");
8       System.out.println("Ziua este de asemenea buna pentru tenis.");
9   }
```

Pentru ca este posibil sa se execute orice cod se doreste in portiunea `else` a unui **bloc `if...else`**, este posibil sa se execute alta instructiune `if` in interiorul instructiunii `else` din prima instructiune `if`. Aceasta este cunoscut in general sub numele de **bloc `if...else if...else`**. Iata un exemplu:

```
1   Date today = new Date();
2   if (today.getDay() == 0)
3       System.out.println("Este duminica.");
4   else if (today.getDay() == 1)
5       System.out.println("Este luni.");
6   else if (today.getDay() == 2)
7       System.out.println("Este marti.");
8   else if (today.getDay() == 3)
9       System.out.println("Este miercuri.");
10  else if (today.getDay() == 4)
11      System.out.println("Este joi.");
12  else if (today.getDay() == 5)
13      System.out.println("Este vineri.");
14  else
15      System.out.println("Este sambata.");
```

### 2.3.7.2. Instructiunea `switch`

Dupa cum se observa din exemplul precedent, o serie lunga de instructiuni `if...else if...else` pot fi alaturate si codul devine din ce in ce mai greu de citit. Aceasta problema se poate evita folosind instructiunea Java `switch`. Ca si in C si C++, **instructiunea `switch`** din Java este ideala pentru **compararea unei singure expresii cu o serie de valori si executarea codului asociat cu valoarea unde se gaseste egalitate**, adica executarea codului ce urmeaza instructiunea `case` care se potriveste:

```
1   Date today = new Date();
2   switch (today.getDay()) {
3       case 0: // duminica
4           System.out.println("Este duminica.");
5           break;
6       case 1: // luni
7           System.out.println("Este luni.");
8           break;
9       case 2: // marti
10          System.out.println("Este marti.");
11          break;
12       case 3: // miercuri
13          System.out.println("Este miercuri.");
14          break;
15       case 4: // joi
16          System.out.println("Este joi.");
17          break;
18       case 5: // vineri
19          System.out.println("Este vineri.");
20          System.out.println("Weekend placut!");
21          break;
22       default: // sambata
23          System.out.println("Este sambata.");
24   }
```



Se observa ca fiecare zi are propria **ramura case** in interiorul lui `switch`. Ramura zilei sambata (unde `today.getDay() = 6`) nu este data explicit, ci de ea se ocupa **instructiunea default**. Fiecare bloc `switch` poate include **optional** un `default` care *se ocupa de valorile netratate explicit* de un `case`.

In interiorul fiecarui `case`, pot fi mai multe linii de cod. Blocul de cod care se va executa de exemplu pentru ramura `case` a lui vineri, contine trei linii. Primele doua linii vor afisa mesaje, iar a treia este **instructiunea break**.

Cuvantul cheie `break` *se foloseste in interiorul unei instructiuni case pentru a indica programului sa execute urmatoarea linie care urmeaza blocului switch*. In acest exemplu, `break` determina programul in executie sa sara la linia care afiseaza "All done!" Instructiunea `break` nu a fost pusa si in blocul `default` pentru ca acolo oricum blocul `switch` se termina, si nu are sens sa folosim o comanda explicita pentru a iesi din `switch`.

**Putem sa nu includem break la sfarsitul fiecarui bloc case.** Sunt cazuri in care nu dorim sa iesim din `switch` dupa executarea codului unui anume `case`. De exemplu, sa consideram urmatorul exemplu, care ar putea fi folosit ca un sistem de planificare a timpului unui medic:

```
1    Date today = new Date();
2    switch (today.getDay()) {
3        case 0:        // duminica
4        case 3:        // miercuri
5        case 6:        // sambata
6            System.out.println("Zi de tenis!");
7            break;
8        case 2:        // marti
9            System.out.println("Inot la 8:00 am");
10       case 1:        // luni
11       case 4:        // joi
12       case 5:        // vineri
13           System.out.println("Program de lucru: 10:00 am - 5:00 pm");
14           break;
15    }
```

Acest exemplu ilustreaza cateva concepte cheie despre instructiunea `switch`. In primul rand, se observa ca *mai multe ramuri case uri pot executa aceeasi portiune de cod*:

```
1        case 0:        // duminica
2        case 3:        // miercuri
3        case 6:        // sambata
4            System.out.println("Zi de tenis!");
5            break;
```

Acest cod va afisa mesajul "Zi de tenis" daca ziua curenta este vineri, sambata sau duminica. Daca alaturam aceste trei ramuri `case` fara a interveni cu nici un `break`, fiecare va executa acelasi cod. Sa vedem ce se intimpla marti cand se executa urmatorul cod:

```
1        case 2:        // marti
2            System.out.println("Inot la 8:00 am");
```

Desigur va fi afisat mesajul, dar acest `case` nu se termina cu un `break`. **Deoarece codul** pentru marti *nu se termina cu un break, programul va continua sa execute codul din urmatoarele case pana intalneste un break*. Aceasta inseamna ca dupa ce se executa partea de cod de la marti, se va executa si codul de la luni, joi si vineri ca mai jos:

```
1        case 2:        // marti
2            System.out.println("Inot la 8:00 am");
3        case 1:        // luni
4        case 4:        // joi
5        case 5:        // vineri
6            System.out.println("Program de lucru: 10:00 - 5:00");
7            break;
```

Urmatoarele mesaje vor fi afisate in fiecare de marti:

Inot la 8:00 am

Program de lucru: 10:00 - 5:00

Luni, joi si vineri, numai ultimul mesaj va fi afisat.

In loc de a scrie instructiuni `switch` care folosesc ramuri `case` intregi, *se pot folosi si valori caracter* ca mai jos:

```
1  switch (aChar) {
2      case 'a':
3      case 'e':
4      case 'i':
5      case 'o':
6      case 'u':
7          System.out.println("Este o vocala!");
8          break;
9      default:
10         System.out.println("Este o consoana!");
11 }
```

### 2.3.7.3. Instructiunea `for`

Iteratiile sunt un concept important in programare. Fara a putea parcurge un set de valori una cate una, adica a le itera, posibilitatea de a rezolva multe din problemele lumii reale ar fi limitata. Instructiunile de iterare din Java sunt aproape identice cu cele din C si C++. Exista bucle `for`, bucle `while`, si bucle `do ... while`.

In prima linie a buclei `for` se specifica *valoarea de inceput a unui contor de bucla, conditia testata pentru iesirea din bucla* si se indica *cum trebuie incrementat contorul*. Aceasta instructiune ofera intr-adevar multe posibilitati.

**Sintaxa instructiunii `for`** in Java este prezentata in continuare:

```
for (<expresieInitializare>; <expresieTestata>; <expresieIncrementare>)
    <instructiuneExecutataRepetat> // cat timp <expresieTestata> ="true"
```

Un exemplu de bucla `for` ar putea fi urmatorul:

```
int count;
for (count=0; count<100; count++)
    System.out.println("Count = " + count);
```

In acest exemplu, in instructiunea de initializare a buclei `for` se seteaza contorul cu 0. Expresia testata, `count < 100`, arata ca bucla trebuie sa continue cat timp `count` este mai mic decat 100. In sfarsit, instructiunea de incrementare mareste valoarea lui `count` cu 1. Cat timp expresia de test este adevarata, instructiunea care urmeaza dupa `for` va fi executata, dupa cum urmeaza:

```
System.out.println("Count = " + count);
```

Pentru a face mai mult de o operatie in interiorul buclei se folosesc acolade pentru a incadra instructiunile de executat in bucla `for`:

```
int count;
for (count=0; count<100; count++) {
    YourMethod(count);
    System.out.println("Count = " + count);
}
```

Este posibil sa scriem *bucle for mai complicate* in Java, incuzand mai multe instructiuni sau conditii. De exemplu, in urmatoarul cod:

```
for (up = 0, down = 20; up < down; up++, down -= 2 ) {
    System.out.println("Up = " + up + "\tDown = " + down);
}
```

Aceasta bucla porneste cu variabila `up` de la 0 si o incrementeaza cu 1. De asemenea porneste cu variabila `down` de la 20 si o decrementeaza cu 2 in fiecare pas al buclei. Bucla continua pana cand `up` a fost incrementat destul incat este mai mare sau egal cu variabila `down`.

*Expresia testata* dintr-o bucla `for` poate fi *orice expresie booleana*. Din aceasta cauza, nu trebuie neaparat sa fie un test simplu ca (`x < 10`), din exemplele precedente. Expresia test *poate fi un apel la o metoda, un apel la o metoda combinat cu testarea unei valori*, sau orice poate fi exprimat printr-o expresie booleana.

### 2.3.7.4. Instructiunea while

Inrudita cu bucla `for` este bucla `while`. **Sintaxa buclei while** este urmatoarea:

```
while (<expresieBooleana>) // "true" indica repetarea
    <instructiuneExecutataRepetat>
```

Cum se vede din simplitatea acestei declaratii, bucla `while` din Java nu are suportul necesar pentru a initializa si a incrementa variabile cum are bucla `for`. Din aceasta cauza, trebuie acordata atentie initializarii contorilor in afara buclei si incrementarii lor in interiorul ei. De exemplu, urmatoarul cod va afisa un mesaj de cinci ori:

```
int count = 0;
while (count < 5) {
    System.out.println("Count = " + count);
    count++;
}
```

### 2.3.7.5. Instructiunea do...while

Ultima constructie pentru buclare din Java este bucla `do...while`. **Sintaxa pentru bucla do...while** este urmatoarea:

```
do {
    <instructiuneExecutataRepetat>
} while (<expresieBooleana>); // "true" indica repetarea
```

Aceasta este similara cu a buclei `while`, doar ca **buclea do...while se executa garantat cel putin o data**, pe cand o **buclea while este posibil sa nu se execute deloc**, totul depinzand de expresia de test folosita in bucla.

De exemplu, sa consideram urmatoarea metoda:

```
public void ShowYears(int year) {
    while (year < 2000) {
        System.out.println("Year is " + year);
        year++;
    }
}
```

Acestei metode ii este pasat un an `year`, iar ea afiseaza un mesaj cat timp anul este mai mic de 2000. Daca `year` incepe cu 1996, atunci vor fi afisate mesaje pentru anii 1996, 1997, 1998, and 1999.

Totusi, ce se intampla *daca year incepe de la 2010*? Din cauza testului initial, `year < 2000`, va fi `false`, iar **bucla `while` nu va fi executata niciodata**. Din fericire, o **bucla `do...while` poate rezolva aceasta problema**. Pentru ca o bucla `do...while` executa expresia test dupa ce executa corpul buclei pas cu pas, *ea va fi executata cel putin o data*.

Aceasta este o distinctie foarte clara intre cele doua tipuri de bucle, dar poate fi si o sursa de erori potentiala. Oricand se foloseste o bucla `do...while`, trebuie acordata atentie faptului ca in primul pas se executa corpul buclei.

### 2.3.7.6. Instructiunea `break`

Java usureaza *iesirea din bucle si controlul altor parti ale executiei programului* cu **instructiunile `break` si `continue`**.

Mai devreme in acest curs, am vazut cum instructiunea `break` este utilizata pentru a iesi dintr-o instructiune `switch`. In acelasi mod **instructiunea `break` poate fi folosit pentru a iesi dintr-o bucla**.

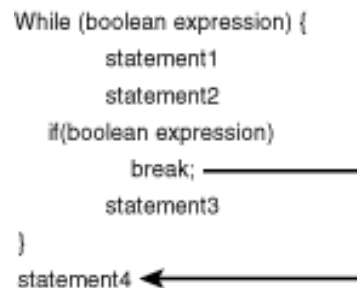


Figura 2.3.1: Controlul executiei programului cu instructiunea `break`.

Cum este ilustrat in Figura 2.3.1, daca o instructiune `break` este intalnita executia va continua cu `statement4`.

### 2.3.7.7. Instructiunea `continue`

Ca si instructiunea `break`, care poate fi folosita pentru a transfera executia programului imediat dupa sfarsitul unei bucle, **instructiunea `continue` poate fi folosita pentru a forta programul sa sara la inceputul buclei**.

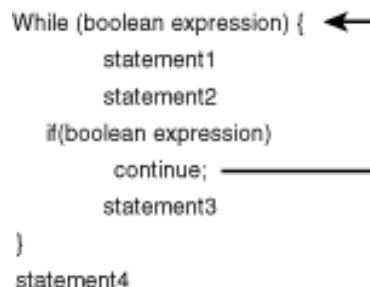


Figura 2.3.2 : Controlul executiei programului cu instructiunea `continue`.

### 2.3.7.8. Folosirea etichetelor

**Java nu include o instructiune goto.** Dar pentru ca `goto` este un cuvânt rezervat, el ar putea fi adaugat in versiunile viitoare. In loc de `goto`, **Java permite combinarea instructiunilor `break` si `continue` cu o eticheta.** Aceasta are un efect similar cu `goto eticheta`, adica permite programului sa-si repositioneze executia.

Pentru a intelege folosirea etichetelor cu `break` si `continue`, sa consideram urmatorul exemplu:

```
1 public void paint(Graphics g) {
2     int line=1;
3     outsideLoop:
4     for(int out=0; out<3; out++) {
5         g.drawString("out = " + out, 5, line * 20);
6         line++;
7         for(int inner=0; inner < 5; inner++) {
8             double randNum = Math.random();
9             g.drawString(Double.toString(randNum), 15, line * 20);
10            line++;
11            if (randNum < .10) {
12                g.drawString("break to outsideLoop", 25, line * 20);
13                line++;
14                break outsideLoop;
15            }
16            if (randNum < .60) {
17                g.drawString("continue to outsideLoop", 25, line * 20);
18                line++;
19                continue outsideLoop;
20            }
21        }
22    }
23    g.drawString("all done", 50, line * 20);
24 }
```

Acest exemplu include doua bucle. Prima bucla pe variabila `out`, cea de a doua pe variabila `inner`. Bucla exterioara a fost etichetata cu urmatoarea linie:

```
outsideLoop:
```

Aceasta instructiune denumeste **bucla exterioara**. Un numar aleator intre 0 si 1 este generat la fiecare iteratie prin bucla interioara. Acest numar este afisat pe ecran. Daca numarul aleator este mai mic decat 0.10 va fi executata instructiunea `break outsideLoop`.

**O instructiune `break` normala** in aceasta pozitie **ar fi facut ca programul sa sara din bucla interioara**. Dar pentru ca aceasta este o instructiune `break etichetata`, ea va face programul sa sara din bucla identificate de numele etichetei. In acest caz, controlul programului sare la linia care afiseaza "all done" deoarece aceasta este prima linie dupa `outsideLoop`.

Pe de alta parte, daca numarul aleator nu este mai mic decat 0.10, numarul este comparat cu 0.60. Daca este mai mic de atat se executa instructiunea `continue outsideLoop`. **O instructiune `continue` normala** intalnit acum **ar fi transferat executia programului la inceputul buclei interioare**. Dar pentru ca aceasta este o instructiune `continue etichetata`, executia va fi transferata la inceputul buclei identificate de numele etichetei.

## 2.4. Tipuri referinta

### 2.4.1. Introducere in tipuri referinta Java

Tipurile referinta Java sunt:

- tipul **tablou**,
- tipul **clasa** si
- tipul **interfata**.

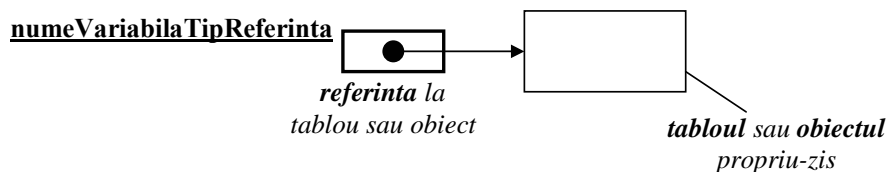
Variabilele de tip referinta sunt:

- variabile **tablou**, al caror **tip** este **un tablou**,
- variabile **obiect**, al caror **tip** este **o clasa sau o interfata**.

Variabilele de tip referinta **contin**:

- **referinta** catre tablou sau obiect (**creata in momentul declararii**),
- **tabloul sau obiectul** propriu-zis (**creat in mod dinamic, cu operatorul new**).

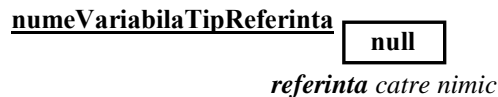
Notatie generala:



**Programatorul nu are acces la continutul referintelor** (ca in alte limbaje, cum ar fi C si C++, unde pointerii si referintele pot fi accesate si tratate ca orice alta variabila), **ci la continutul tablourilor sau al obiectelor referite**.

Pe de alta parte, **programatorul nu poate avea acces la continutul tablourilor sau al obiectelor decat prin intermediul referintelor** catre ele.

O valoare pe care o pot lua referintele este `null`, semnificand **referinta "catre nimic"**. **Simpla declarare a variabilelor referinta conduce la initializarea implicita a referintelor cu valoarea null**.



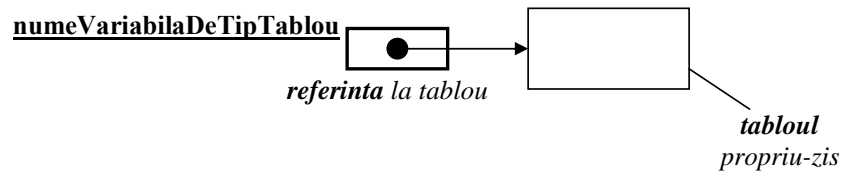
### 2.4.2. Tablouri cu elemente de tip primitiv

#### 2.4.2.1. Componentele interne ale unui tablou Java

Un **tablou Java** este o **structura care contine mai multe valori de acelasi tip**, numite **elemente**.

**Lungimea** unui tablou este **stabilita in momentul crearii tabloului**, in timpul executiei (*at runtime*). Dupa crearea dinamica a structurii, tabloul este o structura de dimensiune fixa. Asadar, lungimea nu poate fi modificata.

**Variabila tablou** este o **simplică referință la tablou**, creată în momentul declarării ei (moment în care poate fi inițializată implicit cu valoarea `null` – referință către nimic). Crearea dinamică a structurii se face utilizând operatorul `new`.



**Lungimea tabloului** poate fi **accesată prin intermediul variabilei membru** (interne) a tabloului care poartă numele `length`, de tip primitiv `int`.

**Elementele tabloului** (valorile încapsulate în structura tabloului) pot fi **accesate pe baza indexului** fiecărui element, care variază de la 0 la `length-1`.

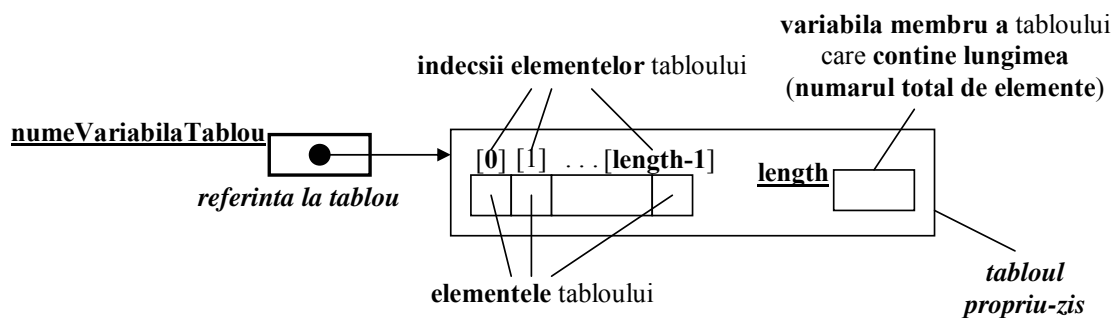


Fig. x. Componentele interne ale unui tablou Java

#### 2.4.2.2. Declararea variabilelor de tip tablou cu elemente de tip primitiv

**Tablourile sunt tipuri referință Java**, ceea ce înseamnă că tablourile sunt accesate prin intermediul unei locații numită **referință**, care **conține adresa tabloului propriu-zis**.

După simplă declarare a unei variabile de tip tablou (care **nu include alocare a memoriei pentru tabloul propriu-zis**), spațiul alocat variabilei arată astfel:

```
numeVariabilaTablou null
                    referinta
```

fiind alocat spațiu de memorie doar pentru locația referință, care e inițializată implicit cu `null` (referință **către nimic**, indicând **lipsa spațiului alocat pentru tablou**!).

În Java **tablourile se declară utilizând paranteze patrulate închise** (`[]`).

**Formatul pentru declararea variabilelor de tip tablou cu elemente de tip primitiv** este următorul:

```
tipPrimitiv[] numeTablouElementeTipPrimitiv;
```

unde `tipPrimitiv` poate fi `byte`, `short`, `int`, `long`, `float`, `double`, `char`, sau `boolean`.

Un **format alternativ** este cel folosit si de limbajele de programare C si C++:

```
tipPrimitiv numeTablouElementeTipPrimitiv[];
```

De exemplu, sa consideram urmatoarele **declaratii de tablouri**:

```
int intArray[];  
float floatArray[];  
double[] doubleArray;  
char charArray[];
```

Observati ca parantezele pot fi plasate inainte sau dupa numele variabilei. Plasand [] **dupa numele variabilei** se urmeaza **conventia din C**.

Exista insa un **avantaj in a plasa parantezele inaintea numelui variabilei**, folosind **formatul introdus de Java**, pentru ca **se pot declara mai usor tablouri multiple**. Ca de exemplu, in urmatoarele declaratii:

```
int[] firstArray, secondArray;  
int thirdArray[], justAnInt;
```

**In prima linie**, atat firstArray cat si secondArray sunt tablouri.

**In a doua linie**, thirdArray este un tablou, dar justAnInt este, dupa cum ii arata numele, un intreg. Posibilitatea de a declara variabile primitive si tablouri in aceeasi linie de program, ca in a doua linie din exemplul precedent, cauzeaza multe probleme in alte limbaje de programare. Java previne aparitia acestui tip de probleme oferind o sintaxa alternativa usoara pentru declararea tablourilor.

### 2.4.2.3. Alocarea si initializarea tablourilor cu elemente de tip primitiv

**Odata declarat, un tablou trebuie alocat (dinamic)**. Dimensiunea tablourilor nu a fost specificata in exemplele precedente. **In Java, toate tablourile trebuie alocate cu new**. Urmatoarea declaratie de tablou genereaza o eroare la compilare de genul:

```
int intArray[10]; // aceasta declaratie va produce eroare
```

Pentru a **aloca spatiul de memorie** necesar unui tablouri este utilizat **operatorul de generare dinamica new**, ca in urmatoarele exemple:

```
// stil C  
int intArray[] = new int[100]; // declaratie si definitie (initializare cu 0)  
float floatArray[]; // simpla declaratie  
floatArray = new float[100]; // definitie (si initializare implicita cu 0)  
  
// stil Java  
long[] longArray = new long[100]; // declaratie si definitie  
double[][] doubleArray = new double[10][10]; // tablou de tablouri
```

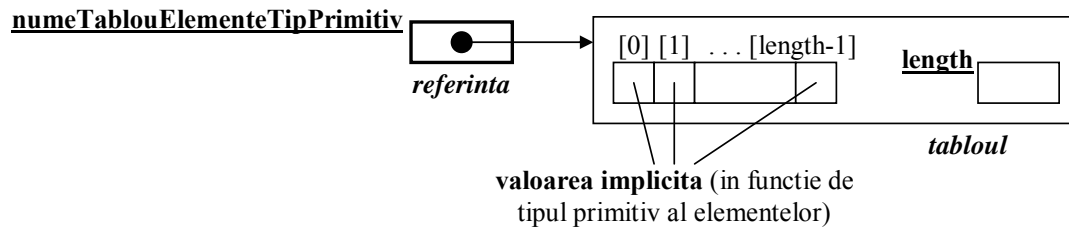
Operatorul **new** trebuie sa fie urmat de **initializarea lungimii tabloului (numele tipului elementelor tabloului urmat de paranteze drepte intre care se afla numarul de elemente al tabloului)**.

**Formatul pentru alocarea si initializarea lungimii variabilelor de tip tablou cu elemente de tip primitiv** este urmatorul:

```
numeTablouElementeTipPrimitiv = new tipPrimitiv[numarElementeTablou];
```



Dupa declarare (care include alocare si initializare, fie ea implicita sau explicita), spatiul alocat variabilei arata astfel:



O cale alternativa de a alocare si initializa un tablou in Java este sa se specifice o lista cu elementele tabloului initializate in momentul declararii tabloului, ca mai jos:

```
int intArray[] = {1,2,3,4,5};
char[] charArray = {'a', 'b', 'c'};
```

In acest caz, `intArray` va fi un tablou de cinci elemente care contine valorile de la 1 la 5. Tabloul de trei elemente `charArray` va contine caracterele 'a', 'b' si 'c'.

#### 2.4.2.4. Accesul la variabile de tip tablou cu elemente de tip primitiv

Dupa declararea, alocarea si initializarea variabilelor de tip tablou cu elemente de tip primitiv, acestea pot fi accesate, fie atribuind valori elementelor tabloului (scriind in tablou), fie folosind valorile elementelor tabloului (citind din tablou).

Sintaxa pentru obtinerea dimensiunii unui tablou este urmatoarea:

```
numeVariabilaTablou.length
```

De exemplu, pentru declaratiile de mai sus, `intArray.length` are valoarea 5, iar `charArray.length` are valoarea 3.

Tablourile in Java sunt numerotate de la 0 la numarul componentelor din tabel minus 1. Daca se incerca accesarea unui tablou in afara limitelor sale se va genera o exceptie in timpul rularii programului, `ArrayIndexOutOfBoundsException`.

De exemplu, pentru declaratiile de mai sus, `intArray[5]` si `charArray[6]` conduc la generarea exceptiei `ArrayIndexOutOfBoundsException`.

Sintaxa pentru accesul la elementul de index `index` al unui tablou este urmatoarea:

```
numeVariabilaTablou[index]
```

Inercarea de a accesa elementele sau variabila membru lungime ale unui tablou care a fost doar declarat (care nu a fost alocat cu `new`) ar conduce la generarea unei exceptii de tip `NullPointerException`, ca in cazurile urmatoare:

```
int[] tablouIntregi;
System.out.println(tablouIntregi.length); // exceptie NullPointerException
int n = tablouIntregi[0]; // exceptie NullPointerException
```

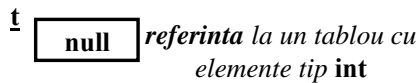
### 2.4.2.5. Exemple de lucru cu variabile de tip tablou cu elemente de tip primitiv

Sa consideram urmatoarele **declaratii**:

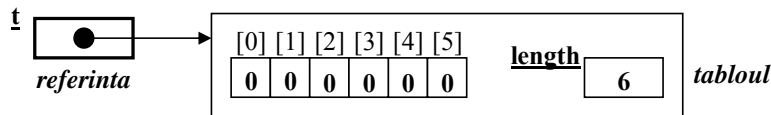
```

1   int[] t;           // declarare simpla
2   t = new int[6];   // alocare si initializare
3   int[] v;         // declarare simpla
4   v = t;           // copiere referinte
5   int[] u = { 1, 2, 3, 4 }; // declarare, alocare si initializare
6   t[1] = u[0];     // atribuire intre elemente
7   v = u;           // copiere referinte
    
```

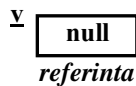
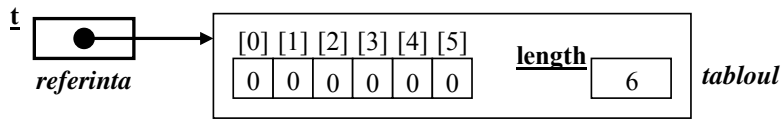
Dupa declaratia 1 se obtine:



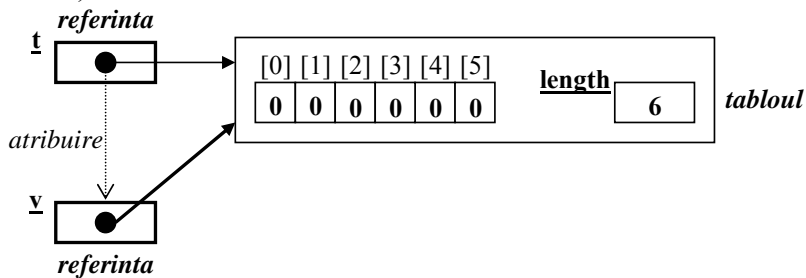
Dupa declaratia 2:



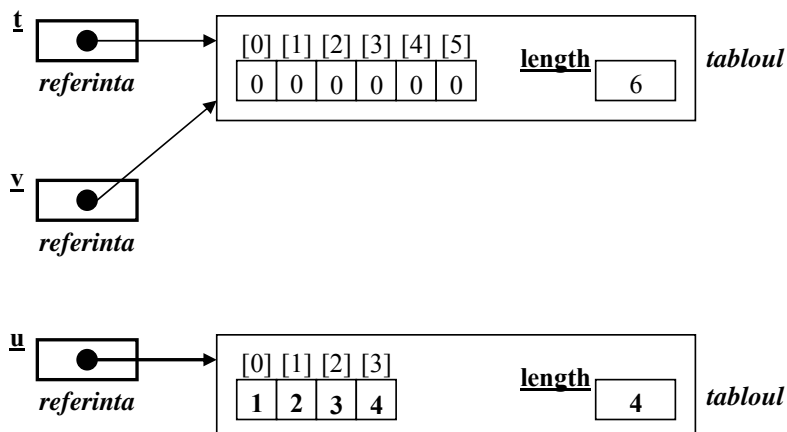
Dupa declaratia 3:



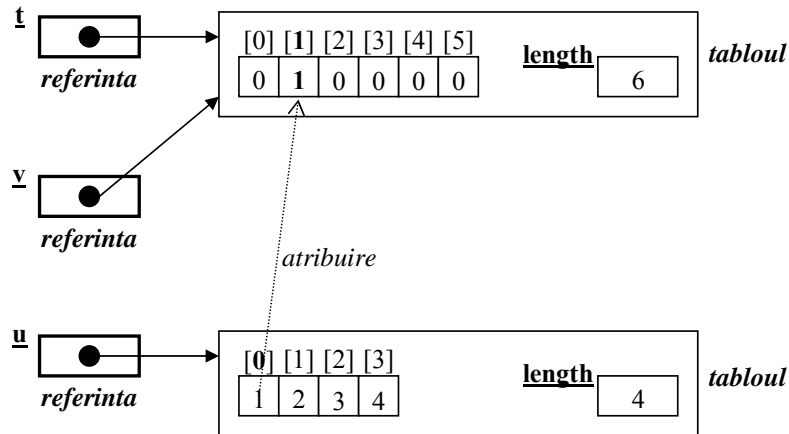
Dupa declaratia 4 (folosirea valorii referintei t pentru a fi atribuita referintei v, astfel incat (t==v) are valoarea true) se obtine:



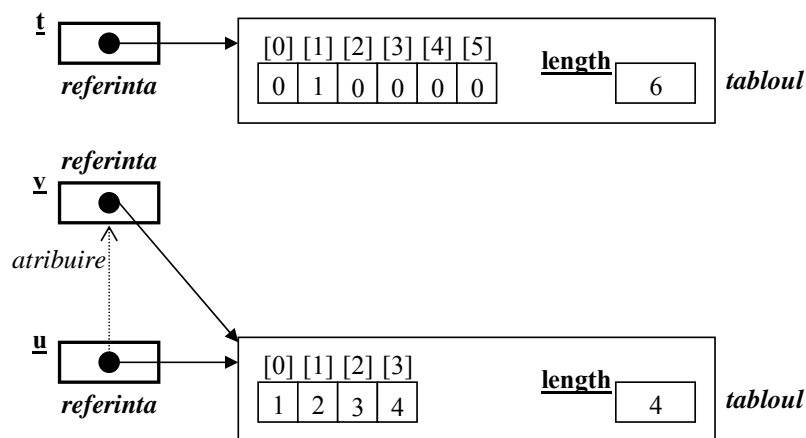
Dupa declaratia 5:



Dupa declaratia 6 (folosirea valorii continute in elementul de index 0 al variabilei **u** pentru a fi atribuita elementului de index 1 al variabilei **t**) se obtine:



Dupa declaratia 7, (**t==v**) are valoarea **false**, pe cand (**u==v**) are valoarea **true**:



### Exemplu de program complet (crearea si afisarea valorilor unui tablou de 10 intregi):

```
public class DemoTablouri {
    public static void main(String[] args) {
        int[] unTablou; // declarare tablou de int

        unTablou = new int[10]; // alocare si initializare tablou

        for (int i=0; i< unTablou.length; i++) { // folosire lungime tablou

            unTablou[i] = i; // initializare element

            System.out.print(unTablou[i] + " "); // folosire (afisare) element
        }
        System.out.println();
    }
}
```

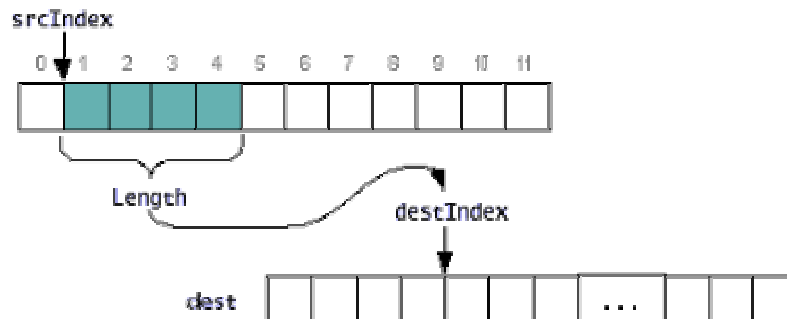
Pentru copierea tablourilor, clasa **system** din pachetul de clase implicit importate (`java.lang`) pune la dispozitie metoda `arraycopy()`.

Folosind urmatorul format de prezentare:

[modif.] tipReturnat	<b>numeMetoda</b> ([tipParametru numeParametru [, tipParametru numeParametru]]) Descrierea metodei
----------------------	--

in continuare este prezentata metoda `arraycopy()` declarata in clasa **system**:

static void	<b>arraycopy</b> (Object src, int srcPos, Object dest, int destPos, int length) <b>Copiază un (sub)tablou din tabloul sursa</b> specificat <b>src</b> , de lungime <b>length</b> , începând de la poziția specificată de indexul <b>srcPos</b> , la poziția specificată de indexul <b>destPos</b> , <b>in tabloul destinație</b> specificat <b>dest</b> .
-------------	--



### Rezultatul executiei programului:

```
public class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] tablouSursa = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                               'i', 'n', 'a', 't', 'e', 'd' };
        char[] tablouDestinatie = new char[7];

        System.arraycopy(tablouSursa, 2, tablouDestinatie, 0, 7);
        System.out.println(new String(tablouDestinatie));
    }
}
```

este afisarea caracterelor:

```
cafein
```

### 2.4.3. Variabile obiect (de tip clasa) in Java

**Clasa** reprezintă **tipul (domeniul de definiție)** unor **variabile** numite **obiecte**.

**Clasa** este o **structura complexă** care reunește **elemente de date** numite **atribute** (variabile membru, proprietăți, câmpuri, etc.) și **algoritmi** numiți **operatii** (funcții membru, metode, etc.).

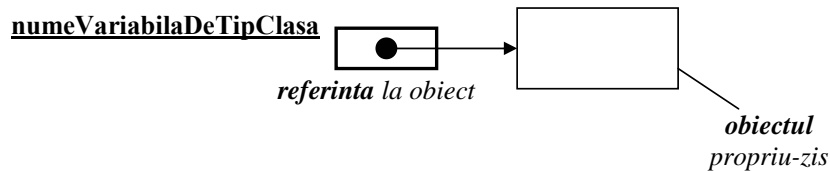
**Clasele Java** sunt **tipuri referință Java**, ceea ce înseamnă că **obiectele** sunt accesate prin intermediul unei locații numite **referință**, care **conține adresa obiectului propriu-zis**.

**Variabila obiect** este o **simplă referință la obiect**, creată **în momentul declarării ei** (moment în care poate fi inițializată implicit cu valoarea **null** – referință către nimic):

```
NumeClasa numeVariabilaObiect;
```

**Crearea dinamică a structurii obiectului** se face utilizând operatorul **new** (**funcția care are același nume cu clasa**, numită **constructor**, doar **inițializează obiectul**, adică **atributele lui**).

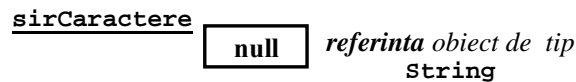
```
numeVariabilaObiect = new NumeClasa(listaDeParametri);
```



In cazul clasei `String` care incapsuleaza siruri de caractere, existenta in pachetul de clase implicit importate (`java.lang`), declaratia:

```
String sirDeCaractere; // simpla declaratie
```

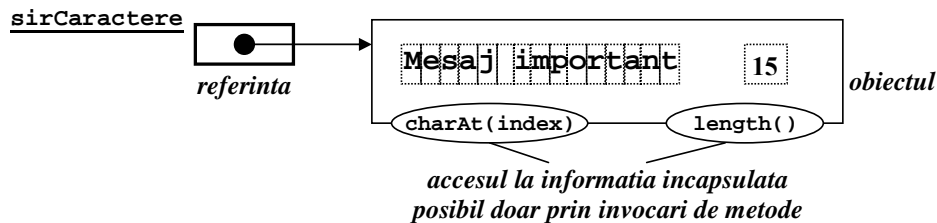
creaza doar o referinta la obiect de tip `String`, numita `sirDeCaractere`, initializata implicit cu `null`:



### Declaratia:

```
sirDeCaractere = new String("Mesaj important"); // alocare si initializare
```

creaza in mod dinamic un obiect de tip `String` a carui adresa este plasa in referinta `sirDeCaractere`, obiect care incapsuleaza sirul de caractere "Mesaj important".



Accesul la un anumit caracter, identificat printr-un anumit index, dintr-un sir de caractere incapsulat intr-un obiect `String`, se poate face doar folosind metoda `charAt()`.

De exemplu, pentru accesul la caracterul de index 0 (primul caracter) se poate folosi expresia:

```
sirDeCaractere.charAt(0)
```

De asemenea, accesul la informatia privind numarul de caractere al sirului incapsulat (lungimea sirului) este posibil doar prin intermediul metodei `length()`:

```
sirDeCaractere.length()
```

Pentru comparatie, in cazul unui tablou de caractere (care in Java este cu totul alt fel de structura):

```
char[] tablouDeCaractere = {'M','e','s','a','j',' ','i','m','p','o','r','t','a','n','t'};
```

pentru accesul la caracterul de index 0 (primul caracter) se foloseste expresia:

```
tablouDeCaractere[0]
```

iar pentru accesul la informatia privind numarul de caractere (lungimea tabloului):

```
tablouDeCaractere.length
```

## 2.4.4. Tablouri cu elemente de tip referinta

Tablourile cu elemente de tip referinta sunt de doua tipuri:

- tablouri de obiecte,
- tablouri de tablouri (in Java nu exista tablouri multi-dimensionale!).

In ambele cazuri, elementele tabloului sunt referinte la structurile propriu-zise, ale obiectelor sau ale tablourilor.

### 2.4.4.2. Declararea variabilelor de tip tablou cu elemente de tip referinta

Formatul pentru declararea variabilelor de tip tablou cu elemente de tip referinta este fie:

```
TipReferinta [] numeTablouElementeTipReferinta; // format Java
```

fie:

```
TipReferinta numeTablouElementeTipReferinta []; // format C, C++
```

unde `TipReferinta` poate fi numele unei clase, cum ar fi `String`, sau declaratia unui tablou, cum ar fi `int []`.

De exemplu:

```
String [] tablouDeSiruri; // tablou de elemente referinta la obiecte
String
int [] [] tablouDeTablouriDeIntregi; // tablou de tablouri de intregi
```

Dupa simpla declarare a unei variabile de tip tablou de referinte, spatiul alocat variabilei arata astfel:

numeVariabilaTablou null  
referinta

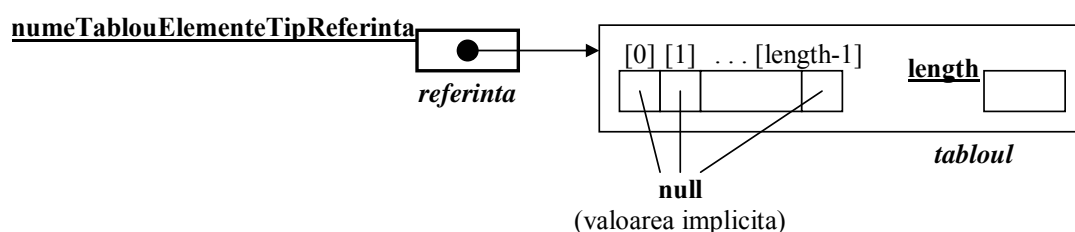
fiind alocat spatiu de memorie doar pentru locatia referinta, care e initializata implicit cu `null`.

### 2.4.4.3. Alocarea si initializarea tablourilor cu elemente de tip referinta

Formatul pentru alocarea si initializarea lungimii variabilelor de tip tablou cu elemente de tip referinta este urmatorul:

```
numeTablouElementeTipReferinta = new TipReferinta [numarElementeTablou];
```

Dupa declarare (care include alocare si initializare), spatiul alocat variabilei arata astfel:



elementele tabloului fiind initializate cu valoarea `null`.

Dupa o astfel de declarare, alocare si initializare, elementele tabloului trebuie la randul lor alocate si initializate, inainte de a fi utilizate.

In caz contrar, **incercarea de a accesa aceste elemente** (incercarea de a le atribui valori sau de a le utiliza valorile) **inainte de a fi alocate cu new va conduce la generarea unei exceptii de tip `NullPointerException`**.

Ca in situatiile:

```
String[] tablouDeSiruri;  
System.out.println(tablouDeSiruri.length); // exceptie NullPointerException  
String s = tablouDeSiruri[0]; // exceptie NullPointerException
```

O cale alternativa de a aloca si initializa un tablou in Java este sa se specifice o lista cu elementele tabloului initializate in momentul declararii tabloului, ca mai jos:

```
tablouLitereGrecesti = { "alfa", "beta", "gama", "delta" };  
tablouDeTablouriDeNumere = { { 1, 2 } , { 2, 3, 4 } , { 3, 4, 5, 6 } };
```

In acest caz, `tablouLitereGrecesti` va fi un tablou de 4 elemente de tip `String`, iar `tablouDeTablouriDeNumere` va fi un tablou care contine 3 elemente de tip tablou de intregi (fiecare avand alt numar de elemente!).

In ambele cazuri, **utilizand aceasta metoda de initializare, sunt alocate si initializate atat tablourile cat si elementele lor** (tablouri sau obiecte), astfel incat se elimina posibilitatea generarii exceptiei `NullPointerException`.

#### 2.4.4.4. Accesul la variabile de tip tablou cu elemente de tip referinta

Dupa declararea, alocarea si initializarea variabilelor de tip tablou cu elemente de tip referinta, acestea pot fi **acesate**.

**Sintaxa pentru obtinerea dimensiunii unui tablou** este urmatoarea:

```
numeTablouElementeTipReferinta.length
```

pe cand **sintaxa pentru accesul la elementul de index `index` al unui tablou** este urmatoarea:

```
numeTablouElementeTipReferinta[index]
```

De exemplu, pentru declaratiile de mai sus, `tablouLitereGrecesti.length` are valoarea 4, iar `tablouDeTablouriDeNumere.length` are valoarea 3.

Indecsi tablourilor Java pot lua valori **de la 0 la dimensiunea tabloului minus 1**. Incercarea de a accesa un tablou in afara limitelor sale genereaza o **exceptie `ArrayIndexOutOfBoundsException`**. De exemplu, `tablouLitereGrecesti[5]` si `tablouDeTablouriDeNumere[3]` conduc la generarea exceptiei `ArrayIndexOutOfBoundsException`.

### 2.4.4.5. Exemple de lucru cu variabile de tip tablou cu elemente de tip referinta

Urmatorul program afiseaza numarul de caractere al sirurilor de caractere continute in tabloul `tablouSiruri`, exemplificand lucrul cu un tablou de referinte la obiecte `String`.

```
public class AfisareTablouSiruri {
    public static void main(String[] args) {
        String[] tablouSiruri = {"Primul sir", "Al doilea sir", "Al treilea sir "};

        for (int index = 0; index < tablouSiruri.length; index++) {
            System.out.println(tablouSiruri[index].length());
        }
    }
}
```

Se poate observa diferenta dintre modul de a obtine numarul de elemente al unui tablou, folosind variabila sa membru `length`, si modul de a obtine lungimea numarul de caractere al unui sir de caractere `String`, folosind metoda sa `length()`.

Urmatorul program creeaza, populeaza si apoi afiseaza elementele unui matrice, exemplificand lucrul cu un tablou de tablouri cu elemente de tip primitiv `int`.

```
public class PopulareSiAfisareMatriceIntregi {
    public static void main(String[] args) {

        // crearea matricii
        int[][] matrice = new int[4][]; // crearea tabloului

        // popularea matricii
        for (int i=0; i < matrice.length; i++) { // matrice.length = 4
            matrice[i] = new int[5]; // crearea sub-tablourilor
            for (int j=0; j < matrice[i].length; j++) {
                matrice[i][j] = i+j; // initializarea elementelor sub-tablourilor
            }
        }

        // afisarea matricii
        for (int i=0; i < matrice.length; i++) {
            for (int j=0; j < matrice[i].length; j++) {
                System.out.print(matrice[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Tabloul `matrice` contine un numar de `matrice.length = 4` tablouri, fiecare dintre ele avand un numar de `matrice[i].length = 5` elemente.

Sub-tablourile pot fi accesate individual folosind sintaxa `matrice[i]`, iar elementele tabloului `i` pot fi accesate folosind sintaxa `matrice[i][j]`. Elementul `matrice[0][2]` reprezinta, de exemplu, al treilea element al primului tablou, iar elementul `matrice[2][0]` reprezinta primul element al celui de-al treilea tablou.

Se observa ca mai intai a fost declarat, alocat si initializat tabloul `matrice`, apoi au fost alocate si initializate cele 4 sub-tablouri, si abia apoi au fost populate cu valori.



Urmatorul program initializeaza cu bloc de initializare a tablourilor si apoi afiseaza elementele unui tablou cartoons, **exemplificand** lucrul cu un tablou de tablouri **de referinta la obiecte string**.

```
public class AfisarePersonajeDeseneAnimate {  
    public static void main(String[] args) {  
        String[][] cartoons =  
        { { "Flintstones", "Fred", "Wilma", "Pebbles", "Dino" },  
          { "Rubbles", "Barney", "Betty", "Bam Bam" },  
          { "Jetsons", "George", "Jane", "Elroy", "Judy", "Rosie",  
            "Astro" },  
          { "Scooby Doo Gang", "Scooby Doo", "Shaggy", "Velma",  
            "Fred", "Daphne" }  
        };  
  
        for (int i = 0; i < cartoons.length; i++) {  
            System.out.print(cartoons[i][0] + ": ");  
            for (int j = 1; j < cartoons[i].length; j++) {  
                System.out.print(cartoons[i][j] + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

Tabloul **cartoons** contine un numar de **cartoons.length = 4** tablouri, fiecare dintre ele avand alt numar de elemente **cartoons[i].length** (**cartoons[0].length = 5**, **cartoons[1].length = 4**, **cartoons[2].length = 7**, **cartoons[3].length = 6**).

Sub-tablourile pot fi accesate individual folosind sintaxa **cartoons[i]**, iar elementele tabloului **i** pot fi accesate folosind sintaxa **cartoons[i][j]**.

Elementul **cartoons[0][2]** reprezinta, de exemplu, al treilea element al primului tablou ("**Wilma**"), iar elementul **cartoons[2][0]** reprezinta primul element al celui de-al treilea tablou ("**Jetsons**").

Elementele **cartoons[i][0]** reprezinta numele unor desene animate, iar elementele **cartoons[i][j]** (**j>1**) reprezinta numele unor personaje din acele desene animate.