Cuprins detaliat Curs SwRTc

(**2005**-2006)

1. Introducere

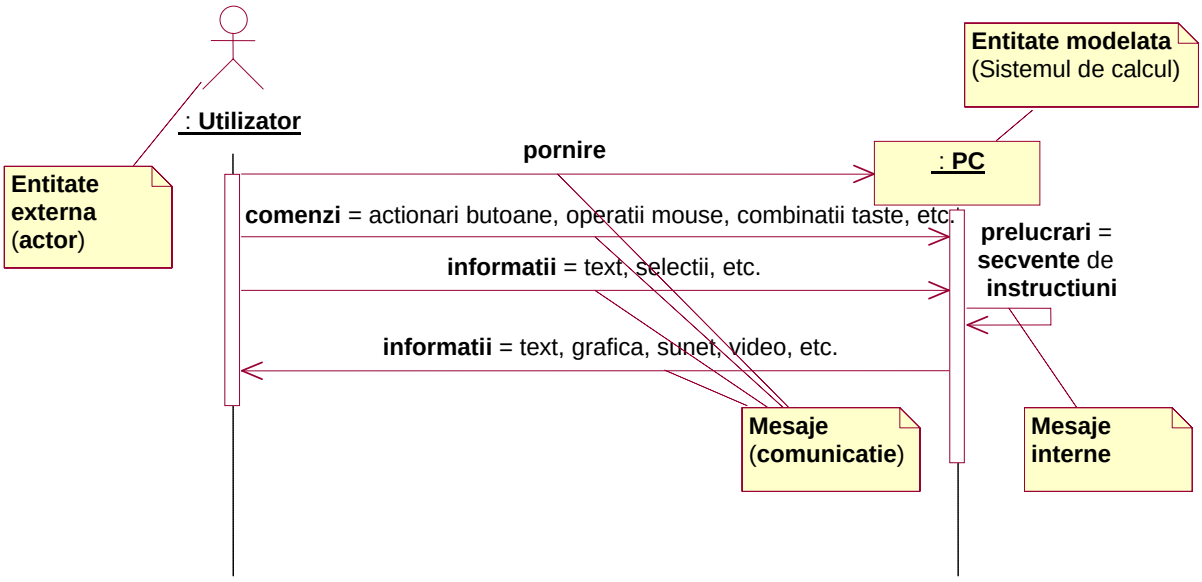
1.1. Introducere in sisteme programabile (calculatoare) si programare

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*.

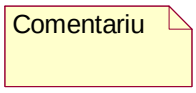
Hardware-ul cuprinde in primul rand **procesorul** (care efectueaza calculele, prelucrarile) si **memoria** (in care se stocheaza programe si datele necesare acestora pentru a realiza sarcinile date), care sunt direct conectate. In plus mai contine **elemente periferice** (memorie externa, tastatura, 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:



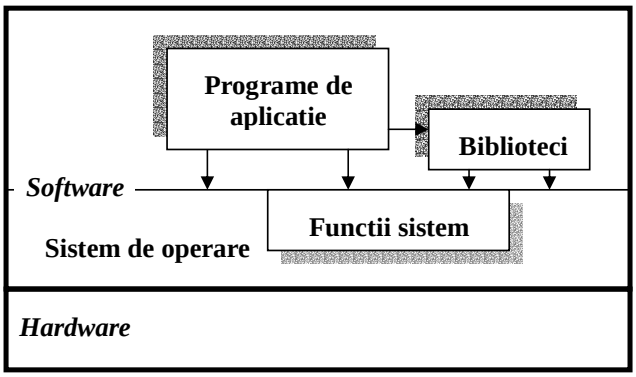
Pentru comentarea elementelor diagramei s-a folosit simbolul UML:



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).

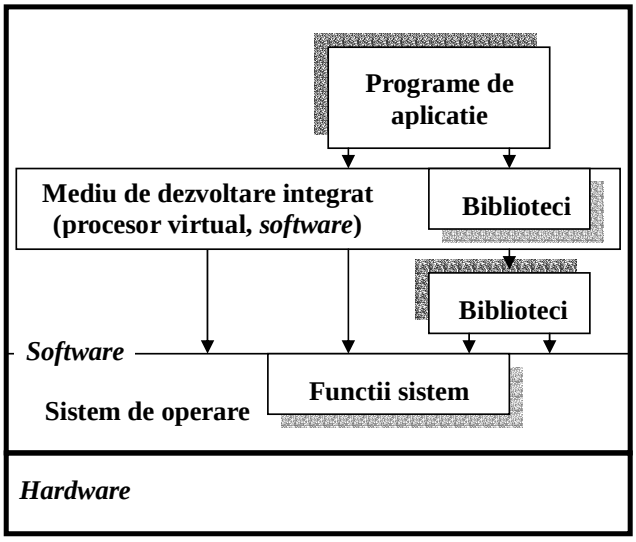
Programele sunt **secvente** de operatii de baza (numite **instructiuni**) efectuate de procesor. *Software-ul* cuprinde **un nivel de baza**, numit **sistem de operare**, care face posibil lucrul in mod transparent cu elementele *hardware*. Sistemul de operare ofera programatorilor un set de functii (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 creaza 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:

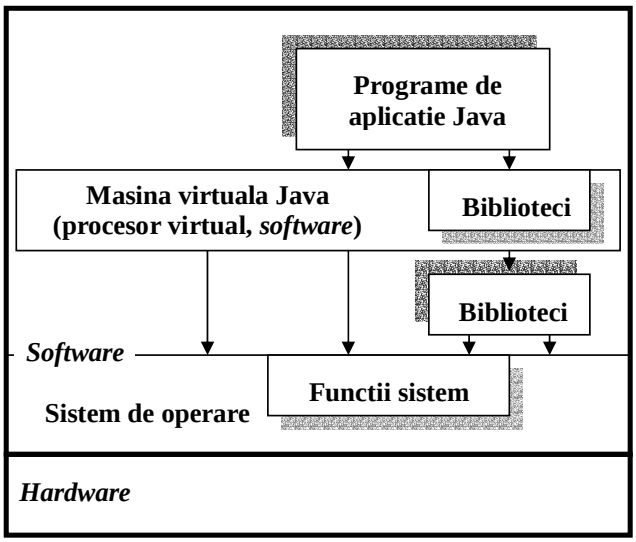


O **posibila problema** este in acest caz **dependenta introdusa de bibliotecile pe care le ofera IDE-ul**. Totusi, un IDE bun ofera o multitudine de **unelte de dezvoltare** bine organizate si exercita un **control mai strict** (decat cel exercitat de sistemul de operare) **al resurselor utilizate**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

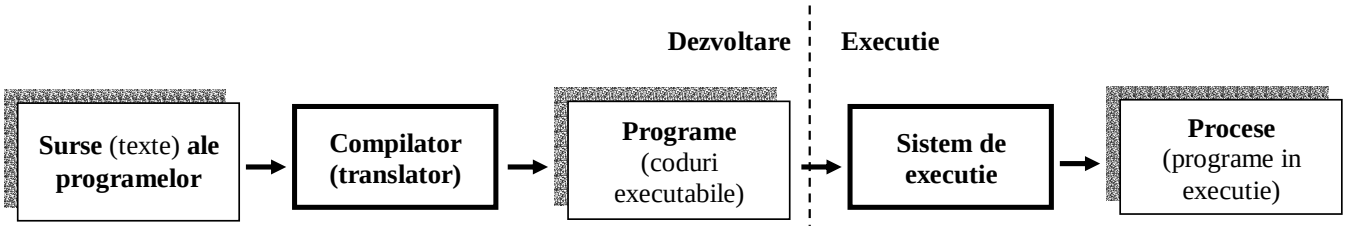
1.2. Dezvoltarea programelor in limbajul Java. Masina virtuala Java (JVM)

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.

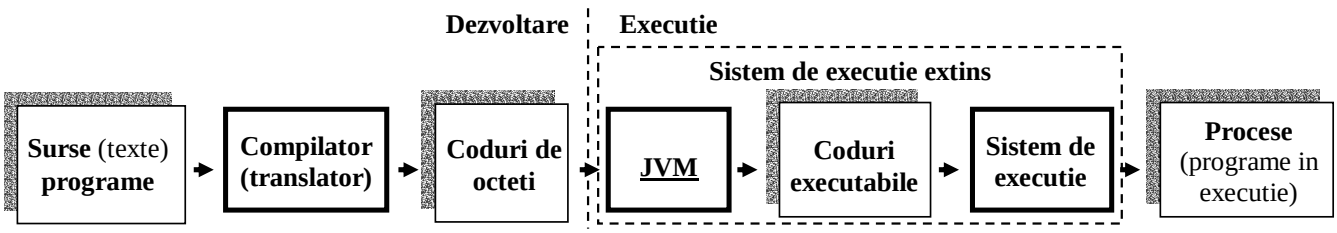
Executia programelor de catre masina virtuala Java se numeste **interpretare**, deoarece codul de octeti Java (obtinut in urma compilarii) este interpretat de JVM, adica este convertit in timp real in cod direct executabil pe sistemul de operare curent utilizat, pentru ca apoi sa fie executat. Stiva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



Pe un sistem programabil traditional programele sunt mai intai **convertite** (traduse, translatate, ***compilate***) 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**.

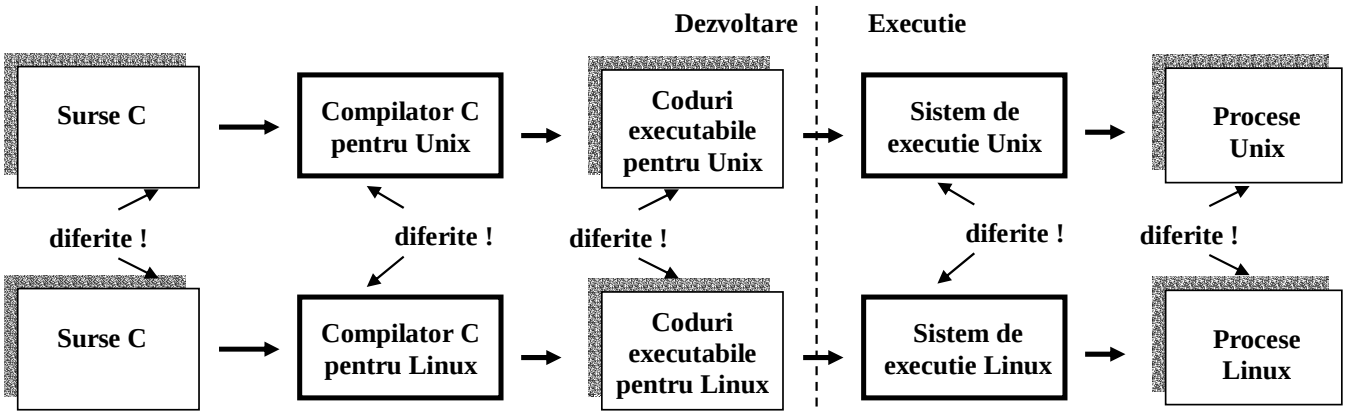


In **sistemul de programare Java** programele sunt ***compilate*** de la limbajul de programare Java la coduri executabile de procesorul software (JVM), numite coduri de octeti (*bytecodes*), pentru ca apoi *codurile de octeti* sa fie **executate de JVM** (iar **JVM este executata sub controlul sistemului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

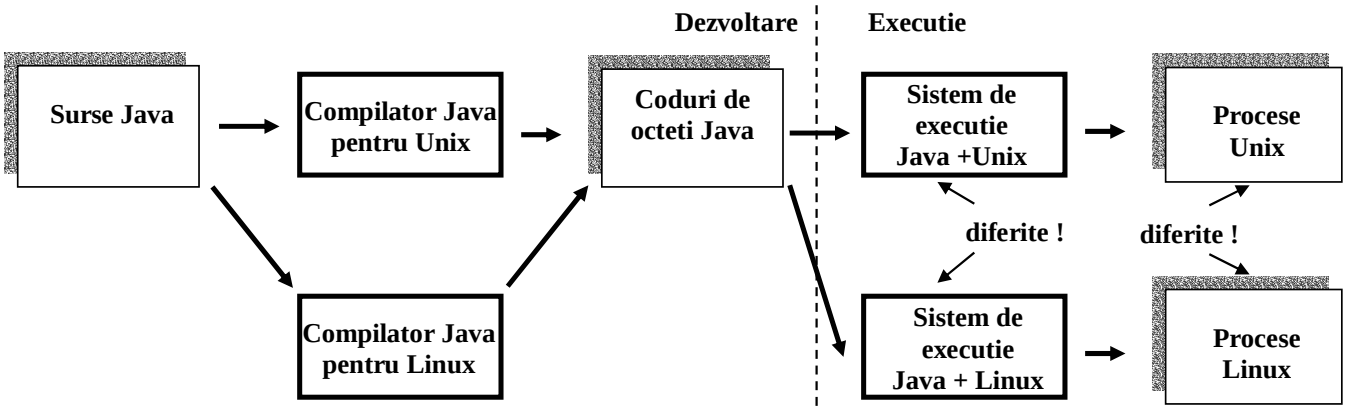


Avantajul obtinut de Java este acela ca, daca se creaza cate un procesor *software* Java (masina virtuala Java – JVM) pentru fiecare tip de sistem de operare existent, atunci pe orice calculator poate fi instalata o JVM, pe care apoi pot fi executate *coduri de octeti* Java care au fost compilate pe 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 nivelui interpretorului si la nivelul sistemului de operare si al *hardware*-ului).

O **solutie de imbunatatire a vitezei de executie** este utilizarea unui **compilator 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 evidentiate 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, modificator) 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, modificator) 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 cuvant 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 caracterе** (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 alte 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 iesrire), 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. Aceat 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

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

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 (**exceptii**)	public (**OO**)
boolean	**float**	**return**
break	**for**	**short**
byte	**if**	**static**
case	implements (**OO**)	super (**OO**)
catch (**exceptii**)	import	**switch**
char	instanceof (**OO**)	synchronized
class (**OO**)	**int**	this (**OO**)
continue	interface (**OO**)	throw (**exceptii**)
default	**long**	throws (**exceptii**)
do	native	transient
double	new (**OO**)	try (**exceptii**)
else	package	**void**
extends (**OO**)	private (**OO**)	**volatile**
final (**OO**)	protected (**OO**)	**while**
strictfp (*din Java2*)		

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 construieste 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 intregi primitive din Java.

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

Tabelul 2.3.6. Operatori pentru tipuri intregi 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.

Operator	***Operatie***
`=`	Egalitate
`!=`	Inequalitate
`>`	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 fiecarei 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.

Primitiva	***Valoare predefinita***
`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.

Din	***In***
`byte`	`short`, `int`, `long`, `float`, **or** `double`
`short`	`int`, `long`, `float`, **or** `double`
`char`	`int`, `long`, `float`, **or** `double`
`int`	`long`, `float`, **or** `double`
`long`	`float` **or** `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

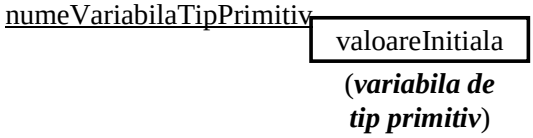
Odata cu declararea, variabilelor de tip primitiv Java **li se aloca spatiul de memorie** necesar (1B pentru `byte`, 2B pentru `short`, 4B pentru `int`, 8B pentru `long`, 4B pentru `float`, 8B pentru `double`, 2B pentru `char`, si 1b pentru `boolean`).

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

Este posibila si **initializarea explicita a variabilelor de tip primitiv**, folosind urmatorul format:

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

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



(***variabila de tip primitiv***)

3. Accesul la variabile de tip primitiv

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

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

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

Scrierea (atribuirea unei valori) **si citirea** (folosirea valorii pe care o contine) unei variabile se numesc colectiv **acces** la acea variabila.

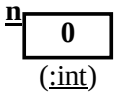
Pentru exemplificare vom folosi urmatoarele linii de cod Java:

```
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: rezulta:

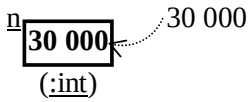
```
1       int n;
```



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

Dupa: rezulta:

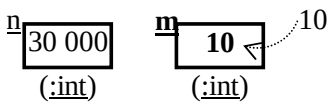
```
2       n = 30000;
```



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

Dupa: rezulta:

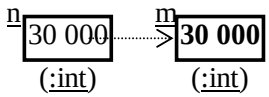
```
3       int m = 10;
```



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

Dupa: rezulta:

```
4       m = n;
```



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 **O**. Pentru **exemple in privinta folosirii valorilor intregi literale**, consultati Tabelul 2.3.10.

Tabelul 2.3.10. Exemple de valori intregi literale.

Integer	*Long*	*Octal*	*Hexadecimal*
0	0**L**	**0**	**0x**0
1	1**L**	**0**1	**0x**1
10	10**L**	**0**12	**0x**A
15	15**L**	**0**17	**0X**F
16	16**L**	**0**20	**0x**10
100	100**L**	**0**144	**0x**64

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 un "**`F`**". Pentru a arata ca trebuie tratat in dubla precizie (***double***), i se adauga un "**`d`**" sau un "**`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

Desi **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 putin pentru stocarea mesajelor de eroare. O ***valoare literala sir de caractere*** este format din **zero sau mai multe caractere** (incluzand secventele escape din Tabelul 10) **inchise intre 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 literala 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.

Operator	***Operatie***	***Exemplu***
`=`	Atribuire	`a = b`
`==`	Egalitate	`a == b`
`!=`	Inegualitate	`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.

Operator	***Operatie***	***Exemplu***
`!`	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 faca 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 continuie 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 urmatorul 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, urmatorul 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 **bucla `do…while` *se executa garantat cel putin o data,*** pe cand o **bucla `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***.

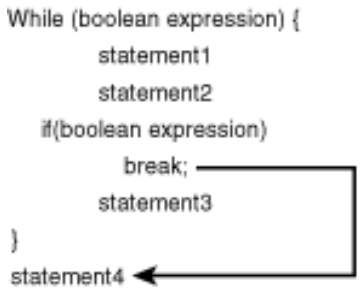


Figura 2.3.1: *Controlul executiei programului cu instructiunea break.*

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***.

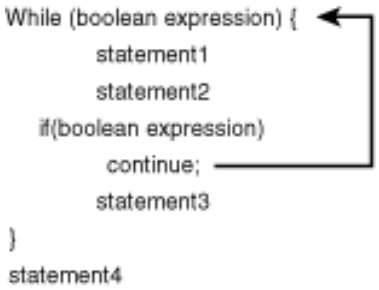


Figura 2.3.2 : *Controlul executiei programului cu instructiunea continue.*

2.3.7.8. Folosirea etichetelor

Java nu include o instructiune `goto`. Dar pentru ca `goto` este un cuvant 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 repozitioneze 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

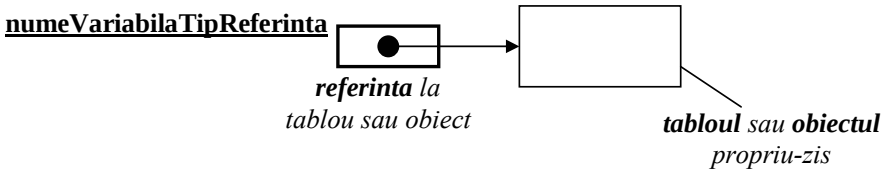
2.4.1. Introducere in tipuri referinta Java

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`**).

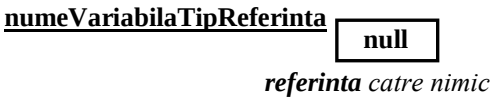
Notatie generala:



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`**.



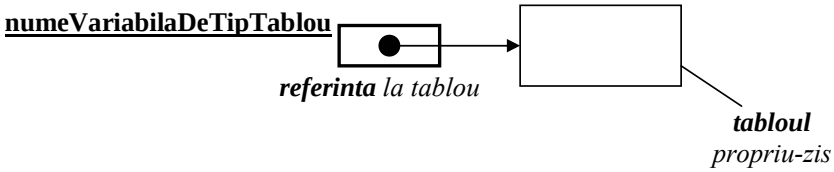
2.4.2. Tablouri cu elemente de tip primitiv

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 **simpla referinta la tablou**, **creata in momentul declararii ei** (moment in care poate fi initializata implicit cu valoarea **`null`** – referinta catre nimic). Crearea dinamica a structurii se face utilizand operatorul **`new`**.



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

Elementele tabloului (valorile incapsulate in structura tabloului) pot fi **accesate pe baza indexului** fiecarui element, care variaza de la **`0`** la **`length-1`**.

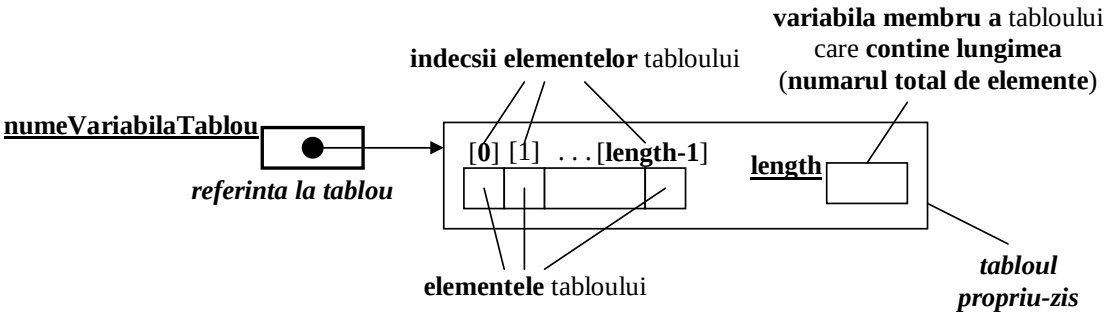


Fig. x. Componentele interne ale unui tablou Java

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

Tablourile sunt tipuri referinta Java, ceea ce inseamna ca tablourile sunt accesate prin intermediul unei locatii numita **referinta**, care **contine adresa tabloului propriu-zis**.

Dupa simpla declarare a unei variabile de tip tablou (care **nu include alocare a memoriei pentru tabloul propriu-zis**), spatiul alocat variabilei arata astfel:

fiind alocat spatiu de memorie doar pentru locatia referinta, care e initializata implicit cu **null** (referinta **catre nimic**, indicand **lipsa spatiului alocat pentru tablou** !).

In Java **tablourile se declara** utilizand **paranteze patrate inchise** (`[]`).

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

```
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:

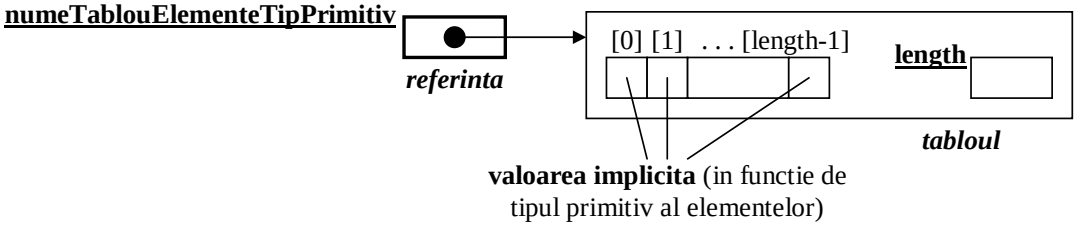
```
// stil C
int intArray[] = new int[100]; // declaratie si definitie (initializare cu 0)
float floatArray[];            // simpla declaratie
floatArray = new float[100];   // definitie (si initializare implicita cu 0)

// 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 aloca 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 incearca 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]
```

Incercarea 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:

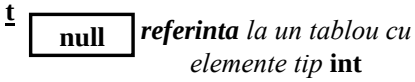
```
int[] tablouIntregi;
System.out.println(tablouIntregi.length); // exceptie NullPointerException
int n = tablouIntregi[0];                 // exceptie NullPointerException
```

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

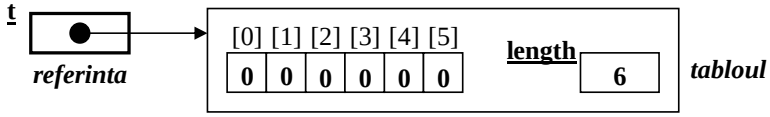
Sa consideram urmatoarele **declaratii**:

```
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
```

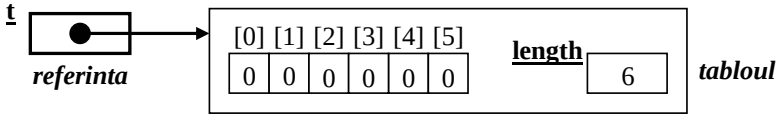
Dupa declaratia **1** se obtine:



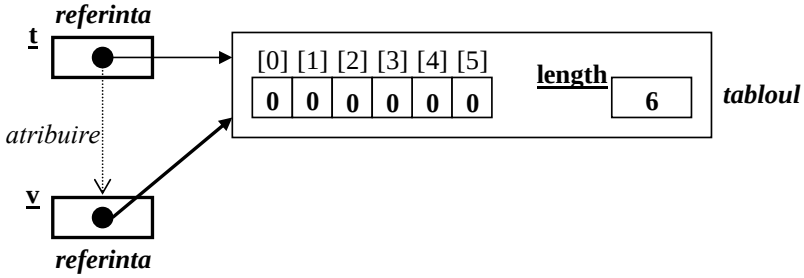
Dupa declaratia **2**:



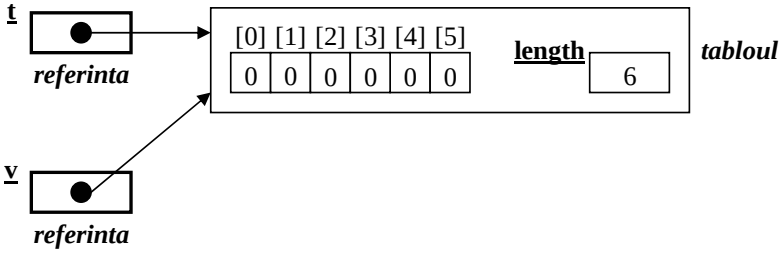
Dupa declaratia **3**:

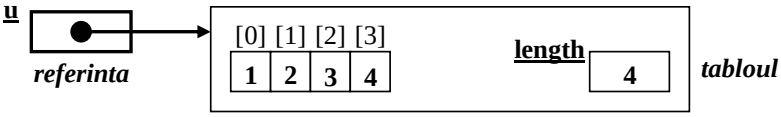


Dupa declaratia **4** (**folosirea valorii referintei t** pentru a fi **atribuita referintei v**, astfel incat **`(t==v)`** are valoarea **`true`**) se obtine:

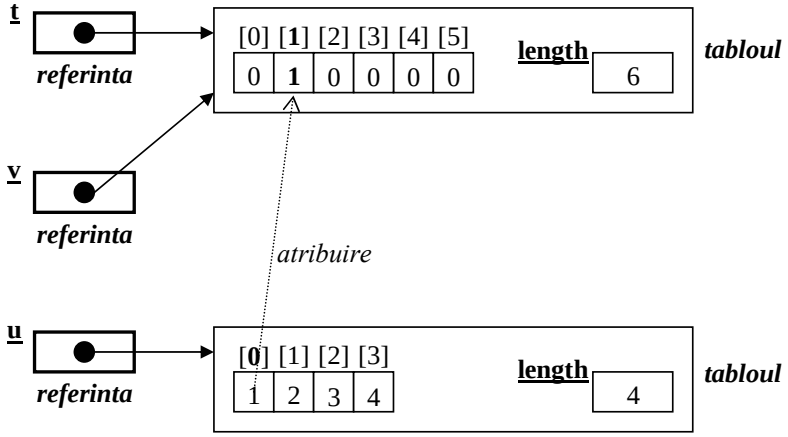


Dupa declaratia **5**:

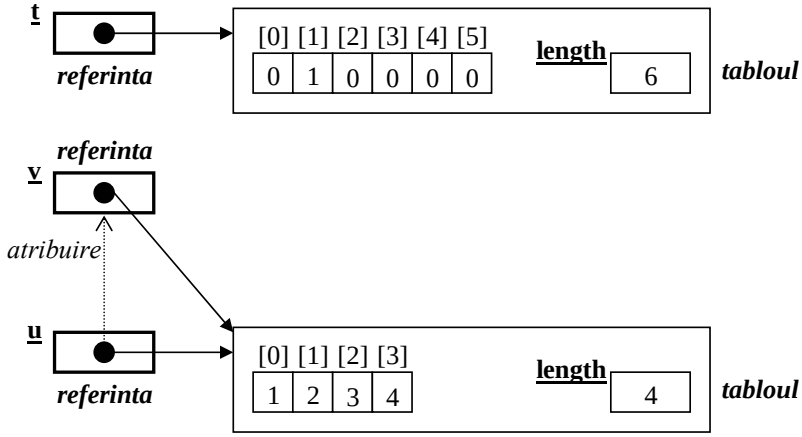




Dupa declaratia **6** (**folosirea valorii continute in elementul** de index **0** al variabilei **u** pentru a fi **atribuita elementului** de index **1** al variabilei **t**) se obtine:



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();
  }
}
```

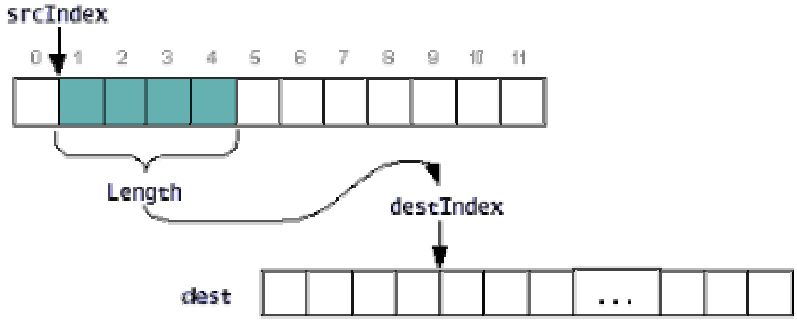
Pentru copierea tablourilor, clasa **`System`** din pachetul de clase implicit importate (`java.lang`) pune la dispozitie metoda **`arraycopy()`**.

Folosind urmatorul format de prezentare:

`[modif.] tipReturnat`	**`numeMetoda`**`([tipParametru numeParametru [, tipParametru numeParametru]])` Descrierea metodei

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

`static void`	**`arraycopy`**`(Object src, int srcPos, Object dest, int destPos, int length)` **Copiaza un (sub)tablou din tabloul sursa** specificat **`src`**, de lungime **`length`**, incepand de la pozitia specificata de indexul **`srcPos`**, la pozitia specificata de indexul **`destPos`**, **in tabloul destinatie** specificat **`dest`**.



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:

```
caffein
```

2.4.3. Variabile obiect (de tip clasa) in Java

Clasa reprezinta **tipul (domeniul de definitie)** unor **variabile** numite **obiecte**.

Clasa este **o structura complexa** care reuneste **elemente de date** numite **atribute** (variabile membru, proprietati, campuri, etc.) **si algoritmi** numiti **operatii** (functii membru, metode, etc.).

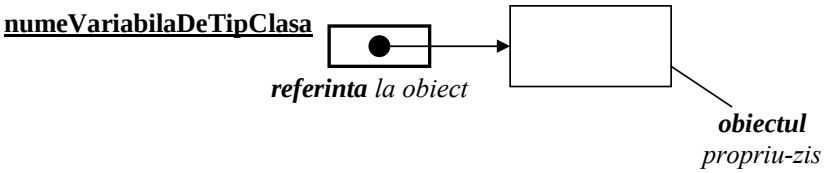
Clasele Java sunt **tipuri referinta Java**, ceea ce inseamna ca **obiectele** sunt accesate prin intermediul unei locatii numita **referinta**, care **contine adresa obiectului propriu-zis**.

Variabila obiect este o **simpla referinta la obiect**, **creata in momentul declararii ei** (moment in care poate fi initializata implicit cu valoarea **`null`** – referinta catre nimic):

```
NumeClasa  numeVariabilaObiect;
```

Crearea dinamica a structurii obiectului se face utilizand operatorul **`new`** (**functia care are acelasi nume cu clasa**, numita **constructor**, doar **initializeaza obiectul**, adica **atributele** lui).

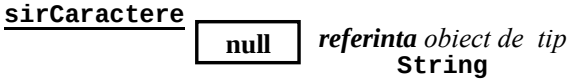
```
numeVariabilaObiect = new NumeClasa(listaDeParametri);
```



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

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

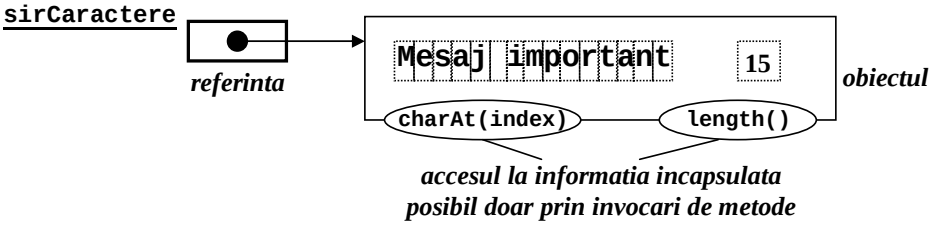
creaza doar o referinta la obiect de tip **`String`,** numita **`sirDeCaractere`,** initializata implicit cu **`null`**:



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:

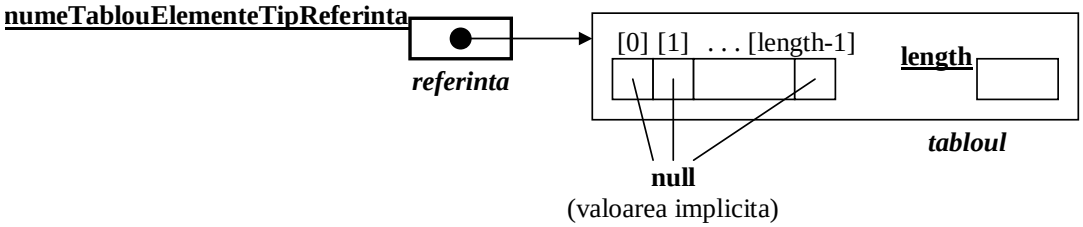
fiind **alocat spatiu de memorie doar pentru locatia referinta**, care e **initializata implicit** cu **null**.

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 **accesate**.

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**.

Indecsii tablourilor Java pot lua valori **de la 0 la dimensiunea tabloului minus 1**. Incearcarea 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 referinte 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.

2.5. Clase Java pentru lucrul cu (siruri de) caractere

2.5.1. Incapsularea caracterelor si a sirurilor de caractere

Platforma Java contine trei **clase care pot fi folosite pentru lucrul cu date de tip (sir de) caractere**:
- **Character** - clasa care incapsuleaza o singura **valoare caracter**
- **String** - clasa care incapsuleaza un **sir de caractere ne**modificabil (*immutable*)
- **StringBuffer** - clasa care incapsuleaza un **sir de caractere modificabil** (*mutable*)

2.5.2. Clasa care incapsuleaza caractere Unicode (Character) – interfata publica

Clasa **`Character`** incapsuleaza o singura **valoare caracter**, **nemodificabila** (*immutable*). Caracterul incapsulat este astfel protejat, poate fi pasat in interiorul unui obiect (accesat prin referinta), comparat, convertit. De asemenea, caracterului ii poate fi determinat tipul (litera, numar, etc.).

2.5.2.1. Declaratia clasei Character

Declaratiile de pachet si clasa ale clasei **`Character`** sunt urmatoarele:

```
package java.lang;

public final class Character extends Object implements java.io.Serializable,
                                                       Comparable {
  // corpul clasei Character
}
```

2.5.2.2. Constructorii clasei Character

Constructorul clasei `Character`:

`Character``(char value)` Construieste un obiect `Character` nou care incapsuleaza valoarea `char` specificata ca argument.

2.5.2.3. Metodele clasei Character

Folosind urmatorul format de prezentare:

`[modif.] tipReturnat`	**`numeMetoda`**`([tipParametru numeParametru [,` `tipParametru numeParametru]])` Descrierea metodei

in continuare sunt detaliate cateva dintre **metodele** declarate in clasa **`Character`**:

char	**charValue**() Returneaza valoarea primitiva incapsulata in acest obiect `Character`.
int	**compareTo**(Character anotherCharacter) Compara numeric doua obiecte `Character`. Rezultatul este diferenta intre codificarea caracterului incapsulat obiectul curent si codificarea incapsulat in caracterul primit ca parametru.
int	**compareTo**(Object o) Compara acest `Character` (`this`) cu un obiect al clasei `Object`. Daca obiectul argument este un `Character` functia se comporta ca `compareTo(Character)`. Altfel, metoda arunca o exceptie de tipul `ClassCastException` (obiectele `Character` fiind comparabile doar intre ele).
boolean	**equals**(Object obj) Compara continutul obiectului curent cu continutul obiectului primit ca parametru.
static int	**getNumericValue**(char ch) Returneaza valoare `int` care reprezinta codificarea Unicode a caracterului primit ca parametru.
static int	**getType**(char ch) Returneaza o value indicand categoria generala a caracterului specificat ca parametru.
static boolean	**isDefined**(char ch) Determina daca un caracter es if a character is defined in Unicode.
static boolean	**isDigit**(char ch) Determina daca un caracter specificat este digit (caz in care `Character.getType(ch)` este `DECIMAL_DIGIT_NUMBER`.).
static boolean	**isLetter**(char ch) Determina daca un caracter specificat este litera.
static boolean	**isLetterOrDigit**(char ch) Determina daca un caracter specificat este litera sau digit.
static boolean	**isLowerCase**(char ch) Determina daca un caracter specificat este caracter litera mica.
static boolean	**isSpaceChar**(char ch) Determina daca un caracter specificat este caracter Unicode spatiu.
static boolean	**isUpperCase**(char ch) Determina daca un caracter specificat este caracter litera mare.
static boolean	**isWhitespace**(char ch) Determina daca un caracter specificat este *white space* conform Java (vezi documentatia Java).
static char	**toLowerCase**(char ch) Converteste caracterul primit ca argument la litera mica.
String	**toString**() Returneaza un obiect `String` care incapsuleaza caracterul curent.
static String	**toString**(char c) Returneaza un obiect `String` care incapsuleaza caracterul specificat ca argument (`char`).
static char	**toUpperCase**(char ch) Converteste caracterul primit ca argument la litera mare.

Asadar, **clasa `Character` permite**:
- **comparatii** intre caractere (**`compareTo()`**, **`equals()`**, etc.),
- **determinarea tipului** de caracter (**`isLetter()`**, **`isDigit()`**, **`isLowerCase()`**, **`isUpperCase()`**, etc.),
- **conversii** (**`toLowerCase()`**,**`toUpperCase()`**, **`toString()`**, etc.).

Exemplu de utilizare a clasei **`Character`**:

```
public class CharacterDemo {
   public static void main(String args[]) {
      Character a = new Character('a');
      Character a2 = new Character('a');
      Character b = new Character('b');

      int difference = a.compareTo(b);

      if (difference == 0) {
         System.out.println("a este egal cu b.");
      } else if (difference < 0) {
         System.out.println("a este mai mic decat b.");
      } else if (difference > 0) {
         System.out.println("a este mai mare decat b.");
      }
      System.out.println("a is " + ((a.equals(a2)) ? "equal" : "not equal")
         + " to a2.");
      System.out.println("Caracterul" + a.toString() + " este "
         + (Character.isUpperCase(a.charValue()) ? "upper" : "lower") + "case.");
   }
}
```

Rezultatul executiei programului este urmatorul:

```
a este mai mic decat b.
a este egal cu a2.
Caracterul a este lowercase.
```

2.5.3. Clasa care incapsuleaza siruri de caractere nemodificabile (String) – interfata publica

Clasa **`String`** incapsuleaza **un sir de caractere**, **nemodificabil** (*immutable*). Sirul de caractere incapsulat este astfel protejat, poate fi pasat in interiorul unui obiect (accesat prin referinta), comparat, convertit la valori numerice sau la tablouri de octeti sau caractere. De asemenea, caracterele pe care le contine pot fi accesate individual.

2.5.3.1. Declaratia clasei String

Declaratiile de pachet si clasa ale clasei **`String`** sunt urmatoarele:

```
package java.lang;

public final class String implements java.io.Serializable, Comparable,
CharSequence {
  // corpul clasei String
}
```

2.5.3.2. Constructorii clasei String

Principalii constructori ai clasei `String`:

`String``()` Initializeaza un obiect `String` nou creat astfel incat sa reprezinte un sir de caractere vid.
`String``(byte[] bytes)` Construieste un nou obiect `String` prin decodarea unui tablou de octeti specificat utilizand setul de caractere implicit al platformei pe care se lucreaza. `bytes` reprezinta tabloul de octeti ce va fi decodat in caractere.
`String``(byte[] bytes, int offset, int length)` Construieste un nou obiect `String` prin decodarea unui sub-tablou de octeti specificat (`bytes` reprezinta tabloul de octeti ce va fi decodat in caractere, `offset` indexul primului octet din tablou ce va fi decodat iar `length` numarul de octeti ce va fi decodat) utilizand setul de caractere implicit al platformei pe care se lucreaza..
`String``(char[] value)` Construieste un nou obiect `String` astfel incat sa reprezinte sirul de caractere continut in tabloul de caractere primit ca argument (`value`). Continutul tabloului de caractere este copiat, astfel incat modificarile ulterioare ale tabloului de caractere nu afecteaza nou creatul sir de caractere.
`String``(char[] value, int offset, int count)` Construieste un nou obiect `String` astfel incat sa reprezinte subsirul de caractere continut in tabloul de caractere primit ca argument (`value`).
`String``(String original)` Initializeaza un obiect `String` nou creat astfel incat sa reprezinte acelasi sir de caractere ca argumentul (sirul nou creat e o copie a celui primit ca argument).
`String``(StringBuffer buffer)` Construieste un nou obiect `String` care contine sirul de caractere curent continut in argumentul de tip sir de caractere modificabil (*string buffer*).

2.5.3.3. Metodele clasei String

Declaratiile si descrierea catorva metode ale clasei **`String`**:

`char`	**`charAt`**`(int index)` Returneaza caracterul aflat la indexul specificat.
`int`	**`compareTo`**`(Object o)` Compara acest `String` (`this`) cu un obiect al clasei `Object`. Daca obiectul argument este un `String` functia se comporta ca `compareTo(String)`. Altfel, ea arunca (declanseaza) o exceptie `ClassCastException` (obiectele `String` sunt comparabile doar cu alte obiecte `String`).
`int`	**`compareTo`**`(String anotherString)` Compara doua siruri de caractere din punct de vedere lexicografic. Comparatia este bazata pe valoarea Unicode a fiecarui caracter al sirului. Rezultatul este un intreg negativ daca obiectul curent (`this`) precede argumentul si pozitiv daca urmeaza dupa el. Rezultatul e nul daca sirurile sunt egale. Daca sirurile au caractere diferite la unul sau mai multe pozitii ale indexului, fie *indexMin* cel mai mic astfel de index, atunci sirul al carui caracter de index *indexMin* are valoarea cea mai mica

	(determinate utilizand operatorul <) precede lexicographic pe celalalt sir (iar valoarea returnata este in acest caz egala cu `this.charAt(indexMin)-anotherString.charAt(indexMin)`). Daca sirurile sunt de lungime diferita, iar sirul mai scurt nu difera pe toata lungimea lui de caracterele sirului mai lung, atunci el precede sirul mai lung (iar valoarea returnata este in acest caz egala cu `this.length()-anotherString.length()`).
`int`	**`compareToIgnoreCase`**`(String str)` Compara doua siruri de caractere din punct de vedere lexicografic, ignorand diferentele intre literele mari si mici.
`String`	**`concat`**`(String str)` Concateneaza sirul de caractere primit ca argument (`str`) la sfarsitul sirului curent (`this`).
`boolean`	**`contentEquals`**`(StringBuffer sb)` Returneaza `true` daca si numai daca sirul curent (`this`) reprezinta acelasi sir de caractere ca obiectul `StringBuffer` specificat ca argument (`sb`).
`static String`	**`copyValueOf`**`(char[] data)` Returneaza un obiect `String` care reprezinta sirul de caractere din tabloul de caractere the array specificat ca argument (`data`).
`static String`	**`copyValueOf`**`(char[] data, int offset, int count)` Returneaza un obiect `String` care reprezinta subsirul de caractere din tabloul de caractere the array specificat ca argument (`data`).
`boolean`	**`endsWith`**`(String suffix)` Testeaza daca acest `String` se incheie cu sufixul specificat ca argument (`suffix`).
`boolean`	**`equals`**`(Object anObject)` Compara acest sir de caractere cu obiectul specificat ca argument.
`boolean`	**`equalsIgnoreCase`**`(String anotherString)` Compara acest `String` cu alt `String`, ignorand diferentele intre literele mari si cele mici.
`byte[]`	**`getBytes`**`()` Codeaza sirul de caractere curent intr-un sir de octeti, utilizand setul de caractere implicit al platformei pe care se lucreaza, stocand rezultatul intr-un nou tablou de octeti.
`void`	**`getChars`**`(int srcBegin, int srcEnd, char[] dst, int dstBegin)` Copiaza caracterele din sirul curent in tabloul de caractere destinatie. Primul caracter copiat are index `srcBegin`; ultimul caracter copiat are `srcEnd-1` (astfel incat numarul total de caractere copiate este `srcEnd-srcBegin`). Caracterele sunt copiate intr-un subtablou `dst` incepand cu indexul `dstBegin` si terminand la indexul: `dstbegin+(srcEnd-srcBegin)-1`
`int`	**`hashCode`**`()` Returneaza un *hash code* pentru acest sir de caractere, folosind relatia: s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1] utilizand aritmetica `int`, unde `s[i]` este al *i*-lea caracter al sirului, `n` este lungimea sirului, iar `^` indica exponentierea. (Valoarea *hash* a unui sir vid este zero.)
`int`	**`indexOf`**`(int ch)` Returneaza indexul minim in sirul de caractere curent la care apare caracterul specificat ca argument, si valoarea `-1` daca nu exista acel caracter in sirul de caractere curent.

int	**indexOf**(int ch, int fromIndex) Returneaza indexul minim in sirul de caractere curent, incepand cu indexul specificat ca argument (fromIndex), la care apare caracterul specificat ca argument (ch), si valoarea -1 daca nu exista acel caracter in sirul de caractere curent.
int	**indexOf**(String str) Returneaza indexul minim in sirul de caractere curent la care apare subsirul de caractere specificat ca argument (str), si valoarea -1 daca nu exista acel subsir in sirul de caractere curent.
int	**indexOf**(String str, int fromIndex) Returneaza indexul minim in sirul de caractere curent, incepand cu indexul specificat ca argument (fromIndex), la care apare subsirul de caractere specificat ca argument (str), si valoarea -1 daca nu exista acel subsir in sirul de caractere curent.
int	**lastIndexOf**(int ch) Returneaza indexul maxim in sirul de caractere curent la care apare caracterul specificat ca argument, si valoarea -1 daca nu exista acel caracter in sirul de caractere curent.
int	**lastIndexOf**(int ch, int fromIndex) Returneaza indexul maxim in sirul de caractere curent, incepand cu indexul specificat ca argument (fromIndex), la care apare caracterul specificat ca argument (ch), si valoarea -1 daca nu exista acel caracter in sirul de caractere curent.
int	**lastIndexOf**(String str) Returneaza indexul maxim in sirul de caractere curent la care apare subsirul de caractere specificat ca argument (str), si valoarea -1 daca nu exista acel subsir in sirul de caractere curent.
int	**lastIndexOf**(String str, int fromIndex) Returneaza indexul maxim in sirul de caractere curent, incepand cu indexul specificat ca argument (fromIndex), la care apare subsirul de caractere specificat ca argument (str), si valoarea -1 daca nu exista acel subsir in sirul de caractere curent.
int	**length**() Returneaza lungimea sirului de caractere curent (numarul de caractere pe care le contine).
String	**replace**(char oldChar, char newChar) Returneaza un nou sir obtinut prin inlocuirea tuturor aparitiilor caracterului oldChar in sirul curent cu caracterul newChar.
boolean	**startsWith**(String prefix) Testeaza daca sirul de caractere curent incepe cu prefixul specificat ca argument.
boolean	**startsWith**(String prefix, int toffset) Testeaza daca subsirul de caractere al sirului curent care debuteaza la indexul toffset incepe cu prefixul specificat ca argument.
String	**substring**(int beginIndex) Returneaza ca nou sir de caractere acel subsir al sirului curent care incepe la indexul beginIndex si se termina la indexul endIndex – 1, lungimea noului sir fiind endIndex-beginIndex.
String	**substring**(int beginIndex, int endIndex) Returneaza ca nou sir de caractere acel subsir al sirului curent care incepe la indexul beginIndex si se termina la ultimul caracter al sirului curent.
char[]	**toCharArray**() Converteste sirul curent la un nou tablou de caractere.

`String`	**`toLowerCase`**`()` Converteste toate caracterele din sirul curent la litere mici, utilizand regulile locale implicite.
`String`	**`toLowerCase`**`(Locale locale)` Converteste toate caracterele din sirul curent la litere mici, utilizand regulile locale specificate.
`String`	**`toString`**`()` Returneaza un obiect `String` nou creat care reprezinta acelasi sir de caractere ca sirul curent (sirul nou creat e o copie a celui curent).
`String`	**`toUpperCase`**`()` Converteste toate caracterele din sirul curent la litere mari, utilizand regulile locale implicite.
`String`	**`toUpperCase`**`(Locale locale)` Converteste toate caracterele din sirul curent la litere mari, utilizand regulile locale specificate.
`static String`	**`valueOf`**`(boolean b)` Returneaza reprezentarea ca sir de caractere a argumentului de tip boolean (`boolean`).
`static String`	**`valueOf`**`(char c)` Returneaza reprezentarea ca sir de caractere a argumentului de tip caracter (`char`).
`static String`	**`valueOf`**`(char[] data)` Returneaza reprezentarea ca sir de caractere a argumentului tablou de caractere (`char`).
`static String`	**`valueOf`**`(char[] data, int offset, int count)` Returneaza reprezentarea ca sir de caractere a subtabloului specificat ca argument.
`static String`	**`valueOf`**`(double d)` Returneaza reprezentarea ca sir de caractere a argumentului `double`.
`static String`	**`valueOf`**`(float f)` Returneaza reprezentarea ca sir de caractere a argumentului `float`.
`static String`	**`valueOf`**`(int i)` Returneaza reprezentarea ca sir de caractere a argumentului de tip intreg (`int`).
`static String`	**`valueOf`**`(long l)` Returneaza reprezentarea ca sir de caractere a argumentului `long`.
`static String`	**`valueOf`**`(Object obj)` Returneaza reprezentarea ca sir de caractere a argumentului `Object`.

Se observa urmatoarele **echivalente functionale**:

- constructorul **`String`**`(char[] value)` cu metoda **`static`** `String` **`valueOf`**`(char[] data)`:

```
char[] caractere = {'t', 'e', 's', 't'};
String sir = new String(caractere);
// echivalent cu    String sir = String.valueOf(caractere);
```

- constructorul **`String`**`(char[] value, int offset, int count)` cu metoda **`static`** `String` **`valueOf`**`(char[] value, int offset, int count)`:

```
char[] caractere = {'t', 'e', 's', 't', 'a', 'r', 'e'};
String sir = new String(caractere, 2, 5);
// echivalent cu    String sir = String.valueOf(caractere, 2, 5);
```

- constructorul **String**(String original) cu metoda String **toString**(), dar si cu alte metode:

```
String original = "sir";
String copie = new String(original);
// echivalent cu    String copie = original.tostring();
// echivalent cu    String copie = String.valueOf(original);
// echivalent cu    String copie = original.substring(0);
// etc.
```

Se observa si urmatoarele **complementaritati functionale**:

- constructorul **String**(byte[] bytes) cu metoda byte[] **getBytes**():

```
String sir = "test";
byte[] octeti = sir.getBytes();
String copieSir = new String(octeti);
```

O **parte din codul clasei String**:

```
1   public final class String { // finala - nu poate fi extinsa prin mostenire
2
3     private char[] value; // tabloul de caractere care stocheaza sirul
4     private int offset;  // indexul primei locatii utilizate pentru stocare
5     private int count;   // numarul de caractere al sirului
6
7     // Diferiti constructori String
8     public String() {
9       value = new char[0];
10    }
11    public String(String value) {
12      count = value.length();
13      this.value = new char[count];
14      value.getChars(0, count, this.value, 0);
15    }
16    public String(char[] value) {
17      this.count = value.length;
18      this.value = new char[count];
19      System.arraycopy(value, 0, this.value, 0, count);
20    }
21
22    // Metode publice, de obiect
23    public boolean equals(Object obj) {
24      if ((obj != null) && (obj instanceof String)) {
25        String otherString = (String)obj;            // cast
26        int n = this.count;
27        if (n == otherString.count) {
28          char v1[] = this.value;
29          char v2[] = otherString.value;;
30          int i = this.offset;
31          int j = otherString.offset;
32          while (n-- != 0)
33            if (v1[i++] != v2[j++]) return false;
34          return true;
35        }
36      }
37      return false;
38    }
39    public int length() {
40      return count;
41    }
42    public static String valueOf(int i) {
43      return Integer.toString(i, 10);
44    }
45    // ... multe alte metode
46  }
```

Exemple de lucru cu obiecte de tip **`String`**.

```
1   // variabile referinta
2
3   String a;         // referinta la String initializata implicit cu null
4
5   String b = null;  // referinta la String initializata explicit cu null
6
7   // constructie siruri de caractere utilizand constructori String()
8
9   String sirVid = new String();       // sirVid.length = 0, sirVid = ""
10
11  byte[] tabByte = {65, 110, 110, 97};         // coduri ASCII
12  String sirTablouByte = new String(tabByte);  // sirTablouByte = "Anna"
13
14  char[] tabChar = {'T', 'e', 's', 't'};
15  String sirTabChar = new String(tabChar);     // sirTabChar = "Test"
16
17  String s = "Sir de caractere";
18  String sir = new String(s);                  // sir = "Sir de caractere"
19
20  // constructie siruri de caractere utilizand metoda toString()
21
22  String sirCopie = sir.toString();
23
24  // constructie siruri de caractere utilizand metode de clasa
25
26  boolean adevarat = true;
27  String sirBoolean = String.valueOf(adevarat);   // sirBoolean = "true"
28  // echivalent cu   String sirBoolean = String.valueOf(true);
29
30  char caracter = 'x';
31  String sirChar = String.valueOf(caracter);      // sirChar = "x"
32
33  char[] tab2Char = {'A', 'l', 't', ' ', 't', 'e', 's', 't'};
34  String sirTab2Char = String.valueOf(tab2Char);  // sirTabChar2="Alt test"
35
36  int numar = 10000;
37  String sirInt = String.valueOf(numar);          // sirInt = "1000"
38
39  double altNumar = 2.3;
40  String sirDouble = String.valueOf(altNumar);    // sirDouble = "2.3"
41
42  // conversia sirurilor de caractere la alte tipuri
43
44  String sirTest = "ABC abc";
45  byte[] sirTestByte = sirTest.getBytes();         // coduri ASCII
46
47  System.out.print("sirTestByte = ");
48  for (int i=0; i < sirTestByte.length; i++)
49     System.out.print(sirTestByte[i] + " ");
50  System.out.println();
51
52  char[] sirTestChar = sirTest.toCharArray();      // caractere UNICODE
53
54  System.out.print("sirTestChar =  ");
55  for (int i=0; i < sirTestChar.length; i++)
56     System.out.print(sirTestChar[i] + "  ");
57  System.out.println();
58
59  // concatenare
60
61  String f = "Sir " + "de " + "caractere";
62  // echivalent cu:  String f = "Sir de caractere";
```

2.5.4. Clasa care incapsuleaza siruri de caractere modificabile (StringBuffer) – interfata publica

Clasa **`StringBuffer`** incapsuleaza **siruri de caractere modificabile** (*mutable*). Un sir de caractere modificabil este asemanator unui **`String`**, dar continutul lui poate fi modificat.

In orice moment el contine un anumit sir de caractere, dar **lungimea si continutul sirului pot fi modificate prin apelul anumitor metode**.

Sirurile de caractere modificabile (**`StringBuffer`**) sunt sigure din punct de vedere al programelor multifilare (*multithreaded*). Metodele clasei `StringBuffer` sunt implicit sincronizate atunci cand e necesar, astfel incat toate operatiile asupra oricarei instante se desfasoara ca si cum ar aparea intr-o ordine seriala (secventiala) consistenta cu ordinea in care apelurile sunt facute de fiecare fir (*thread*) individual implicat.

2.5.4.1. Declaratia clasei StringBuffer

Declaratiile de pachet si clasa ale clasei **`StringBuffer`** sunt urmatoarele:

```
package java.lang;

public final class StringBuffer implements java.io.Serializable, CharSequence {
  // corpul clasei StringBuffer
}
```

2.5.4.2. Constructorii clasei StringBuffer

Constructorii clasei `StringBuffer`:

`StringBuffer``()` Construieste un obiect sir de caractere modificabil (*string buffer*) fara caractere, a carui capacitate initiala este de 16 caractere.
`StringBuffer``(int length)` Construieste un obiect sir de caractere modificabil (*string buffer*) fara caractere, a carui capacitate initiala este specificata de argumentul `length`.
`StringBuffer``(String str)` Construieste un obiect sir de caractere modificabil (*string buffer*) care incapsuleaza acelasi sir de caractere ca argumentul `str` (continutul initialal *bufferului* de caractere este o copie a argumentului.

2.5.4.3. Metodele clasei StringBuffer

Declaratiile si descrierea catorva metode ale clasei **`StringBuffer`**:

`StringBuffer`	**`append`**`(boolean b)` Adauga la sirul de caractere modificabil curent reprezentarea ca sir de caractere a argumentului `boolean b`.
`StringBuffer`	**`append`**`(char c)` Adauga la sirul de caractere modificabil curent reprezentarea ca sir de caractere a argumentului `char c`.

StringBuffer	**append**(char[] str) Adauga la sirul de caractere modificabil curent reprezentarea ca sir de caractere a argumentului tablou de caractere str.
StringBuffer	**append**(char[] str, int offset, int len) Adauga la sirul de caractere modificabil curent subsirul de caractere a specificat de argumente.
StringBuffer	**append**(double d) Adauga la sirul de caractere modificabil curent reprezentarea ca sir de caractere a argumentului double d.
StringBuffer	**append**(float f) Adauga la sirul de caractere modificabil curent reprezentarea ca sir de caractere a argumentului float f.
StringBuffer	**append**(int i) Adauga la sirul de caractere modificabil curent reprezentarea ca sir de caractere a argumentului int i.
StringBuffer	**append**(long l) Adauga la sirul de caractere modificabil curent reprezentarea ca sir de caractere a argumentului long l.
StringBuffer	**append**(Object obj) Adauga la sirul de caractere modificabil curent reprezentarea ca sir de caractere a argumentului Object obj.
StringBuffer	**append**(String str) Adauga la sirul de caractere modificabil curent sirul de caractere primit ca argument.
StringBuffer	**append**(StringBuffer sb) Adauga la sirul de caractere modificabil curent sirul de caractere modificabil primit ca argument.
int	**capacity**() Returneaza capacitatea curenta a sirului de caractere modificabil curent.
char	**charAt**(int index) Returneaza caracterul indicat de argumentul index din sirul de caractere modificabil curent.
StringBuffer	**delete**(int start, int end) Elimina caracterele aflate intre indexul start si indexul end-1 din sirul de caractere modificabil curent.
StringBuffer	**deleteCharAt**(int index) Elimina caracterul aflat la pozitia specificata de argumentul index din sirul de caractere modificabil curent (reducand cu 1 lungimea sirului).
void	**ensureCapacity**(int minimumCapacity) Asigura o capacitate minima a sirului de caractere modificabil curent cel putin egala cu minimul specificat ca argument.
void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Caracterele specificate (aflate intre indexul srcBeginsi indexul srcEnd-1) sunt copiate din sirul de caractere modificabil curent in tabloul de caractere destinatie dst (incepand de la index dstBegin si pana la index dstbegin + (srcEnd-srcBegin) - 1).
int	**indexOf**(String str) Returneaza indexul minim in sirul de caractere curent la care apare subsirul de caractere specificat ca argument (str), si valoarea -1 daca nu exista acel subsir in sirul de caractere curent.

int	**indexOf**(String str, int fromIndex) Returneaza indexul minim in sirul de caractere curent, incepand cu indexul specificat ca argument (fromIndex), la care apare subsirul de caractere specificat ca argument (str), si valoarea -1 daca nu exista acel subsir in sirul de caractere curent.
StringBuffer	**insert**(int offset, boolean b) Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, reprezentarea ca sir de caractere a argumentului boolean b.
StringBuffer	**insert**(int offset, char c) Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, reprezentarea ca sir de caractere a argumentului char c.
StringBuffer	**insert**(int offset, char[] str) Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, reprezentarea ca sir de caractere a argumentului tablou de caractere str.
StringBuffer	**insert**(int index, char[] str, int offset, int len) Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, a subsirului de caractere specificat de argumente.
StringBuffer	**insert**(int offset, double d) Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, reprezentarea ca sir de caractere a argumentului double d.
StringBuffer	**insert**(int offset, float f) Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, reprezentarea ca sir de caractere a argumentului float f.
StringBuffer	**insert**(int offset, int i) Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, reprezentarea ca sir de caractere a argumentului int i.
StringBuffer	**insert**(int offset, long l) Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, reprezentarea ca sir de caractere a argumentului long l.
StringBuffer	**insert**(int offset, Object obj) Inserts Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, reprezentarea ca sir de caractere a argumentului Object obj.
StringBuffer	**insert**(int offset, String str) Insereaza in sirul de caractere modificabil curent, pe pozitia specificata de argumentul offset, sirul de caractere primit ca argument.
int	**lastIndexOf**(String str) Returneaza indexul maxim in sirul de caractere curent la care apare subsirul de caractere specificat ca argument (str), si valoarea -1 daca nu exista acel subsir in sirul de caractere curent.
int	**lastIndexOf**(String str, int fromIndex) Returneaza indexul maxim in sirul de caractere curent, incepand cu indexul specificat ca argument (fromIndex), la care apare subsirul de caractere specificat ca argument (str), si valoarea -1 daca nu exista acel subsir in sirul de caractere curent.
int	**length**() Returneaza numarul de caractere al sirului de caractere modificabil curent.
StringBuffer	**replace**(int start, int end, String str) Inlocuieste caracterele specificate (aflate intre indexul start si indexul end-1) din sirul de caractere modificabil curent cu sirul de caractere specificat ca argument (sirul de caractere modificabil fiind redimensionat daca este necesar pentru a cuprinde toate caracterele din sirul str).

`StringBuffer`	**`reverse`**`()` Inlocuieste caracterele din sirul de caractere modificabil curent cu sirul de caractere obtinut prin inversarea ordinii caracterelor.
`void`	**`setCharAt`**`(int index, char ch)` Inlocuieste caracterul specificat (prin indexul `index`) din sirul de caractere modificabil curent cu argumentul `ch`.
`void`	**`setLength`**`(int newLength)` Stabileste lungimea sirului de caractere modificabil curent.
`String`	**`substring`**`(int start)` Returneaza un nou obiect `String` care contine subsirul de caractere din sirul de caractere modificabil curent care incepe la indexul `start` si se incheie la sfarsitul sirului de caractere modificabil curent.
`String`	**`substring`**`(int start, int end)` Returneaza un nou obiect `String` care contine subsirul de caractere din sirul de caractere modificabil curent care incepe la indexul `start` si se incheie la indexul `end-1`.
`String`	**`toString`**`()` Converteste la reprezentare sir de caractere (nemodificabil) datele din sirul de caractere modificabil curent.

Sirurile de caractere modificabile (`StringBuffer`) sunt utilizate de compilator pentru a implementa operatorul + de concatenare binara a sirurilor. De exemplu, **codul**:

```
x = "a" + 4 + "c";
```

este compilat ca:

```
x = new StringBuffer().append("a").append(4).append("c").toString();
```

adica:
- **se creaza un nou sir de caractere modificabil** (`StringBuffer`), initial gol,
- **se adauga la sirul de caractere modificabil** (`StringBuffer`) reprezentarea sir de caractere (`String`) a fiecarui operand, si apoi
- **se converteste continutul sirului de caractere modificabil** (`StringBuffer`) la un sir de caractere (`String`).

In acest fel **se evita crearea temporara a mai multor obiecte `String`**.

Principalele operatii asupra unui `StringBuffer` sunt
- **adaugarea la sfarsitul sirului modificabil curent** (metoda **`append()`**) si
- **inserarea intr-o pozitie specificata in sirul modificabil curent** (metoda **`insert()`**),

al caror nume sunt supraincarcate astfel incat accepta date de orice tip.

In general, daca sb este o instanta `StringBuffer`, atunci `sb.append(x)` are acelasi efect ca `sb.insert(sb.length(), x)`.

Utilizarea metodei `insert()`:

```
StringBuffer sb = new StringBuffer("Drink Java!");
sb.insert(6, "Hot ");
System.out.println(sb.toString());
```

Rezultatul executiei programului este urmatorul:

```
Drink Hot Java!
```

Fiecare sir de caractere modificabil are o **capacitate**. Cat timp lungimea sirului de caractere continut nu depaseste capacitatea, nu este necesara alocarea unui nou tablou de caractere intern. Daca este depasita capacitatea acestui tablou, el este in mod automat largit.

Program de inversare a unui sir folosind `String` **si** `StringBuffer`**:**

```
 1    class ReverseString {
 2      public static String reverseIt(String source) {
 3         int i, len = source.length();
 4         StringBuffer dest = new StringBuffer(len);
 5
 6         for (i = (len - 1); i >= 0; i--)
 7            dest.append(source.charAt(i));
 8         return dest.toString();
 9      }
10    }
11    public class StringsDemo {
12      public static void main(String[] args) {
13         String palindrome = "Dot saw I was Tod";
14         String reversed = ReverseString.reverseIt(palindrome);
15         System.out.println(reversed);
16      }
17    }
```

Rezultatul executiei programului este urmatorul:

```
doT saw I was toD
```

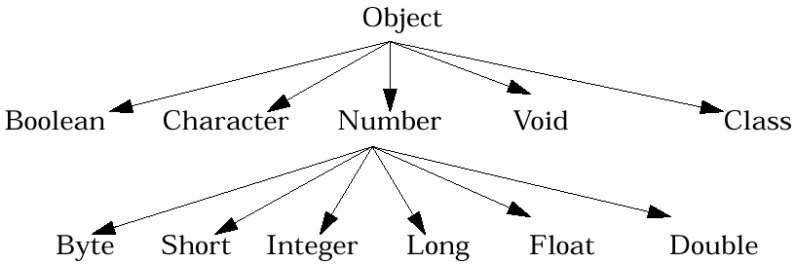
Utilizarea metodei `valueOf()`:

```
String piStr = "3.14159";
Float pi = Float.valueOf(piStr);
```

2.6. Clase predefinite pentru incapsularea tipurilor primitive. Conversii

2.6.1. Incapsularea tipurilor primitive

Ca si caracterele, si tipurile primitive numerice pot fi incapsulate in obiecte al caror continut este nemodificabil (*immutable*) dupa initializare. Iata, mai jos, o parte din ierarhia de clase Java.



Vom analiza clasa **`Integer`**, care incapsuleaza valori intregi primitive de tip **`Integer`**. Celelalte clase care incapsuleaza valori numerice primitive (**`Byte`**, **`Short`**, **`Long`**, **`Float`**, **`Double`**) si clasa care incapsuleaza valori logice primitive (**`Boolean`**) au o interfata si un mod de lucru asemanator.

2.6.2. Clasa care incapsuleaza intregi de tip int (Integer) – interfata publica

Clasa **`Integer`** permite incapsularea valorilor primitive de tip **`int`** in obiecte, dar si conversia intre intregi si alte tipuri primitive. Un obiect de tip **`Integer`** contine un singur atribut (camp) al carui tip este **`int`**.

2.6.2.1. Declaratia clasei Integer

Declaratiile de pachet si clasa ale clasei **`Integer`** sunt urmatoarele:

```
package java.lang;

public final class Integer extends Number implements Comparable {
  // corpul clasei Integer
}
```

2.6.2.2. Constructorii clasei Integer

Constructorii clasei `Integer`:

`Integer``(int value)` Construieste un nou obiect `Integer` care incapsuleaza valoarea de tip `int` specificata ca argument (`value`).
`Integer``(String s)` Construieste un nou obiect `Integer` care reprezinta valoarea de tip `int` indicata de parametrul de tip `String s`.

2.6.3.3. Metodele clasei Integer

Declaratiile si descrierea catorva metode ale clasei **`Integer`**:

`byte`	**`byteValue`**`()` Returneaza sub forma de `byte` valoarea intregului incapsulat in obiectul `Integer` curent.
`int`	**`compareTo`**`(Integer anotherInteger)` Compara numeric obiectul curent cu obiectul primit ca parametru (returnand diferenta dintre intregul incapsulat in obiectul curent si intregul incapsulat in obiectul primit ca parametru). Altfel, metoda arunca o exceptie de tipul `ClassCastException` (obiectele `Integer` fiind comparabile doar intre ele).
`int`	**`compareTo`**`(Object o)` Compara obiectul `Integer` curent cu un alt obiect. Daca obiectul argument este un `Integer` functia se comporta ca `compareTo(Integer)`.
`static Integer`	**`decode`**`(String nm)` Decodeaza un sir de caractere (care contine o valoare literala zecimala, hexazecimala sau octala) intr-un obiect `Integer`.
`double`	**`doubleValue`**`()` Returneaza sub forma de `double` valoarea intregului incapsulat in obiectul `Integer` curent.
`boolean`	**`equals`**`(Object obj)` Compara continutul obiectului curent cu continutul obiectului primit ca parametru.
`float`	**`floatValue`**`()` Returneaza sub forma de `float` valoarea intregului incapsulat in obiectul `Integer` curent.
`int`	**`intValue`**`()` Returneaza sub forma de `int` valoarea intregului incapsulat in obiectul `Integer` curent.
`long`	**`longValue`**`()` Returneaza sub forma de `long` valoarea intregului incapsulat in obiectul `Integer` curent.
`static int`	**`parseInt`**`(String s)` Analizeaza lexical argumentul sir de caractere, returnand intregul zecimal cu semn corespunzator. Metoda arunca exceptia `NumberFormatException` daca argumentul nu respecta formatul unui intreg.
`static int`	**`parseInt`**`(String s, int radix)` Analizeaza lexical argumentul sir de caractere `s`, returnand intregul cu semn corespunzator, in baza indicate de argumentul `radix`. Metoda arunca exceptia `NumberFormatException` daca argumentul nu respecta formatul unui intreg in baza indicata.
`short`	**`shortValue`**`()` Returneaza sub forma de `short` valoarea intregului incapsulat in obiectul `Integer` curent.

`static String`	**`toBinaryString`**`(int i)` Returneaza reprezentarea ca sir de caractere (literala) a argumentului intreg sub forma de intreg fara semn in baza 2.
`static String`	**`toHexString`**`(int i)` Returneaza reprezentarea ca sir de caractere (literala) a argumentului intreg sub forma de intreg fara semn in baza 16.
`static String`	**`toOctalString`**`(int i)` Returneaza reprezentarea ca sir de caractere (literala) a argumentului intreg sub forma de intreg fara semn in baza 8.
`String`	**`toString`**`()` Returneaza reprezentarea ca sir de caractere a valorii incapsulate in obiectul `Integer` curent, convertita la intreg zecimal cu semn.
`static String`	**`toString`**`(int i)` Returneaza reprezentarea ca sir de caractere a valorii intregului de tip `int` primit ca parametru, convertita la intreg zecimal cu semn.
`static String`	**`toString`**`(int i, int radix)` Returneaza reprezentarea ca sir de caractere a intregului de tip `int` primit ca parametru, interpretat in baxa indicata de argumentul `radix`.
`Static Integer`	**`valueOf`**`(String s)` Returneaza un obiect `Integer` care incapsuleaza valoarea specificata de argument, convertita la intreg zecimal cu semn. Metoda este echivalenta cu `new Integer(Integer.parseInt(s))`.
`Static Integer`	**`valueOf`**`(String s, int radix)` Returneaza un obiect `Integer` care incapsuleaza valoarea specificata de argument, interpretat in baxa indicata de argumentul `radix`. Metoda este echivalenta cu `new Integer(Integer.parseInt(s, radix))`.

Metodele clasei `Integer` permit:
- **incapsularea** valorilor intregi primitive (folosind constructori, dar si metode de clasa **`valueOf()`**),
- **comparatii** intre intregi (**`compareTo()`**, **`equals()`**, etc.),
- **conversii** la tipuri numerice primitive (**`byteValue()`**, **`doubleValue()`**, etc.),
- **conversii** la siruri de caractere (**`toString()`**, **`toBinaryString()`**, etc.),
- **conversii** ale sirurilor de caractere la valori intregi primitive (cu metoda de clasa **`parseInt()`**).

Si **celelalte clase care incapsuleaza valori numerice primitive au metode similare metodei `parseInt()`** a clasei `Integer`, cu ajutorul carora pot crea valori numerice primitive din reprezentarile sub forma de siruri de caractere ale valorilor literale. Clasa **`Byte`** are o metoda de clasa **`parseByte()`** care primeste ca parametru un sir de caractere si returneaza valoarea numerica primitiva de tip **`byte`** corespunzatoare, clasa **`Short`** are o metoda de clasa **`parseShort()`**, clasa **`Long`** are o metoda de clasa **`parseLong()`**, clasa **`Float`** are o metoda de clasa **`parseFloat()`**, clasa **`Double`** are o metoda de clasa **`parseDouble()`**, clasa **`Boolean`** are o metoda de clasa **`parseBoolean()`**.

Apelul metodei **`parseInt()`** poate genera exceptie de tip **`NumberFormatException`** in cazul in care argumentul nu are format intreg. In acest caz, **trebuie tratata exceptia** de tip **`NumberFormatException`**, definita in clasa cu acelasi nume din pachetul **`java.lang`**, cu ajutorul unui bloc de tip:

```
try { /* secventa care poate genera exceptia */
}
catch (NumberFormatException ex) {/* secventa care trateaza exceptia */
}
```

Ca in cazul programului urmator:

```
1   public class AfisareArgumenteProgramIntregi2 {
2     public static void main(String[] args) {
3       int i;
4       for ( i=0; i < args.length; i++ ) {
5         try {
6           System.out.println(Integer.parseInt(args[i]));
7         }
8         catch (NumberFormatException ex) {
9           System.out.println("Argumentul '" + args[i] +
10                                        "' nu are format numeric intreg");
11        }
12      }
13    }
14  }
```

Alte **exemple de lucru cu obiecte** de tip **`Integer`**.

```
1   int     i, j, k;      // intregi ca variabile de tip primitiv
2
3   Integer m, n, o;      // intregi incapsulati in obiecte Integer
4
5   String  s, r, t;      // siruri de caractere (incapsulate in obiecte)
6
7   // constructia intregilor incapsulati
8   // utilizand constructori ai clasei Integer - 2 variante
9
10  i = 1000;
11  m = new Integer(i);        // echivalent cu   m = new Integer(1000);
12
13  r = new String("30");
14  n = new Integer(r);        // echivalent cu   n = new Integer("30");
15
16  // constructia intregilor incapsulati
17  // utilizand metode de clasa ale clasei Integer
18
19  t = "40";
20  o = Integer.valueOf(t);    // echivalent cu   o = new Integer("40");
21
22  // conversia intregilor incapsulati la valori numerice primitive
23
24  byte  iByte = m.byteValue();         // diferit de 1000! (trunchiat)
25  short iShort = m.shortValue();       // = 1000
26  int   iInt = m.intValue();           // = 1000
27  long  iLong = m.longValue();         // = 1000L
28
29  float iFloat = m.floatValue();       // = 1000.0F
30  double iDouble = m.doubleValue();    // = 1000.0
31
32  // conversia intregilor incapsulati la obiecte sir de caractere
33
34  String iString = m.toString();       // metoda de obiect (non-statica)
35
36  // conversia valorilor intregi primitive la siruri de caractere
37
38  String douaSute = Integer.toString(200);  // metoda de clasa (statica)
39
40  String oMieBinary = Integer.toBinaryString(1000); // metoda de clasa
41  String oMieOctal = Integer.toOctalString(1000);   // metoda de clasa
42  String oMieHex = Integer.toHexString(1000);       // metoda de clasa
43
44  // conversia sirurilor de caractere la valori intregi primitive
45
46  int oSuta = Integer.parseInt("100");      // metoda de clasa (statica)
```

2.7. Clase Java pentru operatii de intrare-iesire (IO)

Programele pot avea nevoie de a:

- **prelua** informatii **de la surse** externe, sau
- **trimite** informatii **catre destinatii** externe.

Sursa/destinatia poate fi: ***fisier pe disc, retea, memorie (alt program), dispozitive IO standard (ecran, tastatura)***.

Tipurile informatiilor sunt diverse: ***caractere, obiecte, imagini, sunete.***

Pentru **preluarea** informatiilor programul **deschide un flux de la o sursa** de informatii si **citeste serial** (secvential) informatiile, astfel:

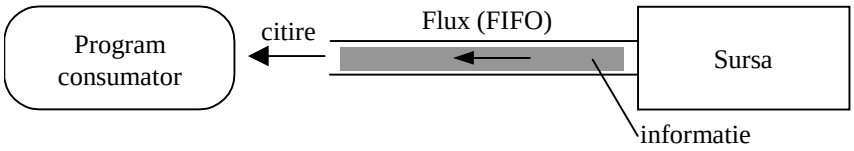


Figura 1. Fluxuri de intrare (citire) Java

Pentru **trimiterea** informatiei programul **deschide un flux catre o destinatie** de informatie si **scrie serial** (secvential) informatiile, astfel:

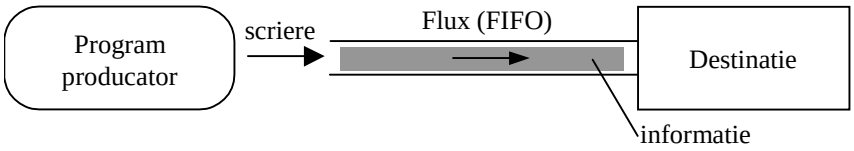


Figura 2. Fluxuri de iesire (scriere) Java

2.7.1. Clasificarea fluxurilor IO in functie de tipul de data transferate

In functie de **tipul de date** transferate, clasele din pachetul **`java.io`** se impart in**:**

- fluxuri **de caractere** (date reprezentate in UNICODE pe **16b**), avand ca radacini ale arborilor de clase derivate superclasele abstracte **`Reader`** (de intrare) si **`Writer`** (de iesire)
- fluxuri **de octeti** (date reprezentate pe **8b**), avand ca radacini ale arborilor de clase derivate superclasele abstracte **`InputStream`** (de intrare) si **`OutputStream`** (de iesire)

2.7.1.1. Constructorii claselor Reader si Writer

Constructorii clasei `Reader`:

`protected`	**`Reader`**`()` Creaza un nou flux de citire a caracterelor ale carui sectiuni critice se vor sincroniza pe fluxul insusi.
`protected`	**`Reader`**`(Object lock)` Creaza un nou flux de citire a caracterelor ale carui sectiuni critice se vor sincroniza pe obiectul specificat ca parametru.

Constructorii clasei `Writer`:

`protected`	**`Writer`**`()` Creaza un nou flux de scriere a caracterelor ale carui sectiuni critice se vor sincroniza pe fluxul insusi.
`protected`	**`Writer`**`(Object lock)` Creaza un nou flux de scriere a caracterelor ale carui sectiuni critice se vor sincroniza pe obiectul specificat ca parametru.

2.7.1.2. Metodele claselor Reader si Writer

Declaratiile si descrierea catorva metode ale clasei **`Reader`**:

`int`	**`read`**`()` Citeste din flux un caracter. Metoda se blocheaza pana cand este disponibil un caracter (caz in care returneaza caracterul ca intreg in intervalul de la 0 la 65535, adica de la `0x0000` la `0xffff`), apare o eroare I/O (caz in care arunca o exceptie `IOException`), sau este atins sfarsitul fluxului (caz in care retureaza `-1`).
`int`	**`read`**`(char[] cbuf)` Citeste din flux caractere in tabloul de caractere primit ca parametru. Metoda se blocheaza pana cand sunt disponibile caractere, apare o eroare I/O, sau este atins sfarsitul fluxului.
`abstract int`	**`read`**`(char[] cbuf, int off, int len)` Citeste din flux caractere in subtabloul de caractere specificat de parametrii primiti. Metoda se blocheaza pana cand sunt disponibile caractere, apare o eroare I/O, sau este atins sfarsitul fluxului.
`abstract void`	**`close`**`()` Inchide fluxul curent.
`void`	**`mark`**`(int readAheadLimit)` Marcheaza pozitia curenta in flux.
`boolean`	**`markSupported`**`()` Returneaza o valoare logica indicand existenta suportului pentru operatia `mark()`. Implementarea implicita returneaza intotdeauna `false`, subclasele urmand sa rescrie aceasta metoda daca se doreste sa returneze `true`.
`boolean`	**`ready`**`()` Returneaza o valoare logica indicand daca fluxul curent este pregatit pentru a fi citit (caz in care urmatorul apel `read()` sigur nu se blocheaza).
`void`	**`reset`**`()` Reseteaza fluxul. Daca fluxul a fost marcat, se incearca repozitionarea pe marcaj. Daca fluxul nu a fost marcat, se incearca resetarea lui intr-un mod potrivit fiecarui tip particular de flux, de exemplu repozitionand fluxul pe punctual de start. Nu toate fluxurile de intrare a caracterelor suporta operatia `reset()`, iar unele and suporta `reset()` fara a suporta `mark()`.
`long`	**`skip`**`(long n)` Sare peste (si elimina) numarul specificat de caractere. Metoda se blocheaza pana cand sunt disponibile caractere, apare o eroare I/O, sau este atins sfarsitul fluxului.

Declaratiile si descrierea catorva metode ale clasei **`Writer`**:

`void`	**`write`**`(int c)` Scrie in flux caracterul continut in cei mai putin semnificativi 16 biti ai intregului (de 32 de biti) primit ca parametru (sunt neglijati cei mai semnificativi 16 biti ai intregului).
`void`	**`write`**`(char[] cbuf)` Scrie in flux caracterele din tabloul de caractere primit ca parametru.
`abstract void`	**`write`**`(char[] cbuf, int off, int len)` Scrie in flux caracterele din subtabloul de caractere specificat de parametri.
`void`	**`write`**`(String str)` Scrie in flux caracterele din sirul de caractere primit ca parametru.
`void`	**`write`**`(String str, int off, int len)` Scrie in flux caracterele din subsirul de caractere specificat de parametri.
`abstract void`	**`flush`**`()` Goleste fluxul (forteaza trimiterea datelor lui catre destinatie, in cazul in care erau stocate temporar intr-un *buffer*). Daca destinatia este un alt flux, metoda `flush()` pentru fluxul curent duce la apelul metodei `flush()` pentru fluxul destinatie. Astfel, invocarea metodei `flush()`duce la golirea *bufferelor* tuturor fluxurilor inlantuite pana la destinatia finala propriu-zisa.
`abstract void`	**`close`**`()` Inchide fluxul, fortand mai intai golirea lui (*flushing*).

Toate metodele au caracter `public` si arunca exceptia `IOException`.

2.7.1.3. Constructorii claselor InputStream si OutputStream

Constructorul clasei `InputStream`:

`InputStream``()` Creaza un nou obiect al clasei `InputStream` fara a face alte initializari.

Constructorul clasei `OutputStream`:

`OutputStream``()` Creaza un nou obiect al clasei `OutputStream` fara a face alte initializari.

2.7.1.4. Metodele claselor InputStream si OutputStream

Declaratiile si descrierea catorva metode ale clasei **`InputStream`**:

`int`	**`available`**`()` Returneaza numarul de octeti care pot fi cititi (sau peste care se poate sari) din fluxul de intrare curent, fara blocare la urmatorul apel al unei metode pentru fluxul curent.

`abstract int`	**`read`**`()` Returneaza urmatorul octet de date din fluxul de intrare (returneaza valori intre `0` si `255`). Daca este detectata atingerea sfarsitului de flux, valoarea returnata este `-1`. Metoda se blocheaza pana cand sunt disponibili octeti, apare o eroare I/O, sau este atins sfarsitul fluxului.
`int`	**`read`**`(byte[] b)` Citeste un numar de octeti de date din fluxul de intrare (returneaza valori intre `0` si `255`) si ii plaseaza in tabloul de octeti primit ca parametru, incepand de la indexul `0`. Daca este detectata atingerea sfarsitului de flux, valoarea returnata este `-1`. Metoda se blocheaza pana cand sunt disponibili octeti, apare o eroare I/O, sau este atins sfarsitul fluxului.
`int`	**`read`**`(byte[] b, int off, int len)` Citeste cel mult `len` octeti de date din fluxul de intrare (returneaza valori intre `0` si `255`) si ii plaseaza in tabloul de octeti primit ca parametru, incepand de la indexul `off`. Daca este detectata atingerea sfarsitului de flux, valoarea returnata este `-1`. Metoda se blocheaza pana cand sunt disponibili octeti, apare o eroare I/O, sau este atins sfarsitul fluxului.
`long`	**`skip`**`(long n)` Sare peste (si elimina) cel mult numarul specificat de octeti (este returnat numarul exact al octetilor eliminati). Metoda se blocheaza pana cand sunt disponibili octeti, apare o eroare I/O, sau este atins sfarsitul fluxului.
`void`	**`reset`**`()` Reseteaza fluxul curent. Daca fluxul a fost marcat, se incearca repozitionarea pe marcaj. Daca fluxul nu a fost marcat, se incearca resetarea lui intr-un mod potrivit fiecarui tip particular de flux, de exemplu repozitionand fluxul pe punctual de start.
`void`	**`mark`**`(int readlimit)` Marcheaza pozitia curenta in fluxul curent.
`boolean`	**`markSupported`**`()` Returneaza o valoare logica indicand existenta suportului pentru operatia `mark()`.
`void`	**`close`**`()` Inchide fluxul curent si elibereaza toate resursele sale.

Declaratiile si descrierea catorva metode ale clasei **`OutputStream`**:

`abstract void`	**`write`**`(int b)` Scrie in flux octetul continut in cei mai putin semnificativi 8 biti ai intregului (de 32 de biti) primit ca parametru (sunt neglijati cei mai semnificativi 24 biti ai intregului).
`void`	**`write`**`(byte[] b)` Scrie in flux octetii din tabloul de octeti primit ca parametru.
`void`	**`write`**`(byte[] b, int off, int len)` Scrie in flux octetii din subtabloul de octeti specificat de parametri.
`void`	**`flush`**`()` Goleste fluxul (forteaza trimiterea datelor lui catre destinatie, in cazul in care erau stocate temporar intr-un *buffer*). Daca destinatia este un alt flux, metoda `flush()` pentru fluxul curent duce la apelul metodei `flush()` pentru fluxul destinatie.
`void`	**`close`**`()` Inchide fluxul curent si elibereaza toate resursele sale.

2.7.2. Clasificarea fluxurilor IO in functie de specializare

In functie de **specializarea pe care o implementeaza**, subclasele claselor abstracte **`Reader`**, **`Writer`**, **`InputStream`**, si **`OutputStream`** se impart in **doua categorii:**

- fluxuri **terminale** (*data sink*), care **nu au ca sursa / destinatie alte fluxuri**, ci:
  - ***fisierele,***
  - ***memoria (tablourile),***
  - ***reteaua (socketurile),***
  - ***sirurile de caractere (String),***
  - ***alte programe (prin conducte - pipes)***

- fluxuri **de prelucrare** (*processing*), care **au ca sursa / destinatie alte fluxuri**, si au ca rol prelucrarea informatiilor:
  - ***buffer-are (stocare temporara),***
  - ***filtrare de diferite tipuri (conversie, contorizare, etc.)***
  - ***tiparire.***

2.7.3. Fluxuri terminale (*data sink*)

Mai jos sunt prezentate **tipurile de fluxuri Java terminale**.

Tip de Terminal	Utilizare	Fluxuri de caractere	Fluxuri de octeti
Memorie	*Accesul secvential la tablouri*	**`CharArray`**`Reader`	**`ByteArray`**`InputStream`
		`CharArray``Writer`	**`ByteArray`**`OutputStream`
	Accesul secvential la siruri de caractere	**`String`**`Reader`	**`StringBuffer`**`InputStream`
		`String``Writer`	**`StringBuffer`**`OutputStream`
Canal / conducta (*pipe*)	*Conducte intre programe*	**`Piped`**`Reader`	**`Piped`**`InputStream`
		`Piped``Writer`	**`Piped`**`OutputStream`
Fisier	*Accesul la fisiere*	**`File`**`Reader`	**`File`**`InputStream`

2.7.3.1. Fluxul de intrare a octetilor din tablou de octeti (`ByteArrayInputStream`)

Un **`ByteArrayInputStream` contine un *buffer* (tablou de octeti) intern care contine octetii ce pot fi cititi din flux** (sursa fluxului este tabloul de octeti intern). Un contor intern este utilizat pentru a determina care este urmatorul octet ce trebuie oferit metodei `read()`.

Inchiderea unui `ByteArrayInputStream` nu are nici un efect vizibil. Metodele acestei clase pot fi apelate si dupa ce fluxul a fost inchis, fara a se genera o exceptie `IOException`.

Constructorii clasei `ByteArrayInputStream`:

`ByteArrayInputStream``(byte[] buf)` Creaza un obiect `ByteArrayInputStream` care utilizeaza `buf` ca tablou de octeti intern. Tabloul nu este copiat, ci se pastreaza o referinta interna catre el.
`ByteArrayInputStream``(byte[] buf, int offset, int length)` Creaza un obiect `ByteArrayInputStream` care utilizeaza `length` octeti din tabloul `buf` ca tablou de octeti intern, incepand de la indexul `offset`.

Declaratiile si descrierea catorva metode ale clasei **`ByteArrayInputStream`**:

`int`	**`available`**`()` Returneaza numarul de octeti care pot fi cititi (sau eliminati) din fluxul de intrare curent, fara blocare la urmatorul apel al unei metode pentru fluxul curent.
`int`	**`read`**`()` Returneaza urmatorul octet de date din fluxul de intrare (valori intre `0` si `255`).
`int`	**`read`**`(byte[] b, int off, int len)` Citeste cel mult `len` octeti de date din fluxul de intrare (returneaza valori intre `0` si `255`) si ii plaseaza in tabloul de octeti primit ca parametru, incepand de la indexul `off`.
`long`	**`skip`**`(long n)` Elimina cel mult `n` octeti (returneaza numarul exact al octetilor eliminati).
`void`	**`reset`**`()` Reseteaza fluxul curent la pozitia marcata.
`void`	**`mark`**`(int readAheadLimit)` Marcheaza pozitia curenta in fluxul curent.
`boolean`	**`markSupported`**`()` Returneaza o valoare logica indicand existenta suportului pentru operatia `mark()`.
`void`	**`close`**`()` Inchiderea unui `ByteArrayInputStream` nu are nici un effect (**!**).

2.7.3.2. Fluxul de iesire a octetilor catre tablou de octeti (`ByteArrayOutputStream`)

Un **`ByteArrayOutputStream` este un flux de iesire al octetilor care scrie datele intr-un tablou de octeti intern** (destinatia fluxului este tabloul de octeti intern). Datele pot fi regasite utilizand metodele **`toByteArray()`** (care returneaza tabloul de octeti) si `toString()`.

Inchiderea unui `ByteArrayOutputStream` nu are nici un efect vizibil. Metodele acestei clase pot fi apelate si dupa ce fluxul a fost inchis, fara a se genera o exceptie `IOException`.

Constructorii clasei `ByteArrayOutputStream`:

`ByteArrayOutputStream``()` Creaza un obiect `ByteArrayOutputStream` care utilizeaza un tablou de octeti intern de lungime initiala 32 octeti, dar a carui lungime poate creste daca este necesar.
`ByteArrayOutputStream``(int size)` Creaza un obiect `ByteArrayOutputStream` care utilizeaza un tablou de octeti intern cu lungimea initiala specificata de parametrul `size`, dar a carui lungime poate creste daca este necesar.

Atributele clasei `ByteArrayOutputStream`:

`protected byte[]`	**`buf`** Tabloul de octeti intern (*bufferul*) in care sunt stocate datele.
`protected int`	**`count`** Numarul octetilor valizi din tabloul de octeti intern.

Declaratiile si descrierea catorva metode ale clasei **`ByteArrayOutputStream`**:

`void`	**`write`**`(byte[] b, int off, int len)` Scrie in tabloul de octeti intern `len` octeti din tabloul specificat ca parametru (`b`) incepand de la indexul `off`.
`void`	**`write`**`(int b)` Scrie in tabloul de octeti intern octetul specificat ca parametru (de fapt, octetul cel mai putin semnificativ al intregului `b`).
`void`	**`writeTo`**`(OutputStream out)` Scrie intregul continut al tabloului de octeti intern catre fluxul de octeti de iesire specificat ca parametru, ca sic and s-ar utiliza apelul `out.write(buf, 0, count)`.
`void`	**`reset`**`()` Reseteaza atributul `count` la zero, astfel incat octetii acumulati in tabloul intern sunt eliminati.
`int`	**`size`**`()` Returneaza lungimea curenta a tabloului de octeti intern.
`byte[]`	**`toByteArray`**`()` Returneaza un nou tablou de octeti, copie a celui intern.
`String`	**`toString`**`()` Returneaza un sir de caractere care corespunde octetilor din tabloul de octeti intern, folosind codarea implicita a platformei.
`String`	**`toString`**`(String enc)` Returneaza un sir de caractere care corespunde octetilor din tabloul de octeti intern, folosind codarea specificata ca parametru.
`void`	**`close`**`()` Inchiderea unui `ByteArrayOutputStream` nu are nici un effect (**!**).

2.7.3.3. Lucrul cu fluxuri fisier

In continuare sunt ilustrate:

- **citirea dintr-un fisier** prin intermediul unui **flux de caractere** (Unicode!):

```
1    // Crearea unui obiect referinta la fisier pe baza numelui fisierului
2    File inputFile = new File("nume1.txt");
3
4    // Crearea unui flux de intrare a caracterelor dinspre fisierul dat
5    FileReader in = new FileReader(inputFile);
6
7    // Citirea unui caracter din fisier
8    int c = in.read();         // Input
9
10   // Inchiderea fisierului
11   in.close();
```

- **scrierea intr-un fisier** prin intermediul unui **flux de caractere**:

```
1    // Crearea unui obiect referinta la fisier pe baza numelui fisierului
2    File outputFile = new File("nume2.txt");
3
4    // Crearea unui flux de iesire a caracterelor spre fisierul dat
5    FileWrite out = new FileWriter(outputFile);
6
7    char c = 'x';
8    // Scrierea unui caracter in fisier
9    out.write(c);     // Output
10
11   // Inchiderea fisierului
12   out.close();
```

2.7.4. Fluxuri de prelucrare

Mai jos sunt prezentate **tipurile de fluxuri Java de prelucrare**.

Tip de Prelucrare	**Utilizare**	**Fluxuri de Caractere**	**Fluxuri de octeti**
Buffer-are	*Stocare temporară*	**Buffered**Reader	**Buffered**InputStream
		BufferedWriter	**Buffered**OutputStream
Filtrare	*Prelucrare*	**Filter**Reader	**Filter**InputStream
		FilterWriter	**Filter**OutpuStream
Conversie octet/caracter	*Bridge byte-char*	**InputStream**Reader	
		OutputStreamWriter	
Concatenare	*Prelucrare*		**Sequence**InputStream
Serializarea obiectelor			**Object**InputStream
			ObjectOutputStream
Conversia datelor	*Acces la tip date primitiv Java*		**Data**InputStream
			DataOutputStream
Numararea (contorizarea)	*Numarare linii*	**LineNumber**Reader	**LineNumber**InputStream
Testare	*Buffer de 1 byte/char*	**PushBock**Reader	**Pushback**InputStream
Imprimare	*Tiparire*	**Print**Writer	**Print**Stream

2.7.4.1. Fluxul de intrare a caracterelor cu stocare temporara (`BufferedReader`)

Un **`BufferedReader` citeste caracterele din fluxul din amonte** (primit ca parametru de constructor in momentul initializarii) **si le stocheaza temporar pentru a fi citite eficient** caractere, tablouri sau **linii de text**.

Constructorii clasei **`BufferedReader`**:

`BufferedReader``(Reader in)` Creaza un flux de intrare a caracterelor cu stocare temporara (si posibilitate de citire a caracterelor sub forma de linii) din fluxul de intrare a caracterelor primit ca parametru (`in`), utilizand dimensiunea implicita a tabloului intern.
`BufferedReader``(Reader in, int size)` Creaza un flux de intrare a caracterelor cu stocare temporara (si posibilitate de citire a caracterelor sub forma de linii) din fluxul de intrare a caracterelor primit ca parametru (`in`), utilizand dimensiunea specificata ca parametru pentru tabloul intern (`size`).

Metoda oferita de clasa **`BufferedReader`** pentru **citirea liniilor de text** este:

`String`	**`readLine`**`()` Citeste o linie de text (citeste din bufferul intern pana la intalnirea caracterului care semnaleaza terminarea liniei). Se blocheaza in asteptarea caracterului care semnaleaza terminarea liniei.

2.7.4.2. Fluxul de intrare convertor al octetilor la caractere (`InputStreamReader`)

Un **`InputStreamReader`** este un *bridge* (convertor) de la octeti la caractere. El citeste octetii din fluxul din amonte (primit ca parametru de constructor in momentul initializarii) si ii decodeaza folosind un set de caractere dat (`charset`). Daca nu este specificat un set anume, este folosit in mod implicit cel implicit al platformei pe care se lucreaza.

Constructorul tipic al clasei **`InputStreamReader`** este:

`InputStreamReader``(InputStream in)` Creaza un flux de intrare a caracterelor obtinute prin conversia octetilor primiti de la fluxul de intrare primit ca parametru (`in`) utilizand setul de caractere implicit.

Pentru eficienta maxima, este recomandata inlantuirea (plasarea in cascada) a unui **`InputStreamReader`** si a unui **`BufferedReader`**. De exemplu:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

O utilizare tipica a fluxurilor de prelucrare **`BufferedReader`** si **`InputStreamReader`** plasate **in cascada**, este cea care permite **citirea sirurilor de caractere de la consola standard de intrare** (tastatura, care este **incapsulata in obiectul `System.in`**, al carui tip este **`InputStream`** - flux de intrare a octetilor).

Exemplu de citire a unui nume de la tastatura:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Introduceti numele: ");
String nume = in.readLine();
```

2.7.4.3. Fluxul de iesire a caracterelor cu stocare temporara (`BufferedWriter`)

Un **`BufferedWriter` stocheaza temporar caracterele ce urmeaza a fi scrise in fluxul din aval** (primit ca parametru de constructor in momentul initializarii) **pentru a fi scrise eficient** caractere, tablouri sau **siruri de caractere**.

Constructorii clasei **`BufferedWriter`**:

`BufferedWriter``(Writer out)` Creaza un flux de iesire a caracterelor cu stocare temporara inaintea scrierii lor in fluxul de iesire a caracterelor primit ca parametru (`out`), utilizand dimensiunea implicita a tabloului intern.
`BufferedWriter``(Writer out, int size)` Creaza un flux de iesire a caracterelor cu stocare temporara inaintea scrierii lor in fluxul de iesire a caracterelor primit ca parametru (`out`), utilizand dimensiunea specificata ca parametru pentru tabloul intern (`size`).

Metoda oferita de clasa **`BufferedWriter`** pentru **scrierea separatorilor de linie de text** este:

`void`	**`newLine`**`()` Scrie un separator de linie.

2.7.4.4. Fluxul de iesire convertor al caracterelor la octeti (`OutputStreamWriter`)

Un **`OutputStreamWriter`** este un *bridge* (convertor) de la caractere la octeti. El scrie in fluxul din aval (primit ca parametru de constructor in momentul initializarii) octeti obtinuti prin codarea caracterelor, folosind un set de caractere dat (`charset`). Daca nu este specificat un set anume, este folosit in mod implicit cel implicit al platformei pe care se lucreaza.

Constructorul tipic al clasei **`OutputStreamWriter`** este:

`OutputStreamWriter``(OutputStream out)` Creaza un flux de iesire a caracterelor care vor fi convertite la octeti inaintea scrierii lor in fluxul de iesire primit ca parametru (`out`) utilizand setul de caractere implicit.

Pentru eficienta maxima, este recomandata inlantuirea (plasarea in cascada) a unui **`OutputStreamWriter`** si a unui **`BufferedWriter`**. De exemplu:

```
BufferedWriter out = new BufferedWriter (new OutputStreamWriter(System.out));
```

2.7.4.5. Fluxul de iesire a octetilor pentru afisare (`PrintStream`)

Constructorul tipic al clasei **`PrintStream`** este:

`PrintStream``(OutputStream out)` Creaza un flux de iesire a octetilor pentru afisarea lor de catre fluxul de iesire a octetilor primit ca parametru (`out`).

Metodele oferite de clasa **`PrintStream`** pentru afisarea sirurilor de caractere (scrierea intr-un flux de iesire a octetilor a sirurilor de caractere care cuprind si caractere *escape*) sunt:

`void`	**`print`**`(boolean b)` Afiseaza valoarea booleana primita ca parametru.
	… (Similar pentru celelalte tipuri de date primitive Java: char, double, float, int, …)
`void`	**`print`**`(String s)` Afiseaza sirul de caractere primit ca parametru.
`void`	**`println`**`()` Termina linia curenta, adaugand un separator de linie.
`void`	**`println`**`(boolean x)` Afiseaza valoarea booleana primita ca parametru si apoi termina linia.
	… (Similar pentru celelalte tipuri de date primitive Java: char, double, float, int, …)
`void`	**`println`**`(String x)` Afiseaza sirul de caractere primit ca parametru si apoi termina linia.

O utilizare tipica acestui fluxuri de prelucrare este cea care permite **scrierea sirurilor de caractere la consola standard de iesire** (monitorul, care este **incapsulat intr-un `OutputStream` ce serveste ca destinatie obiectului `System.out`**, al carui tip este **`PrintStream`**).

Exemplu: afisarea argumentelor programului curent:

```
PrintStream ps = System.out;
ps.println("Argumentele programului: ");
for (int i=0; i<args.length; i++) {
  ps.print(args[i] + " ");
}
ps.println();
```

2.7.4.6. Fluxul de intrare a octetilor pentru citirea valorilor primitive (`DataInputStream`)

Clasa **`DataInputStream`** permite **citirea datelor formatate (ca tipuri primitive)** de la fluxul de intrare a octetilor primit ca parametru in momentul constructiei (fluxul din amonte).

Constructorul clasei **`DataInputStream`** este:

`DataInputStream``(InputStream in)` Creaza un flux de intrare a octetilor care permite citirea datelor formatate (ca tipuri primitive) de la fluxul de intrare a octetilor primit ca parametru (`in`).

Metodele oferite de clasa **`DataInputStream`** pentru citirea datelor formatate (ca tipuri primitive) de la fluxul de intrare a octetilor primit ca parametru in momentul constructiei sunt:

`boolean`	**`readBoolean`**`()` Citeste un octet din fluxul de intrare a octetilor primit ca parametru in momentul constructiei (din amonte), si returneaza `true` daca este nenul si `false` daca este nul.
`byte`	**`readByte`**`()` Citeste si returneaza un octet.
`char`	**`readChar`**`()` Citeste si returneaza un `char`.
`double`	**`readDouble`**`()` Citeste 8 octeti si returneaza un `double`.
`float`	**`readFloat`**`()` Citeste 4 octeti si returneaza un `float`.
`void`	**`readFully`**`(byte[] b)` Citeste octetii disponibili si ii stocheaza in tabloul primit ca parametru. Metoda se blocheaza pana cand `b.length` octeti sunt disponibili.
`void`	**`readFully`**`(byte[] b, int off, int len)` Citeste `len` octeti si ii stocheaza in tabloul primit ca parametru incepand de la indexul `off`. Metoda se blocheaza pana cand `len` octeti sunt disponibili.
`int`	**`readInt`**`()` Citeste 4 octeti si returneaza un `int`.

`long`	**`readLong`**`()` Citeste 8 octeti si returneaza un `long`.
`short`	**`readShort`**`()` Citeste 2 octeti si returneaza un `short`.
`int`	**`readUnsignedByte`**`()` Citeste un octet, il extinde la tip `int` adaugand 3 octeti nuli, si returneaza rezultatul, care este in gama `0` la `255`.
`int`	**`readUnsignedShort`**`()` Citeste 2 octeti, ii extinde la tip `int` adaugand 2 octeti nuli, si returneaza rezultatul, care este in gama `0` la `65535`.
`String`	**`readUTF`**`()` Citeste un sir de caractere care a fost codat utilizand un format UTF-8 modificat.
`static` `String`	**`readUTF`**`(DataInput in)` Citeste din fluxul primit ca parametru (`in`) o reprezentare a caracterului Unicode codat in Java folosind formatul UTF-8 modificat; si returneaza ca `String` sirul de caractere rezultat.
`int`	**`skipBytes`**`(int n)` Incearca sa arunce (sa sara peste) numarul de octeti specificat ca parametru din fluxul de intrare primit ca parametru in momentul constructiei. Metoda se blocheaza pana cand `n` octeti sunt disponibili.
`String`	**`readLine()`** **Metoda nerecomandata (*deprecated*) pentru citirea sirurilor de caractere terminate cu separator de linie.**

Exemplu: citirea unui nume de la tastatura:

```
DataInputStream in = new DataInputStream(new BufferedInputStream(System.in));
System.out.println("Introduceti numele: ");
String nume = in.readLine();
```

2.7.4.7. Fluxul de iesire a octetilor pentru scrierea valorilor primitive (`DataOutputStream`)

Clasa **`DataOutputStream`** permite scrierea datelor formatate (ca tipuri primitive) in fluxul de iesire a octetilor primit ca parametru in momentul constructiei (fluxul din aval).

Constructorul clasei **`DataOutputStream`** este:

`DataOutputStream``(OutputStream out)` Creaza un flux de iesire a octetilor care permite scrierea datelor formatate (ca tipuri primitive) catre fluxul de iesire a octetilor primit ca parametru (`out`).

Metodele oferite de clasa **`DataOutputStream`** pentru scrierea datelor formatate (ca tipuri primitive) in fluxul de iesire a octetilor primit ca parametru in momentul constructiei (din aval) sunt:

void	**flush**() Forteaza trimiterea datelor scrise in acest flux de iesire catre fluxul de iesire (din aval) primit ca parametru in momentul constructiei.
void	**writeBoolean**(boolean v) Scrie valoarea booleana primita ca parametru in fluxul de iesire (din aval) primit ca parametru in momentul constructiei, ca octet.
void	**writeByte**(int v) Scrie octetul specificat ca parametru (cei mai putin semnificativi 8 biti ai argumentului v) in fluxul de iesire (din aval) primit ca parametru in momentul constructiei.
void	**writeBytes**(String s) Scrie ca secventa de octeti sirul de caractere specificat ca parametru, in fluxul de iesire din aval.
void	**writeChar**(int v) Scrie cei doi octeti ai caracterului Unicode specificat ca parametru (cei mai putin semnificativi 16 biti ai argumentului v) in fluxul de iesire din aval.
void	**writeChars**(String s) Scrie ca secventa de caractere sirul de caractere specificat ca parametru, in fluxul de iesire din aval.
void	**writeDouble**(double v) Converteste argumentul double la un long utilizand metoda doubleToLongBits() din clasa Double, apoi scrie valoarea rezultata ca 8 octeti in fluxul de iesire din aval, cel mai semnificativ octet primul.
void	**writeFloat**(float v) Converteste argumentul float la un int utilizand metoda floatToIntBits() din clasa Float, apoi scrie valoarea rezultata ca 4 octeti in fluxul de iesire din aval, cel mai semnificativ octet primul.
void	**writeInt**(int v) Scrie valoarea specificata ca parametru int ca 4 octeti in fluxul de iesire din aval, cel mai semnificativ octet primul.
void	**writeLong**(long v) Scrie valoarea specificata ca parametru long ca 8 octeti in fluxul de iesire din aval, cel mai semnificativ octet primul.
void	**writeShort**(int v) Scrie valoarea specificata ca parametru short ca 2 octeti in fluxul de iesire din aval, cel mai semnificativ octet primul.
void	**writeUTF**(String str) Scrie sirul de caractere specificat ca parametru in fluxul de iesire (din aval) primit ca parametru in momentul constructiei utilizand codarea UTF-8 modificata a Java, intr-o forma independenta de masina de calcul.

Exemplu: Afisarea argumentelor programului curent:

```
DataOutputStream dos = System.out;
dos.writeBytes("Argumentele programului: \n");
for (int i=0; i<args.length; i++) {
  dos.writeBytes(args[i] + " ");
}
dos.writeChar('\n');
dos.flush();
```

2.7.4.8. Exemple de lucru cu fluxuri IO pentru formatarea valorilor primitive

Scrierea intr-un fisier cu `DataOutputStream`:

```
DataOutputStream dos = new DataOutputStream(new FileOutputStream("invoice.txt"));

 for ( i=0;i<price.length;i++ ) {
    dos.write Double( price[i] );  //preturi
    dos.write Char( `\t` );         //tab
    dos.write Int( units[i] );      //bucati
    dos.write Char( `\t` );         //tab
    dos.write Chars( desers[i] );  //articole
    dos.write Char( `\u` );     }   //linie noua
 }
```

Continutul lui `invoice.txt` este:

```
10.5  10 ……
20.5  15 ……
…..   …. ……
```

Citirea dintr-un fisier cu `DataInputStream`:

```
DataInputStream dis = new DataInputStream(new FileInputStream ("invoice.txt"));
try { while(true){
       price = dis.readDouble( );  // pret
       dis.readChar( );            // salt peste tab
       unit = dis.readInt( );      // bucati
       dis.readChar( );            // salt peste tab
       desc = dis.readLine( );     // articol
       System.out.println(" Ati comandat" + unit + "bucati de" + desc );
       total = total + unit * price;
      }
}
catch ( EOF Exception e )
{ }
System.out.println( "Pentru un pret total de $" + total );
dis.close( );

while (( input = dis.readline( )) != null ) {
    // ……
}
```

Rezultat

```
Ati comandat 10 bucati de ….   la  $10.5
Ati comandat 15 bucati de ….   la $20.5
……..
Pentru un pret total de: $ …..
```

3. Elemente de programare Java pentru retele bazate pe IP

3.1. Introducere in Protocolul Internet (IP) si stiva de protocoale IP

3.1.1. Elemente de terminologie a retelor de comunicatie

O **retea de comunicatie** (*communication network*) este un sistem de comunicatie de forma unui graf format din sisteme intermediare cu rol de retransmisie (noduri interne) si sisteme terminale cu rol de transmitator-receptor (noduri de capat) interconectate.

O **retea de calculatoare** (*computer network*) este o varianta de sistem de comunicatie in care sistemele terminale sunt **calculatoare** (numite **masini, gazde** sau ***host-uri***).

O retea de calculatoare de arie geografica mare, **WAN** (*Wide Area Network*), interconecteaza calculatoare la nivel national sau global.
LAN (*Local Area Network*) = retea locala de calculatoare (interconecteaza calculatoare aflate la cel mult cateva zeci de metri distanta). Vitezele (debitele) de transmisie tipice in retele locale sunt 10-100 Mb/s, desi pot fi intalnite si retele la 1 Gb/s si peste.

Ethernetul este o tehnologie de retele locale cu topologie de tip bus (magistrala, cu acces concurent).

Un ***internet*** este o retea formata din (sub)retele interconectate.

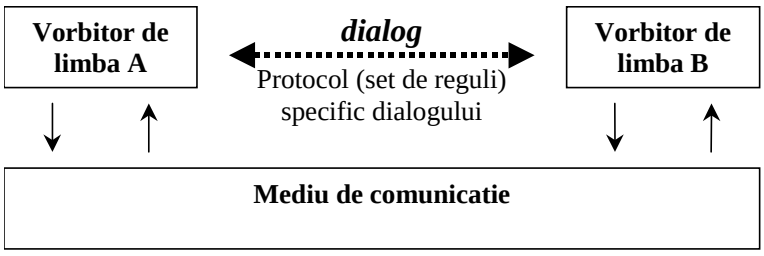
Un ***gateway*** este un element *hardware* sau *software* prin care se realizeaza conectarea subretelelor unui internet.

Internetul este reteaua de calculatoare globala.

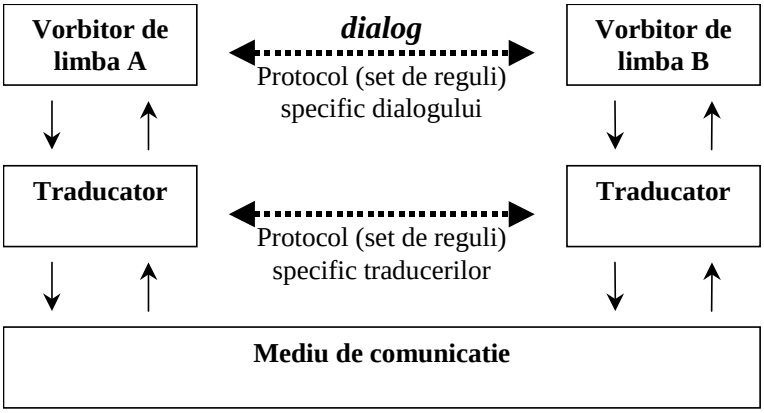
3.1.2. Modele de comunicatie stratificata

Pentru a ilustra modelul de comunicatie stratificata, bazat pe protocoale de comunicatie, consideram exemplul unui **dialog la distanta intre interlocutori de limbi diferite.**

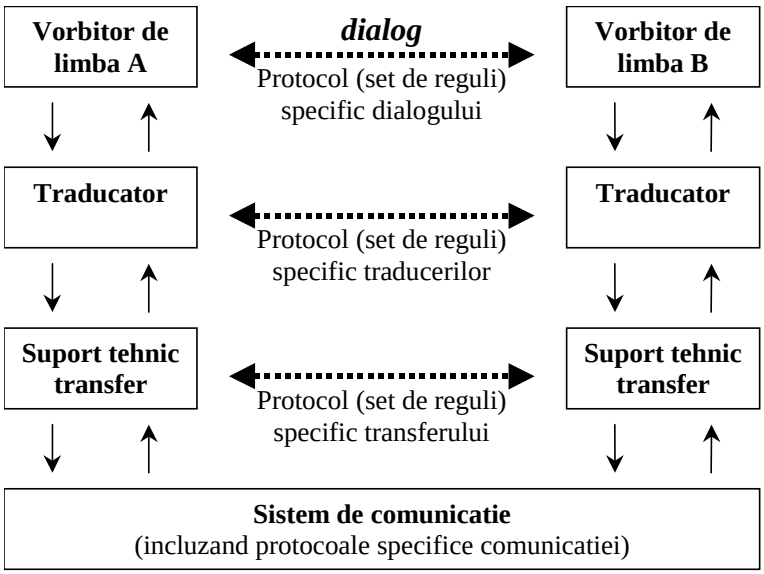
La nivelul cel mai inalt, dialogul poate fi modelat astfel:



Detaliind cazul in care **asigurarea suportului comun** (a unei limbi comune, engleza de exemplu) **prin utilizarea unor traducatori**:



Detaliind cazul in care **comunicatia la distanta include suport tehnic local si sistem de comunicatie**:



Un sistem de comunicatie stratificat utilizeaza acelasi model, bazat pe **niveluri** si **protocoale** de comunicatie.

Un **protocol** de comunicatie este un **set de reguli si conventii** care au fost **convenite intre participantii la o comunicatie** pentru a **asigura buna desfasurare a comunicatiei respective**.

Intr-un sistem de comunicatie stratificat, **interactiunile pe orizontala** se desfasoara **conform protocolului** de la respectivul nivel, interactiuni **virtuale**, realizate prin intermediul interactiunilor de la nivelurile inferioare.

Interactiunile pe verticala cu nivelul inferior au rolul de a transmite spre acest nivel **sarcini** cu scopul transferului mesajelor catre partenerul de la acelasi nivel, si de a prelua de la acest nivel **mesajele** provenite de la partenerul de la acelasi nivel.

Interactiunile pe verticala cu nivelul superior au rolul de a prelua de la acest nivel a **sarcinilor** necesare transferului mesajelor catre partenerul sau (de nivel superior), si de a transmite acestui nivel **mesajele** provenite de la partenerul sau (de nivel superior).

Un serviciu este un **set de functii (sarcini) posibil de oferit de catre nivelul inferior**, respectiv **setul de sarcini posibil de solicitat de catre nivelul superior**, in interactiunile **pe verticala**.

Protocolul este independent de continutul mesajelor schimbate in timpul comunicatiei. **Regulile pot fi diferite pentru sursa** si respectiv **destinatia** mesajelor.

Complexitatea inerenta comunicatiei este **mai usor implementata, controlata, gestionata** prin **separarea in niveluri ierarhice**.

Dualitatea **protocol-serviciu**, ca si dualitatea **implementare-interfata** din programarea bazata pe obiecte, provin din **principiul separarii preocuparilor** (***concern separation***, detaliere a principiului ***divide et impera***).

Esenta acestui principiu este: **simplificarea** (controlului) **prin separarea specificatiei functionale** (setul de functii, interfata, serviciul) **de realizarea propriu-zisa** (protocolul, implementarea).

Prin **aplicarea repetata** a acestui principiu:
- se realizeaza **descompuneri** ierarhice sau obiectuale ale caror **componente** (protocoale sau obiecte) sunt **mai usor de realizat si controlat**,
- specificatiile functionale (serviciile sau interfetele) pot fi realizate in diferite moduri (sunt **independente** de implementari),
- se obtine o **specializare modulara** a nivelurilor ierarhice respectiv obiectelor.

3.1.3. Modelul de interconectare a sistemelor deschise (modelul OSI)

Modelul OSI cuprinde **7 niveluri** ierarhice:

- **nivelul aplicatie (7)** - asigura interfete comune diferitelor aplicatii oferite utilizatorilor
- **nivelul prezentare (6)** - asigura sintaxe comune intre aplicatii sau utilizatori
- **nivelul sesiune (5)** - asigura gestiunea dialogului intre aplicatii sau utilizatori
- **nivelul transport (4)** - asigura diferite clase de transfer cap-la-cap (intre sistemele terminale utilizate de aplicatii)
- **nivelul retea (3)** - asigura transferul cap-la-cap (intre sistemele terminale ale retelei)
- **nivelul legatura de date (2)** - asigura transferul intre nodurile intermediare ale retelei
- **nivelul fizic (1)** - specifica parametrii electrici si mecanici ai comunicatiei

Pot fi identificate urmatoarele **subsisteme ierarhice**:

- **superior (subsistemul aplicatie)** – cuprinde nivelurile 7..4 OSI, include aspectele direct legate de aplicatie
- **inferior (subsistemul retea)** – cuprinde nivelurile 3..1 OSI, include aspectele direct legate de reteaua de comunicatie

Rolul nivelului transport:

- **ascunde detaliile aplicatiei** pentru retea (comunicatia propriu-zisa)
- **ascunde detaliile retelei** (comunicatiei propriu-zise) pentru aplicatie
- **cerintele specificate de aplicatie** si **oferta specificata pentru retea** sunt tratate similar (aduse la un numitor comun) la acest nivel, prin intermediul **parametrilor de calitate a serviciilor**

3.1.4. Modelul de comunicatie si protocoalele Internet

Modelul de comunicatie Internet cuprinde **4 niveluri** ierarhice:

- **aplicatie** (4 = 7..5 OSI) - corespunde subsistemului aplicatie OSI
- **transport** (3 = 4 OSI) - corespunde nivelului transport OSI
- **retea** (2 = 3 OSI) - corespunde nivelului retea OSI
- **interfata (acces) retea** (1 = 2..1 OSI) - corespunde partii inferioare a subsistemului retea OSI

Protocoale si servicii reprezentative la nivel **acces retea: Ethernet**, **Token Ring**, etc.

Protocoale si servicii reprezentative la nivel **retea** (responsabil de rutarea pachetelor Internet):

- **IP** (***Internet Protocol***) - protocol care ofera servicii Internet **fara conexiune**, asigurand transmiterea **nefiabila** a pachetelor IP intre sisteme terminale, pe baza unor adrese unice, specifice fiecarui nod, numite **adrese Internet** sau **adrese IP**; utilizat de **TCP** si **UDP**;
- **ARP** (*Address Resolution Protocol*) si **RARP** (*Reverse Address Resolution Protocol*) - protocoale care realizeaza **translatia dinamica a adreselor Internet in adrese hard unice** (de exemplu Ethernet) **din reteaua locala, si invers**, pe baza tabelelor de translatare ARP;
- **ICMP** (*Internet Control Message Protocol*) - protocol care furnizeaza mesaje de **eroare si control** pentru IP; etc.

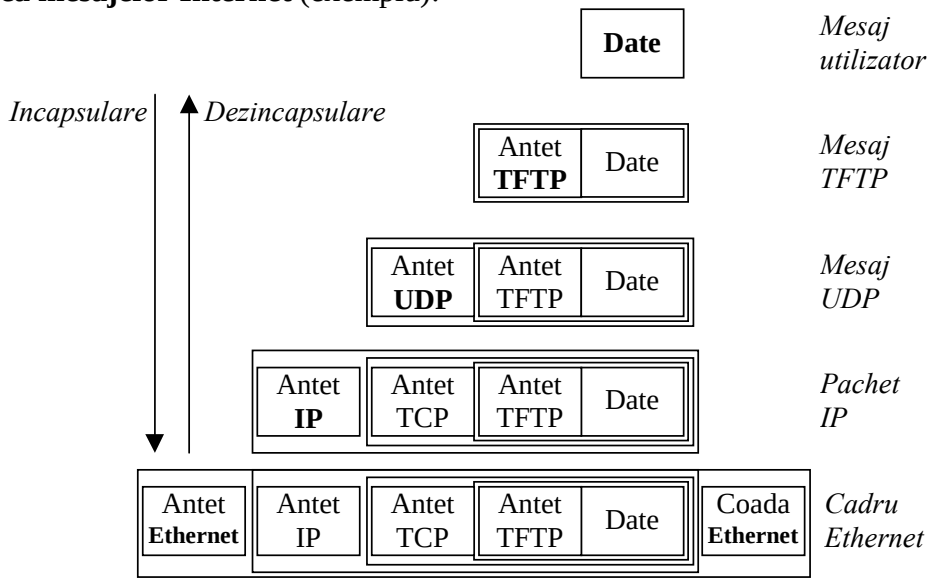
Protocoale si servicii reprezentative la nivel **transport:**

- **TCP** (*Transport Control Protocol*) - protocol care ofera servicii Internet **orientate spre conexiune** (circuite virtuale, similare sistemului **telefonic** clasic), asigurand:
  - transferul **fiabil** (sigur, fara pierderi de informatie, ordonat) al datelor intre aplicatiile sursa si destinatie,
  - in **flux continuu**,
  - **controlul fluxului** de date,
  - **multiplexarea fluxurilor** de date al mai multor procese si **controlul conexiunilor**,
- **UDP** (*User Datagram Protocol*) - protocol care ofera servicii Internet **fara conexiune**, bazate pe **transmisia independenta a fiecarui mesaj** (datagrama UDP, similara unei scrisori), asigurand:
  - transferul **nefiabil** (nesigur, cu pierderi, neordonat),
  - adaugarea unei **sume de control** la pachetele IP, si
  - **multiplexarea** fluxurilor de mesaje al mai multor procese; etc.

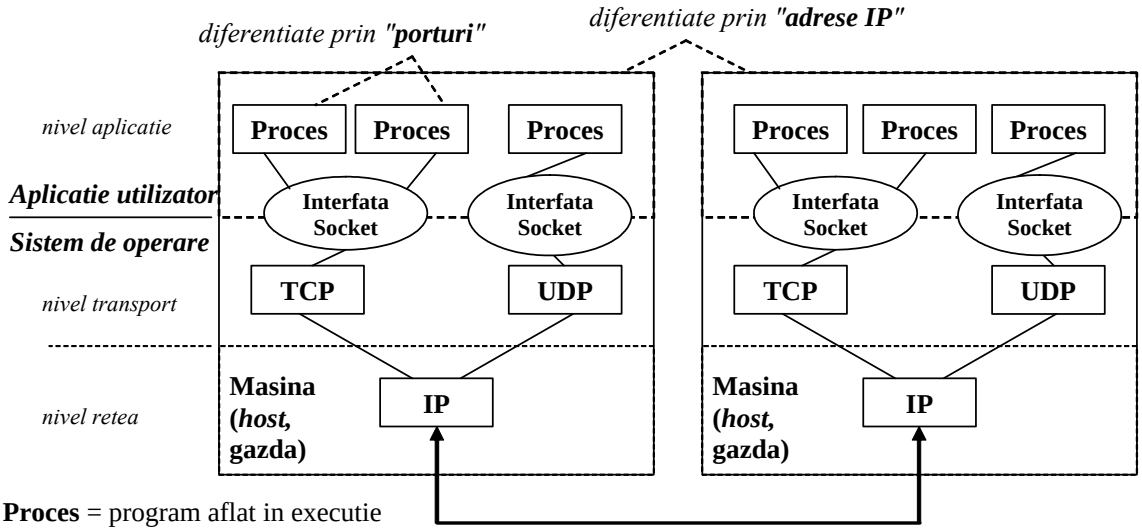
Protocoale si servicii reprezentative la nivel **aplicatie:**

- **FTP** (*File Transfer Protocol*) - protocol pentru transport de fisiere; utilizeaza **TCP**;
- **TFTP** (*Trivial File Transfer Protocol*) - protocol pentru transport de fisiere; utilizeaza **UDP**;
- **HTTP** (*Hyper-Text Transfer Protocol*) - protocol pentru transport de *hyper-pagini*; utilizeaza **TCP**;
- **Telnet** - protocol pentru transferul la distanta (pe un alt sistem terminal) al actiunilor utilizatorului; utilizeaza **TCP**; etc.
- **NFS** (*Network File System*) - protocol pentru accesul in comun al sistemelor de fisiere al terminalelor dintr-o retea locala de catre utilizatori, ca unic sistem de fisiere; utilizeaza **TCP**.

Incapsularea mesajelor Internet (exemplu):



3.1.5. Detalii utile in programare privind protocoalele Internet



Adresele IP au **32 biti** (32b), adica **4 octeti** (4B). Pentru reprezentare se folosesc diferite forme:

- **numar binar** (reprezentarea corespunzatoare stocarii in sistemele de calcul, dar cea mai ineficienta, si neutilizata de catre programator); exemplu:
  10001111 01010101 00101011 00001111
- **numar zecimal** (reprezentare neutilizata de programatorilor, desi e forma uzuala de reprezentare matematica); exemplu: 2 404 723 471
- **numar hexazecimal** (reprezentare condensata a numarului binar, neutilizata de programator); exemplu: 0x 8F 55 2B 0F
- **notatie zecimala cu puncte** ("*dotted quad*" sau "*dotted decimal*", reprezentare utilizata de programator); exemplu: 143**.**85**.**43**.**15

Clase de adrese IP:

Clasa adrese	Primii biti	Lungime ID retea (adresa retea)	Lungime ID host	Exemplu	Observatii
A	**0**	**1 B = 8 b** valori **0** .. **126**	**3 B = 32 b**	124.95.44.15	Alocate retelelor de dimensiuni **mari**, deoarece fiecare ID retea corespunde unui numar de **peste 16 milioane** ID-uri host.
B	**1 0**	**2 B = 16 b** valori **128**.000 .. **191**.255	**2 B = 16 b**	151.10.13.28	Alocate retelelor de dimensiuni **medii**, deoarece fiecare ID retea corespunde unui numar de **peste 65 000** ID-uri host
C	1 **1 0**	**3 B = 32 b** valori **192**.000.000 .. **223**.255.255	**1 B = 8 b**	201.110.213.2	Alocate retelelor de dimensiuni **mici**, deoarece fiecare ID retea corespunde unui numar de **254** ID-uri host
D, E	1 1 **1**	rezervate			

Adresele IP ale caror componente (adrese sau ID-uri) host au toti bitii egali cu 0 sunt rezervate pentru **adresarea retelei**. De exemplu, adresa de clasa A "**110**.0.0.0" contine *host-ul* "110.23.52.12". Un exemplu similar pentru clasa B poate fi "**186.10**.0.0" pentru reteaua cu adresa "**186.10**".

O facilitate asociata adreselor de retea este **difuzarea** (***broadcast-ul***) pachetelor IP catre intreaga retea de clasa A "**110**", prin trimiterea lor pe adresa "**110**.255.255.255", respectiv *broadcast-ul* pachetelor IP catre intreaga retea de clasa B "**186.10**", prin trimiterea lor pe adresa "**186.10**.255.255".

Porturile TCP si UDP sunt identificate prin numere de port de **16 biti** (16b), adica **2 octeti** (2B). Alocarea numerelor de port standard (rezervate):

- 0 .. 255 - alocate unor **protocoale de nivel aplicatie publice, standard**; de exemplu:
  - 7 - protocol ecou (*echo*)
  - 21 - FTP
  - 23 - Telnet
  - 25 - SMTP (Simple Mail Transfer Protocol)
  - 69 - TFTP
  - 80 - HTTP
- 256 .. 1023 - alocate unor **companii** de pe piata comunicatiilor;
- 1024 .. 65 535 - **nealocate**, libere pentru aplicatii Internet.

Pentru comunicatia intre doua procese care se desfasoara pe doua masini **este necesar sa se specifice**:

- adresele celor doua retele in care se afla host-urile;
- adresele celor doua host-uri in cadrul retelor lor;
- porturile la care procesele sunt conectate (prin intermediul sistemul de operare) la host-uri;
- protocolul utilizat.

Legatura dintre procese aflate pe masini (gazde, *host-uri*) diferite poarta numele de **conexiune** (*connection*). Perechea **{adresa retea, adresa host}** formeaza **adresa IP** a masinii. Specificarea unei comunicatii intre doua procese aflate pe masini diferite se poate reduce la un cvintuplu, cunoscut sub numele de **asociere** (*association*):

{ Protocol , Adresa IP sursa , Port proces sursa , Adresa IP destinatie , Port proces destinatie}

De exemplu, **pentru o conexiune TCP putem specifica asocierea**:

{ **TCP** , **143.85.43.26** , **3000** , **143.85.43.37** , **4000** }

Pentru o conexiune se pot defini doua **semiasocieri** (*half associations*):

{ Protocol , Adresa IP sursa , Port proces sursa }
{ Protocol , Adresa IP destinatie , Port proces destinatie }

A doua semiasociere poate fi folosita pentru **specificarea destinatiei unor datagrame UDP.** De exemplu:

{ **UDP** , **143.85.43.37** , **4000** }

Ca alternativa la adresele numerice IP exista **numele date host-urilor IP** (sub forma de siruri de caractere), mult mai naturala pentru utilizatori. De exemplu, astfel de nume de host poate fi "**host1.elcom.pub.ro**" unde sectiunea "**elcom.pub.ro**" corespunde domeniului asociat retelei locale, iar sectiunea "**host1**" specifica host-ul in cadrul retelei locale. Adresele complete ale utilizatorilor (de e-mail de exemplu) provin din acestea (de exemplu "user1@host1.elcom.pub.ro").

Asocierile adreselor numerice IP cu numele date host-urilor IP se face la nivelul sistemului de operare, si pot fi obtinute de catre utilizator in mai multe moduri (care fac apel la **serverele domeniilor de nume - DNS** - din Internet).

De exemplu, sub Unix dar si alte sisteme de operare, serviciul **nslookup** permite interogarea DNS pentru obtinerea atat a adresei in forma numerica (*dotted quad*) cat si a numelui de host, atunci cand este furnizata fie adresa in forma numerica fie numele host-ului. Prin apelul comenzii **nslookup** cu parametru "**host1.elcom.pub.ro**" se poate obtine ca raspuns o pereche de genul "**host1.elcom.pub.ro**", "**143.85.41.121**". In reteaua locala (in acest exemplu corespunzatoare domeniului "**elcom.pub.ro**"), se poate obtine acelasi rezultat apeland **nslookup** cu parametru "**host1**". De asemenea, apeland **nslookup** cu parametru "**143.85.41.12**" se poate obtine ca raspuns o pereche de genul "**host1.elcom.pub.ro**", "**143.85.41.121**".

3.1.6. Introducere in *socket-uri*

Socket-ul este un **punct final al unei comunicatii intre procese**, care ofera un punct de acces la servicii de nivel transport (TCP sau UDP) in Internet. Pe de alta parte, *socket-ul* este o resursa alocata de sistemul de operare, mentinuta de sistemul de operare, si accesibila utilizatorilor prin intermediul unui intreg numit descriptor de (fisier) *socket*.

Intr-o conexiune intre procese, ***socket-ul*** corespunde unei semiasocieri.

Tipurile de socket oferite de sistemele de operare actuale corespund celor doua clase mari de servicii de comunicatie:

- ***socket-uri* flux** (*stream*) - pentru **servicii orientate spre conexiune** (**TCP**) si
- ***socket-uri* datagrama** (*datagram*) - pentru **servicii fara conexiune** (**UDP**).

***Socket-urile* Berkeley** (create la Universitatea cu acelasi nume din California) sunt primele protocoale punct–la–punct pentru comunicatii pe baza de stiva TCP/IP. *Socket-urile* au fost introduse in 1981, ca **interfata generica** Unix BSD 4.2 **care suporta comunicatii interproces** (IPC = *interprocess communications*) **Unix–la–Unix**.

Astazi, *socket-urile* sunt suportate de orice sistem de operare. API-ul *Windows* pentru *socket-uri*, cunoscut ca *WinSock* (mai nou *WinSock2*)este o specificatie care standardizeaza folosirea TCP/IP sub sistemul de operare *Windows*. *In sistemele Unix BSD*, *socket-urile* fac parte din nucleu; ele ofera si servicii de sine statoare si de comunicatii intre procese. *Sistemele non – Unix BSD*, MS – DOS, Windows, MacOS, and OS/2 ofera *socket-urile* sub forma unor biblioteci.

Java ofera *socket-urile* ca parte a unei biblioteci de clase standard, `java.net`.

Practic, *socket-urile* ofera standardele portabile curente *de facto* pentru furnizorii de aplicatii pentru retele pe retelele TCP/IP.

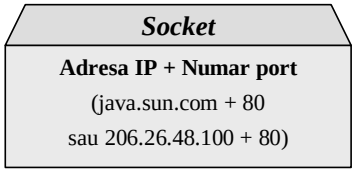
***Socket-urile* datagrame** ofera o interfata la protocolul Internet de transport prin datagrame **UDP** (*User Datagram Protocol*). UDP se ocupa de transmisile prin retea sub forma de **pachete independente** (datagrame) si nu ofera garantii privind vreunul dintre parametrii QoS (rata, intarziere, jitter, pierderi, etc.). UDP nu include verificari de sume, nu incearca sa detecteze pachetele duplicate sau sa mentina orice forma de secventialitate a transmisiunilor multipachet. Protocolul nu prevede nici o confirmare a primirii pachetelor. Informatiile pot fi pierdute, duplicate sau primite in ordine gresita. Aplicatia care utilizeaza UDP-ul este raspunzatoare de retransmisii, cand se petrece o eroare. Un avantaj este ca datagramele se deplaseaza repede si introduc un *overhead* mic. Un dezavantaj este ca datagramele impun o limita a datelor transferate (tipic 64 KB), si necesita ca aplicatia sa-i ofere toate mecanismele de asigurare a fiabilitatii.

***Socket-urile* flux** ofera o interfata la protocolul Internet de transport orientat spre conexiune **TCP** (*Transport Control Protocol*). TCP ofera un serviciu bazat pe sesiune care include controlul fluxului, reasamblarea pachetelor si mentinerea conexiunii. Socket-urile flux garanteaza ca pachetele se trimit fara erori sau duplicari si ca sunt primite in aceeasi ordine in care s-au transmis. Nu este impusa nici o limita informatiei; se considera a fi un **flux de biti**. Ca avantaj, TCP ofera un mecanism fiabil punct–la–punct. Ca dezavantaj, TCP este mai incet decat UDP si necesita mai mult *overhead* de programare.

3.2. Incapsularea adreselor IP in limbajul Java

3.2.1. Incapsularea adreselor IP

O **adresa socket** pe o retea bazata pe IP consta din doua parti: o **adresa IP** si o adresa de port (**numar de port**):



Socket-ul = Adresa IP + Numarul de port

O adresa de Internet (adresa IP) este un numar de **32 biti** (4 octeti), de obicei reprezentat ca un sir de 4 valori numerice intre 0 si 255 despartite prin puncte (de exemplu **206.26.48.100**). Adresa IP mai poate fi reprezentata prin numele domeniului (de exemplu **java.sun.com**).

Un port este un **punct de intrare (dinspre retea) intr-o aplicatie** care se afla pe o masina gazda. El este reprezentat de un numar pe **16 biti** (2 octeti).

3.2.2. Clasa java.net.InetAddress – interfata publica

Pachetul **`java.net`** foloseste clasa **`InetAddress`** pentru a incapsula o adresa IP. In continuare sunt **detaliate cateva dintre metodele** declarate in clasa **`InetAddress`**:

`boolean`	**`equals`**`(Object obj)` Compara obiectul curent (`this`) cu obiectul primit ca parametru (`obj`).
`byte[]`	**`getAddress`**`()` Returneaza adresa IP incapsulata in obiectul caruia i se aplica, sub forma numerica (*raw IP address*). Cei 4 octeti returnati sunt in ordine NBO (*network byte order*), adica octetul de ordin maxim al adresei poate fi gasit in `getAddress()[0]`.
`static InetAddress`	**`getByAddress`**`(byte[] addr)` Returneaza obiectul `InetAddress` care incapsuleaza adresa IP numerica pasata.
`static InetAddress`	**`getByName`**`(String host)` Returneaza obiectul `InetAddress` care incapsuleaza adresa IP a gazdei a carui nume i-a fost pasat (numele gazdei poate fi un nume de masina, de exemplu **`java.sun.com`**, sau adresa IP numerica).
`String`	**`getHostAddress`**`()` Returneaza adresa IP incapsulata in obiectul caruia i se aplica, sub forma de sir de caractere (obiect `String`).
`String`	**`getHostName`**`()` Returneaza numele gazdei a carei adresa IP este incapsulata in obiectul curent.
`static InetAddress`	**`getLocalHost`**`()` Returneaza obiectul `InetAddress` care incapsuleaza adresa IP locala.
`String`	**`toString`**`()` Converteste adresa IP curenta la `String`.

Clasa **`InetAddress` incapsuleaza o adresa IP** intr-un obiect care **poate intoarce informatia utila**. Aceasta informatie utila se obtine invocand metodele unui obiect al acestei clase. De exemplu, metoda **`equals()`** intoarce adevarat daca doua obiecte reprezinta aceeasi adresa IP.

Clasa `InetAddress` nu are constructor public. De aceea, **pentru a crea obiecte ale acestei clase trebuie invocata una dintre metodele** de clasa **`getLocalHost()`** si **`getByName()`**.

Codul:

```
byte[] octetiAdresaServer = { 200, 26, 48, 100 };
InetAddress adresaServer = InetAddress.getByAddress(octetiAdresaServer);
```

este echivalent cu:

```
String numeMasinaServer = "java.sun.com";
InetAddress adresaServer = InetAddress.getByName(numeMasinaServer);
```

si cu:

```
String adresaIPMasinaServer = "200.26.48.100";
InetAddress adresaServer = InetAddress.getByName(adresaIPMasinaServer);
```

Pentru a obtine **obiectul `InetAddress` care incapsuleaza adresa IP locala** poate fi folosit apelul:

```
InetAddress.getLocalHost()
```

Urmatorul program **afiseaza informatii privind masina locala**.

```
 1  import java.net.*;
 2
 3  class InfoLocalHost {
 4    public static void main (String args[]) {
 5      try {
 6
 7        InetAddress adresaLocala = InetAddress.getLocalHost();
 8
 9        System.out.println("Numele meu este " +
10                                 adresaLocala.getHostName());
11
12        System.out.println("Adresa mea IP este " +
13                                 adresaLocala.getHostAddress());
14
15        System.out.println("Octetii adresei mele IP sunt {" +
16            adresaLocala.getAddress()[0] + ", " +
17            adresaLocala.getAddress()[1] + ", " +
18            adresaLocala.getAddress()[2] + ", " +
19            adresaLocala.getAddress()[3] + "}");
20      }
21      catch (UnknownHostException e) {
22        System.out.println("Imi pare rau. Nu imi cunosc numele.");
23      }
24    }
25  }
```

O **adresa IP speciala** este **adresa IP *loopback*** (tot ce este trimis catre aceasta adresa IP se intoarce si devine intrare IP pentru gazda locala), **cu ajutorul careia pot fi testate local programe care utilizeaza *socket-uri***.

Numele "**`localhost`**" si valoarea numerica "**`127.0.0.1`**" sunt folosite pentru a identifica adresa IP *loopback*.

Pentru a obtine **obiectul `InetAddress` care incapsuleaza adresa IP *loopback*** pot fi folosite apelurile:

```
InetAddress.getByName(null)
InetAddress.getByName("localhost")
InetAddress.getByName("127.0.0.1")
```

Metoda `getAddress()` returneaza octetii adresei IP incapsulate, ceea ce poate fi util pentru filtrarea adreselor.

Urmatorul program permite **obtinerea si afisarea informatiilor privind o masina specificata**, inclusiv **clasa adresei IP**.

```
 1  import java.net.*;
 2  import java.io.*;
 3  import javax.swing.*;
 4
 5  public class InformatiiMasinaIP {
 6    public static void main(String args[]) {
 7      String nume;
 8      do {
 9        nume = JOptionPane.showInputDialog
10                    ("Introduceti un nume de masina sau o adresa IP");
11        if (nume != null)
12          afisareaInformatiilor(nume);
13      } while (nume != null);
14    }
15
16    static char clasaIP(byte[] octetiAdresa) {
17      int octetSuperior = 0xff & octetiAdresa[0];
18      return (octetSuperior < 128) ? 'A' : (octetSuperior < 192) ? 'B' :
19        (octetSuperior < 224) ? 'C' : (octetSuperior < 240) ? 'D' : 'E';
20    }
21
22    static void afisareaInformatiilor(String nume) {
23      try {
24        InetAddress masina = InetAddress.getByName (nume);
25        System.out.println ("Numele masinii: " + masina.getHostName());
26        System.out.println ("IP-ul masinii: " + masina.getHostAddress());
27        System.out.println ("Clasa masinii: " +
28                                      clasaIP(masina.getAddress()));
29      } catch (UnknownHostException ex) {
30        System.out.println("Nu am reusit sa gasesc " + nume);
31      }
32    }
33  }
```

Programul urmator contine o metoda de clasa (globala, utilitara) **`afisareInetAddress()`**, pentru **obtinerea si afisarea informatiilor privind o masina specificata**.

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5   *  Afisare informatii privind adresa IP
 6   */
 7  public class InfoInetAddress {
 8
 9    /**
10     *  Afiseaza informatii privind adresa IP specificata ca parametru
11     */
12    public static void afisareInetAddress(InetAddress ia, String nume) {
13      System.out.println("\n -------------------------------------");
14      System.out.println("\n Informatii privind adresa IP " + nume);
15      System.out.println();
16
17      System.out.println("\t Adresa IP ca text: " +
18              ia.getHostAddress());
19
20      System.out.println("\t Numele masinii: " +
21              ia.getHostName());
22
23      System.out.println("\t Numele calificat: " +
24              ia.getCanonicalHostName());
25
26      System.out.println("\t Octetii adresei IP: " +
27              ia.getAddress()[0] + " " + ia.getAddress()[1] + " " +
28              ia.getAddress()[2] + " " + ia.getAddress()[3]);
29
30      System.out.println("\t Obiectul InetAddress convertit la String: "+
31              ia.toString());
32    }
33
34    /**
35     *  Metoda principala cu rol de test
36     *  si exemplificare a modului de utilizare
37     */
38    public static void main (String args[]) {
39
40      InetAddress adresaLocala;
41
42      try {
43        adresaLocala = InetAddress.getLocalHost();
44      }
45      catch (UnknownHostException ex) {
46        System.err.println("Nu poate fi obtinuta adresa locala");
47      }
48
49      afisareInetAddress(adresaLocala, "adresa locala");
50
51      afisareInetAddress(adresaLocala, adresaLocala.toString());
52
53      afisareInetAddress(adresaLocala, null);
54
55      afisareInetAddress(null, null);
56    }
57  }
```

Metoda **`main()`** a programului ilustreaza modul de lucru cu metoda **`afisareInetAddress()`** pentru afisarea informatiilor privind masina locala.

3.3. Socket-uri flux (TCP)

3.3.1. Lucrul cu socket-uri flux (TCP)

Java ofera, in pachetul **`java.net`**, mai multe clase pentru lucrul cu *socket-uri* flux (TCP). Urmatoarele **clase Java** sunt **implicate in realizarea conexiunilor TCP** obisnuite: `ServerSocket` si `Socket`.

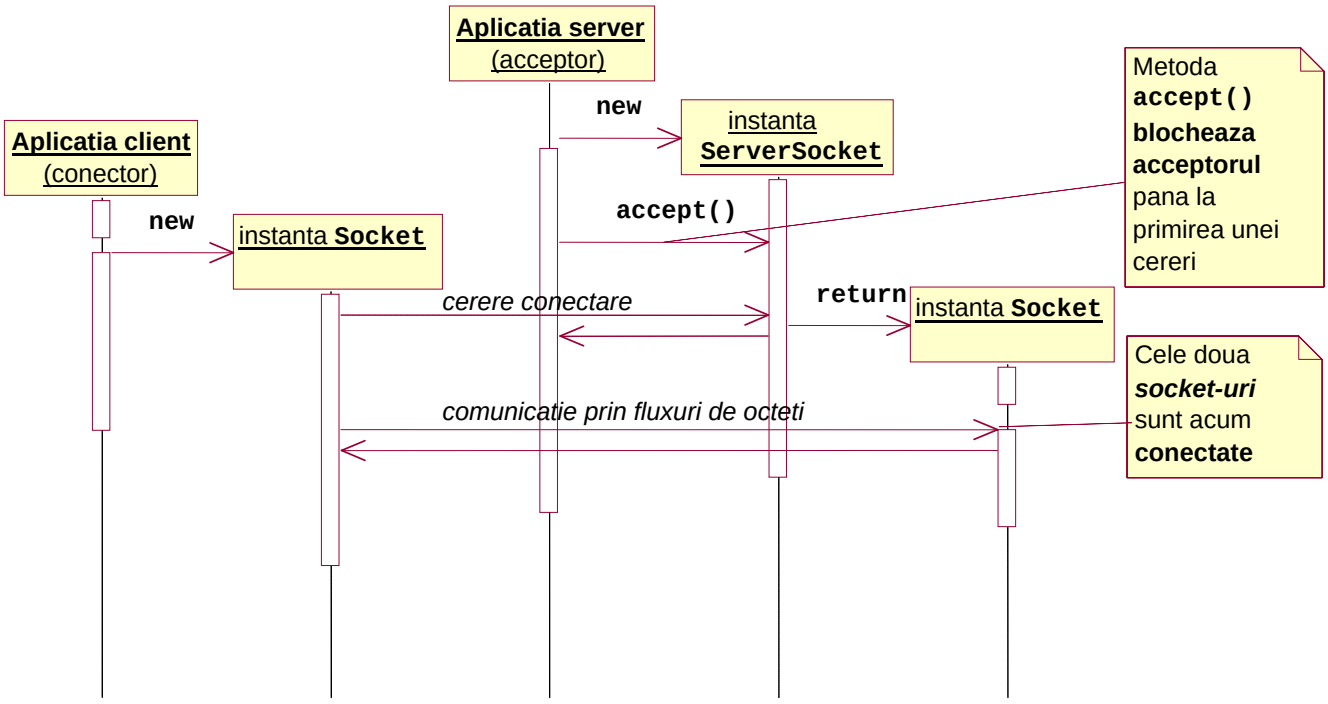
Clasa **`ServerSocket`** reprezinta ***socket-ul*** (aflat eventual pe un server bazat pe TCP) **care asteapta si accepta cereri de conexiune** (eventual de la un client bazat pe TCP).

Clasa **`Socket`** reprezinta **punctul terminal al unei conexiuni TCP** intre doua masini (eventual un client si un server).

Clientul (sau, mai general, **masina conector**) **creeaza un punct terminal `Socket`** in momentul in care cererea sa de conexiune este lansata si acceptata.

Serverul (sau, mai general, **masina acceptor**) **creeaza un `Socket`** in momentul in care primeste si accepta o cerere de conexiune, **si continua sa asculte si sa astepte alte cereri pe `ServerSocket`**.

Secventa tipica a mesajelor schimbate intre client si server este urmatoarea:



Odata conexiunea stabilita, metodele `getInputStream()` si **`getOutputSteam()`** ale clasei **`Socket`** trebuie **utilizate pentru a obtine fluxuri de octeti**, de intrare respectiv iesire, pentru comunicatia intre aplicatii.

3.3.2. Clasa socket flux (TCP) pentru conexiuni (Socket) – interfata publica

1. Principalii constructori ai clasei `Socket`:

`Socket``()` Creaza un socket flux neconectat, cu implementarea (`SocketImpl`) implicita platformei.
`Socket``(String host, int port)` Creaza un socket flux si il conecteaza la portul de numar specificat, la masina a carui nume este specificat.
`Socket``(InetAddress address, int port)` Creaza un socket flux si il conecteaza la portul de numar specificat, la adresa IP specificata.
`Socket``(String host, int port, InetAddress localAddr, int localPort)` Creaza un socket flux si il conecteaza la portul de numar specificat, la masina a carui nume este specificat. Socket-ul se va si lega (*bind*) la portul local specificat, la adresa IP locala specificata.
`Socket``(InetAddress address, int port, InetAddress localAddr, int localPort)` Creaza un socket flux si il conecteaza la portul de numar specificat, la adresa IP specificata. Socket-ul se va si lega (*bind*) la portul local specificat, la adresa IP locala specificata.

2. Declaratiile si descrierea catorva metode ale clasei **`Socket`**:

`void`	**`close`**`()` Inchide *socket-ul* curent.
`InetAddress`	**`getInetAddress`**`()` Returneaza un obiect care incapsuleaza adresa IP la care este conectat *socket-ul* curent.
`InputStream`	**`getInputStream`**`()` Returneaza un flux de intrare a octetilor dinspre *socket-ul* curent.
`boolean`	**`getKeepAlive`**`()` Testeaza (returneaza true daca este validata) optiunea SO_KEEPALIVE.
`InetAddress`	**`getLocalAddress`**`()` Returneaza un obiect care incapsuleaza adresa IP locala la care *socket-ul* curent este legat.
`int`	**`getLocalPort`**`()` Returneaza numarul portului local la care *socket-ul* curent este legat.
`OutputStream`	**`getOutputStream`**`()` Returneaza un flux de iesire a octetilor catre socket-ul curent.
`int`	**`getPort`**`()` Returneaza numarul portului la care *socket-ul* curent este conectat.
`int`	**`getReceiveBufferSize`**`()` Returneaza valoarea optiunii SO_RCVBUF pentru *socket-ul* curent, adica dimensiunea *buffer-ului* utilizat de platforma pentru fluxul de intrare dinspre *socket*.
`int`	**`getSendBufferSize`**`()` Returneaza valoarea optiunii SO_SNDBUF pentru *socket-ul* curent, adica dimensiunea *buffer-ului* utilizat de platforma pentru fluxul de iesire catre *socket*.
`boolean`	**`getTcpNoDelay`**`()` Testeaza (returneaza true daca este validata) optiunea SO_KEEPALIVE (algoritmul Nagle, de confirmare pozitiva, este activat).

`int`	**`getSoTimeout`**`()` Returneaza valoarea optiunii SO_TIMEOUT pentru *socket-ul* curent, adica durata in milisecunde cat apelul **`read()`** asupra fluxului de intrare asociat *socket-ului* curent blocheaza aplicatia, asteptand sosirea unor octeti prin flux. Valoarea 0 semnifica asteptare (blocare) la infinit. Valoarea nu poate fi negativa. La expirarea unei valori nenule a *time-out-ului*, este generata o exceptie **`java.net.SocketTimeoutException`**, dar *socket-ul* curent continua sa functioneze.
`int`	**`getTrafficClass`**`()` Obtine clasa de trafic (*traffic class*) sau tipul de serviciu (ToS) din antetul IP al pachetelor trimise prin *socket-ul* curent.
`boolean`	**`isClosed`**`()` Returneaza true daca *socket-ul* curent este inchis, altfel returneaza false.
`boolean`	**`isConnected`**`()` Returneaza true daca *socket-ul* curent este conectat cu succes, altfel returneaza false.
`boolean`	**`isInputShutdown`**`()` Returneaza true daca fluxul de intrare al *socket-ului* curent este "la sfarsit de flux" (EOF), altfel returneaza false.
`boolean`	**`isOutputShutdown`**`()` Returneaza true daca fluxul de iesire al *socket-ului* curent este valid, altfel returneaza false.
`void`	**`setKeepAlive`**`(boolean on)` Valideaza/invalideaza optiunea SO_KEEPALIVE.
`void`	**`setReceiveBufferSize`**`(int size)` Stabileste valoarea optiunii SO_RCVBUF pentru *socket-ul* curent, adica dimensiunea *buffer-ului* utilizat de platforma pentru fluxul de intrare dinspre *socket*.
`void`	**`setSendBufferSize`**`(int size)` Stabileste valoarea optiunii SO_SNDBUF pentru *socket-ul* curent, adica dimensiunea *buffer-ului* utilizat de platforma pentru fluxul de iesire catre *socket*.
`void`	**`setSoTimeout`**`(int timeout)` Stabileste valoarea optiunii SO_TIMEOUT pentru *socket-ul* curent, adica durata in milisecunde cat apelul **`read()`** blocheaza aplicatia, asteptand sosirea unor octeti prin fluxul de intrare. Valoarea 0 semnifica asteptare (blocare) la infinit. Valoarea nu poate fi negativa. La expirarea unei valori nenule a *time-out-ului*, este generata o exceptie **`java.net.SocketTimeoutException`**, dar *socket-ul* curent continua sa functioneze.
`void`	**`setTcpNoDelay`**`(boolean on)` Valideaza/invalideaza optiunea TCP_NODELAY (Activeaza/inactiveaza algoritmul Nagle).
`void`	**`setTrafficClass`**`(int tc)` Stabileste clasa de trafic (*traffic class*) sau tipul de serviciu (ToS) din antetul IP al pachetelor trimise prin *socket-ul* curent.
`void`	**`shutdownInput`**`()` Plaseaza fluxul de intrare al *socket-ului* curent "la sfarsit de flux" (EOF).
`void`	**`shutdownOutput`**`()` Invalideaza fluxul de iesire al *socket-ului* curent (implicit este valid).
`String`	**`toString`**`()` Returneaza un `String` continand informatii privind *socket-ul* curent.

In continuare este prezentata **secventa tipica pentru crearea *socket-ului* unei aplicatii conector (client)**:

```
// Stabilirea adresei serverului
String adresaServer = "localhost";

// Stabilirea portului serverului
int portServer = 2000;

// Crearea socketului (implicit este realizata conexiunea cu serverul)
Socket socketTCPClient = new Socket(adresaServer, portServer);
```

ca si pentru **crearea fluxurilor de octeti asociate *socket-ului***:

```
// Obtinerea fluxului de intrare octeti TCP
InputStream inTCP = socketTCPClient.getInputStream();

// Obtinerea fluxului de intrare caractere dinspre retea
InputStreamReader inTCPCaractere = new InputStreamReader(inTCP);
// Adaugarea facilitatilor de stocare temporara
BufferedReader inRetea = new BufferedReader(inTCPCaractere);

// Obtinerea fluxului de iesire octeti TCP
OutputStream outTCP = socketTCPClient.getOutputStream();

// Obtinerea fluxului de iesire spre retea,
// cu facilitate de afisare (similare consolei standard de iesire)
PrintStream outRetea  = new PrintStream(outTCP);
```

Socket-ul poate fi utilizat pentru **trimiterea de date**:

```
// Crearea unui mesaj
String mesajDeTrimis = "Continut mesaj";

// Scrierea catre retea (trimiterea mesajului)
outRetea.println(mesajDeTrimis);

// Fortarea trimiterii
outRetea.flush();
```

sau pentru **primirea de date**:

```
// Citirea dinspre retea (receptia unui mesaj)
String mesajPrimit = inRetea.readLine();

// Afisarea mesajului primit
System.out.println(mesajPrimit);
```

Dupa utilizare, *socket-ul* poate fi **inchis**:

```
// Inchiderea socketului (implicit a fluxurilor TCP)
socketTCPClient.close();
```

Metoda **`setTrafficClass`**`(int tc)` stabileste clasa de trafic (*traffic class*) sau tipul de serviciu (ToS) din antetul IP al pachetelor trimise prin *socket-ul* curent. **Deoarece implementarea nivelului retea poate ignora valoarea parametrului `tc`** (in cazul in care **nu exista sau nu este activat controlul de trafic pentru servicii diferentiate**), **aplicatia trebuie sa interpreteze apelul ca pe o sugestie** (*hint*) data nivelului inferior.

Pentru protocolul IPv4 valoarea parametrului **`tc`** este interpretata drept **campurile precedenta si ToS**. **Campul ToS este creat ca set de biti** obtinut prin **aplicarea functiei logice OR valorilor**:

`IPTOS_LOWCOST = 0x02` - indicand cerinte de **cost redus** din partea aplicatiei
`IPTOS_RELIABILITY = 0x04` - indicand cerinte de **fiabilitate** din partea aplicatiei
`IPTOS_THROUGHPUT = 0x08` - indicand cerinte de **banda larga** din partea aplicatiei
`IPTOS_LOWDELAY = 0x10` - indicand cerinte de intarziere redusa (**timp real**) ale aplicatiei

Cel mai putin semnificativ bit al octetului **`tc`** e ignorat, el **trebuind sa fie zero** (*must be zero* – MZB). **Stabilirea bitilor in campul de precedenta** (cei mai semnificativi 3 biti ai octetului **`tc`**) poate produce o **`SocketException`**, indicand imposibilitatea realizarii operatiei.

Pentru protocolul IPv6 valoarea parametrului **`tc`** este plasata in **campul `sin6_flowinfo`** al antetului IP.

In continuare este prezentata o **aplicatie conector (client)**, care permite **obtinerea si afisarea informatiilor privind *socket-ul* creat**. Programul face apel la metoda globala, utilitara, `afisareInetAddress()`, a clasei `InfoInetAddress` (anterior creata) pentru obtinerea si afisarea informatiilor privind o masina specificata.

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4   *  Afisare informatii privind conexiunea TCP
 5   */
 6  public class InfoSocketTCP {
 7  /**
 8   *  Afiseaza informatii privind conexiunea TCP specificata ca parametru
 9   */
10    public static void afisareInfoSocketTCP (Socket socketTCP) {
11      System.out.println("\nConexiune de la adresa " + socketTCP.getLocalAddress() +
12                         " de pe portul " + socketTCP.getLocalPort() +
13                         " catre adresa " + socketTCP.getInetAddress() +
14                         " pe portul " + socketTCP.getPort() + "\n");
15    }
16    /**
17     *  Metoda principala cu rol de test si exemplificare a modului de utilizare
18     */
19    public static void main (String args[]) {
20      BufferedReader inConsola  = new BufferedReader(new
21                                      InputStreamReader(System.in));
22      String adresaIP = null;
23      int numarPort = 0;
24      Socket conexiuneTCP;
25      try {
26        System.out.print("Introduceti adresa IP dorita: ");
27        adresaIP = inConsola.readLine();
28
29        System.out.print("Introduceti numarul de port dorit: ");
30        numarPort = Integer.parseInt(inConsola.readLine());
31
32        conexiuneTCP = new Socket(adresaIP, numarPort);
33
34        afisareInfoSocketTCP(conexiuneTCP);
35
36        InetAddress iaServer = conexiuneTCP.getInetAddress();
37        InfoInetAddress.afisareInetAddress(iaServer, "adresa serverului");
38
39        InetAddress iaLocala = conexiuneTCP.getLocalAddress();
40        InfoInetAddress.afisareInetAddress(iaLocala, "adresa locala");
41
42        conexiuneTCP.close();
43      }
44      catch (NumberFormatException ex) {
45        System.err.println("Numarul de port nu are format intreg");
46      }
47      catch (UnknownHostException ex) {
48        System.err.println("Nu poate fi gasita " + adresaIP);
49      }
50      catch (SocketException ex) {
51        System.err.println("Nu se poate face conexiune cu " + adresaIP +
52                                                          ":" + numarPort);
53      }
54      catch (IOException ex) {
55        System.err.println(ex);
56      }
57    }
58  }
```

3.3.3. Clasa socket TCP pentru server (ServerSocket) – interfata publica

1. Principalii constructori ai clasei **`ServerSocket:`**

`ServerSocket``()` Creaza un socket pentru server (de tip acceptor) nelegat la vreun port.
`ServerSocket``(int port)` Creaza un *socket* pentru server (de tip acceptor) legat la portul local de numar specificat. Valoarea 0 va conduce la crearea unui *socket* legat la un port liber nespecificat explicit. Numarul de indicatii privind cererile de conexiune care pot sta in asteptare la un moment dat este implicit 50.
`ServerSocket``(int port, int backlog)` Creaza un *socket* pentru server (de tip acceptor) legat la portul local de numar specificat. Valoarea 0 va conduce la crearea unui *socket* legat la un port liber nespecificat explicit. Numarul de indicatii privind cererile de conexiune care pot sta in asteptare la un moment dat este specificat de `backlog`.
`ServerSocket``(int port, int backlog, InetAddress bindAddr)` Creaza un *socket* pentru server (de tip acceptor) legat la portul local de numar specificat, la adresa locala specificata. Valoarea 0 va conduce la crearea unui *socket* legat la un port liber nespecificat explicit. Numarul de indicatii privind cererile de conexiune care pot sta in asteptare la un moment dat este specificat de `backlog`.

2. Declaratiile si descrierea catorva metode ale clasei **`ServerSocket`**:

`Socket`	**`accept`**`()` Asteapta cereri de conexiune facute catre *socket-ul* curent si le accepta. Metoda blocheaza executia pana cand e primita o cerere de conexiune. Metoda returneaza un obiect `Socket` prin care se poate desfasura comunicatia utilizand fluxuri de octeti.
`void`	**`close`**`()` Inchide *socket-ul* curent.
`InetAddress`	**`getInetAddress`**`()` Returneaza adresa IP locala a *socket-ului* curent.
`int`	**`getLocalPort`**`()` Returneaza numarul de port local pe care asculta *socket-ul* curent.
`int`	**`getReceiveBufferSize`**`()` Returneaza valoarea optiunii SO_RCVBUF pentru *`ServerSocket`-ul* curent, adica dimensiunea *buffer-ului* propus a fi utilizat de pentru fluxurile de intrare dinspre *socket-urile* obtinute prin acceptarea conexiunilor prin *socket-ul* curent.
`int`	**`getSoTimeout`**`()` Returneaza valoarea optiunii SO_TIMEOUT pentru *socket-ul* curent, adica durata in milisecunde cat apelul **`accept()`** asupra *socket-ului* curent blocheaza aplicatia, asteptand sosirea unor cereri de conexiune. Valoarea 0 semnifica asteptare (blocare) la infinit. Valoarea nu poate fi negativa. La expirarea unei valori nenule a *time-out-ului*, este generata o exceptie **`java.net.SocketTimeoutException`**, dar *socket-ul* curent continua sa functioneze.
`boolean`	**`isClosed`**`()` Returneaza true daca *socket-ul* curent este inchis, altfel returneaza false.
`void`	**`setReceiveBufferSize`**`(int size)` Stabileste valoarea optiunii SO_RCVBUF pentru fluxurile de intrare dinspre *socket-urile* obtinute prin acceptarea conexiunilor prin *socket-ul* curent.

`void`	**`setSoTimeout`**`(int timeout)` Stabileste valoarea optiunii SO_TIMEOUT pentru *socket-ul* curent, adica durata in milisecunde cat apelul **`accept()`** blocheaza aplicatia, asteptand sosirea unor cereri de conexiune. Valoarea 0 semnifica asteptare (blocare) la infinit. Valoarea nu poate fi negativa. La expirarea unei valori nenule a *time-out-ului*, este generata o exceptie **`java.net.SocketTimeoutException`**, dar *socket-ul* curent continua sa functioneze.
`String`	**`toString`**`()` Returneaza un `String` continand informatii (adresa IP si numarul de port) privind *socket-ul* curent.

In continuare este prezentata **secventa tipica pentru crearea *socket-ului* server al unei aplicatii acceptor (server)**:

```
// Stabilirea portului serverului
int portServer = 2000;

// Crearea socketului server (care accepta conexiunile)
ServerSocket serverTCP = new ServerSocket(portServer);

System.out.println("Server in asteptare pe portul "+portServer+"...");
```

ca si pentru **crearea *socket-ului* pentru tratarea conexiunii TCP cu un client**:

```
// Blocare in asteptarea cererii de conexiune - in momentul acceptarii
// cererii se creaza socketul care serveste conexiunea
Socket conexiuneTCP = serverTCP.accept();

System.out.println("Conexiune TCP pe portul " + portServer + "...");
```

Un caz special este acela in care se creaza un ***socket* server** pentru o aplicatie acceptor **fara a se preciza portul pe care asculta serverul pentru a primi cereri de conexiune si a le accepta**. In acest caz, un numar de port aleator este alocat, dintre cele neocupate in acel moment. Acest lucru se realizeaza **prin pasarea** valorii **`0`** constructorului **`ServerSocket()`**.

Programul urmator ilustreaza **crearea unui *socket* server cu numar de port alocat aleator**.

```
 1  import java.net.*;
 2  import java.io.*;
 3  public class InfoPortTCPAleator {
 4    public static void main (String args[]) {
 5      try {
 6        ServerSocket serverTCP = new ServerSocket(0);
 7        System.out.println("\nAcest server ruleaza pe portul " +
 8                           serverTCP.getLocalPort());
 9        serverTCP.close();
10      }
11      catch (IOException ex) {
12        System.err.println(ex);
13      }
14    }
15  }
```

Urmatorul program ilustreaza crearea de ***socket-uri* server** cu scopul **scanarii conexiunilor TCP de pe masina locala**. Programul **identifica porturile locale pe care exista conexiuni** (**pe care nu pot fi create servere**).

```
 1  import java.net.*;
 2  import java.io.*;
 3  public class ScannerPorturiTCPLocale {
 4    public static void main (String args[]) {
 5      ServerSocket serverTCP;
 6
 7      for (int portTCP=1; portTCP < 65536; portTCP++) {
 8        try {
 9
10          // Urmatoarele linii vor genera exceptie prinsa de blocul catch
11          // in cazul in care e deja un server pe portul portTCP
12          serverTCP = new ServerSocket(portTCP);
13          serverTCP.close();
14
15        }
16        catch (IOException ex) {
17          System.err.println("Exista un server TCP pe portul " + portTCP);
18        }
19      }
20    }
21  }
```

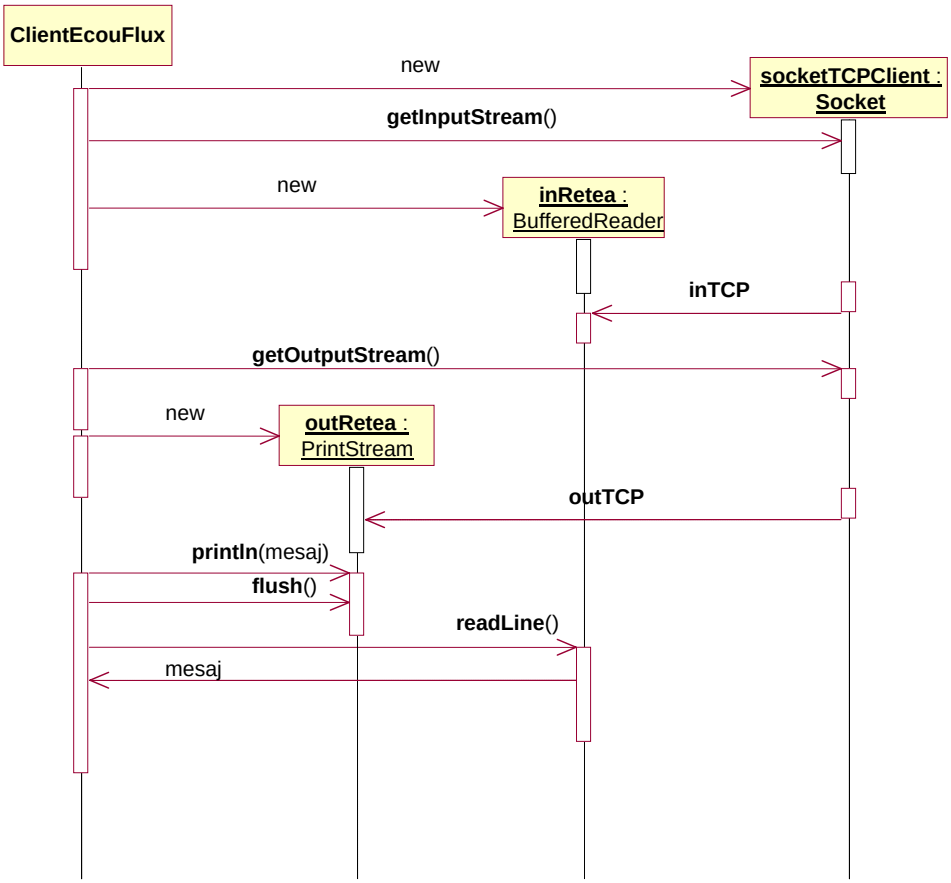
3.3.4. Clienti pentru servere flux (TCP)

Urmatorul program este un **client pentru un server ecou TCP care permite trimiterea unui singur mesaj.**

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  public class ClientEcouFlux {
 5    public static void main(String args[]) throws IOException {
 6      // Stabilirea adresei serverului
 7      String adresaServer = "localhost";
 8
 9      // Stabilirea portului serverului
10      int portServer = 2000;
11
12      // Obtinerea fluxului de intrare caractere dinspre consola
13      InputStreamReader inConsolaCaractere =
14                      new InputStreamReader(System.in);
15
16      // Obtinerea fluxului de intrare caractere dinspre consola,
17      // cu facilitati de stocare temporara
18      BufferedReader inConsola = new BufferedReader(inConsolaCaractere);
19
20      // Obtinerea fluxului de iesire caractere spre consola standard
21      PrintStream outConsola  = System.out;
22
23      // Crearea socketului (conectarea la server)
24      Socket socketTCPClient = new Socket(adresaServer, portServer);
25
26      // Obtinerea fluxului de intrare octeti TCP
27      InputStream inTCP = socketTCPClient.getInputStream();
28
29      // Obtinerea fluxului de intrare caractere dinspre retea
30      InputStreamReader inTCPCaractere = new InputStreamReader(inTCP);
31
32      // Obtinerea fluxului de intrare caractere dinspre retea,
33      // cu facilitati de stocare temporara
34      BufferedReader inRetea = new BufferedReader(inTCPCaractere);
```

```
35
36      // Obtinerea fluxului de iesire octeti TCP
37      OutputStream outTCP = socketTCPClient.getOutputStream();
38
39      // Obtinerea fluxului de iesire caractere, spre retea,
40      // similar consolei standard de iesire
41      PrintStream outRetea  = new PrintStream(outTCP);
42
43
44      // Citirea de la consola a mesajului de trimis
45      outConsola.print("Se trimite: ");
46      String mesajDeTrimis = inConsola.readLine();
47
48      // Scrierea catre retea (trimiterea mesajului)
49      outRetea.println(mesajDeTrimis);
50      // Fortarea trimiterii
51      outRetea.flush();
52
53
54      // Citirea dinspre retea (receptia mesajului)
55      String mesajPrimit = inRetea.readLine();
56
57      // Scrierea la consola (afisarea mesajului)
58      System.out.println("S-a primit: " + mesajPrimit);
59
60      // Inchiderea socketului (implicit a fluxurilor TCP)
61      socketTCPClient.close();
62      System.out.println("Bye!");
63    }
64  }
```

Schimbul de mesaje (creare *socket* si fluxuri, scriere in flux si citire din flux) este ilustrat in urmatoarea diagrama.



Urmatorul program este un **client pentru un server ecou TCP care permite trimiterea mai multor mesaje.** Mesajul format dintr-un punct (".") semnaleaza serverului terminarea mesajelor de trimis, clientul urmand sa isi termine executia.

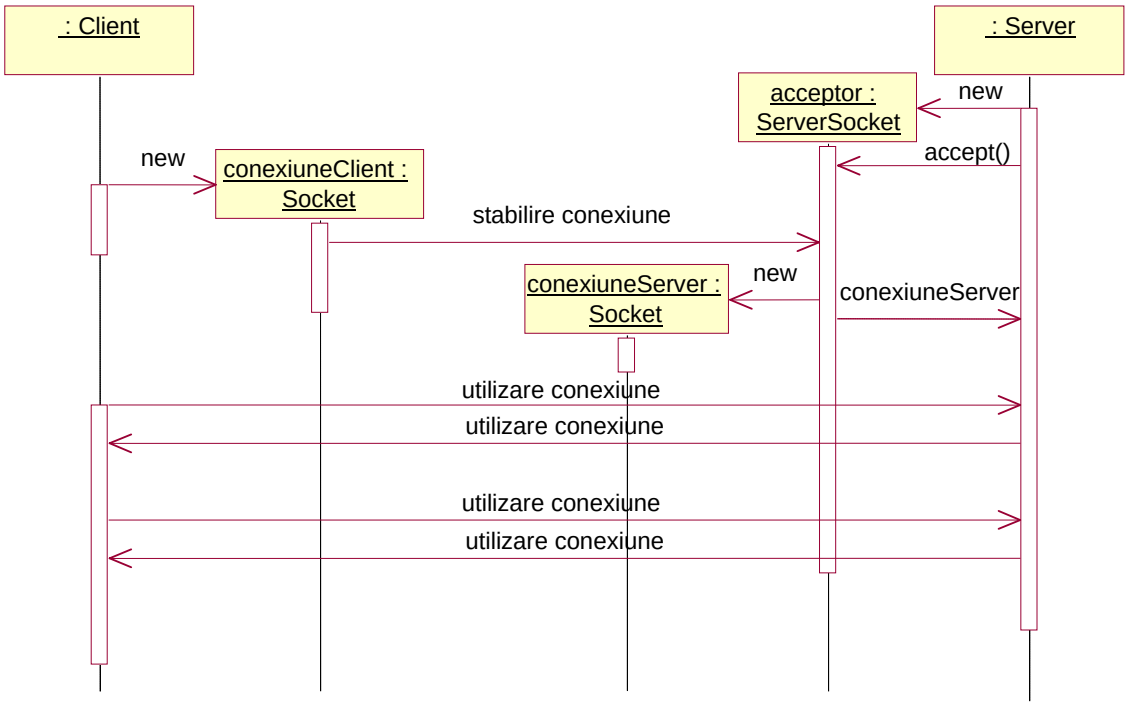
```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5    * Client pentru server ecou flux
 6    */
 7  public class ClientEcouFluxRepetitiv {
 8
 9    public static void main (String args[]) throws IOException {
10      BufferedReader inConsola  = new BufferedReader(new
11                                      InputStreamReader(System.in));
12      System.out.print("Introduceti adresa IP a serverului: ");
13      String adresaServer = inConsola.readLine();
14
15      System.out.print("Introduceti numarul de port al serverului: ");
16      int portServer = Integer.parseInt(inConsola.readLine());
17
18      // Crearea socketului (implicit este realizata conexiunea)
19      Socket conexiuneTCP = new Socket(adresaServer, portServer);
20      System.out.println("Conexiune TCP cu serverul " + adresaServer +
21                         ":" + portServer + "...");
22      System.out.println("Pentru oprire introduceti '.' si <Enter>");
23
24      // Crearea fluxurilor de caractere conectate la fluxurile de octeti
25      // obtinute de la socketul TCP
26      PrintStream outRetea = new
27                   PrintStream(conexiuneTCP.getOutputStream());
28      BufferedReader inRetea = new BufferedReader(
29                   new InputStreamReader(conexiuneTCP.getInputStream()));
30
31      while (true) {
32        // Citirea unei linii de la consola de intrare
33        System.out.print("Se trimite: ");
34        String mesajTrimis = inConsola.readLine();
35
36        // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
37        outRetea.println(mesajTrimis);
38        outRetea.flush();
39
40        // Citirea unei linii din fluxul de intrare TCP
41        String mesajPrimit = inRetea.readLine();
42
43        // Afisarea liniei citite la consola de iesire
44        System.out.println("S-a primit: " + mesajPrimit);
45
46        // Testarea conditiei de oprire a clientului
47        if (mesajPrimit.equals(".")) break;
48      }
49
50      // Inchiderea socketului (si implicit a fluxurilor)
51      conexiuneTCP.close();
52      System.out.println("Bye!");
53    }
54  }
```

3.3.5. Servere flux (TCP) non-concurente

In cazul serverelor care pot trata un singur client, codul necesar pentru stabilirea conexiunii si **pentru comunicatia cu clientul** este urmatorul:

```
// Initializare portServer
// Crearea socketului server (care accepta conexiunile)

ServerSocket serverTCP = new ServerSocket(portServer);

// Blocare in asteptarea cererii de conexiune

Socket conexiuneTCP = serverTCP.accept();

// Crearea fluxurilor conectate la fluxurile obtinute de la socketul TCP
PrintStream out = new PrintStream(conexiuneTCP.getOutputStream());
BufferedReader in = new BufferedReader(
               new InputStreamReader(conexiuneTCP.getInputStream()));

// Tratarea clientului
while (true) {
  // Citiri din fluxul de intrare TCP cu in.readLine();

  // Scrieri in fluxul de iesire TCP cu out.println(); out.flush();
} // Incheierea tratarii clientului

// Inchiderea socketului
conexiuneTCP.close();
```

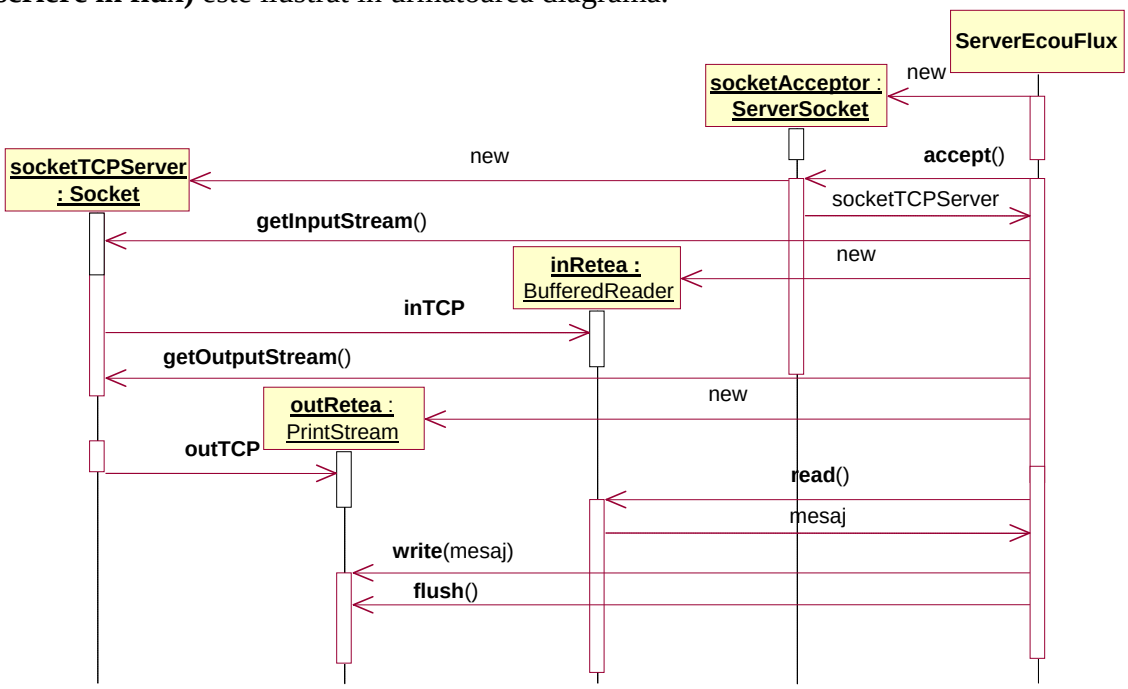
Secventa de mesaje schimbate la client, la server si intre client si server este, in acest caz, urmatoarea:



Urmatorul program este un **server ecou TCP care permite clientului trimiterea unui singur mesaj.**

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4    * Server ecou flux pentru servirea unui client o singura data
 5    */
 6  public class ServerEcouFlux {
 7
 8    public static void main (String args[]) throws IOException {
 9      BufferedReader inConsola  = new BufferedReader(new
10                                      InputStreamReader(System.in));
11
12      System.out.print("Introduceti numarul de port al serverului: ");
13      int portServer = Integer.parseInt(inConsola.readLine());
14
15      // Crearea socketului server (care accepta conexiunile)
16      ServerSocket serverTCP = new ServerSocket(portServer);
17      System.out.println("Server in asteptare pe portul "+portServer+"...");
18
19      // Blocare in asteptarea cererii de conexiune - in momentul acceptarii
20      // cererii se creaza socketul care serveste conexiunea
21      Socket conexiuneTCP = serverTCP.accept();
22      System.out.println("Conexiune TCP pe portul " + portServer + "...");
23
24      // Crearea fluxurilor de caractere conectate la fluxurile de octeti
25      // obtinute de la socketul TCP
26      PrintStream outRetea = new PrintStream(conexiuneTCP.getOutputStream());
27      BufferedReader inRetea = new BufferedReader(
28                      new InputStreamReader(conexiuneTCP.getInputStream()));
29
30      // Citirea unei linii din fluxul de intrare TCP
31      String mesajPrimit = inRetea.readLine();
32
33      // Afisarea liniei citite la consola de iesire
34      System.out.println("Mesaj primit: " + mesajPrimit);
35
36      // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
37      outRetea.println(mesajPrimit);
38      outRetea.flush();
39
40      // Inchiderea socketului (si implicit a fluxurilor)
41      conexiuneTCP.close();
42    }
43  }
```

Schimbul de mesaje la server (creare *socket* server, *socket* tratare client si fluxuri, citire din flux si scriere in flux) este ilustrat in urmatoarea diagrama.

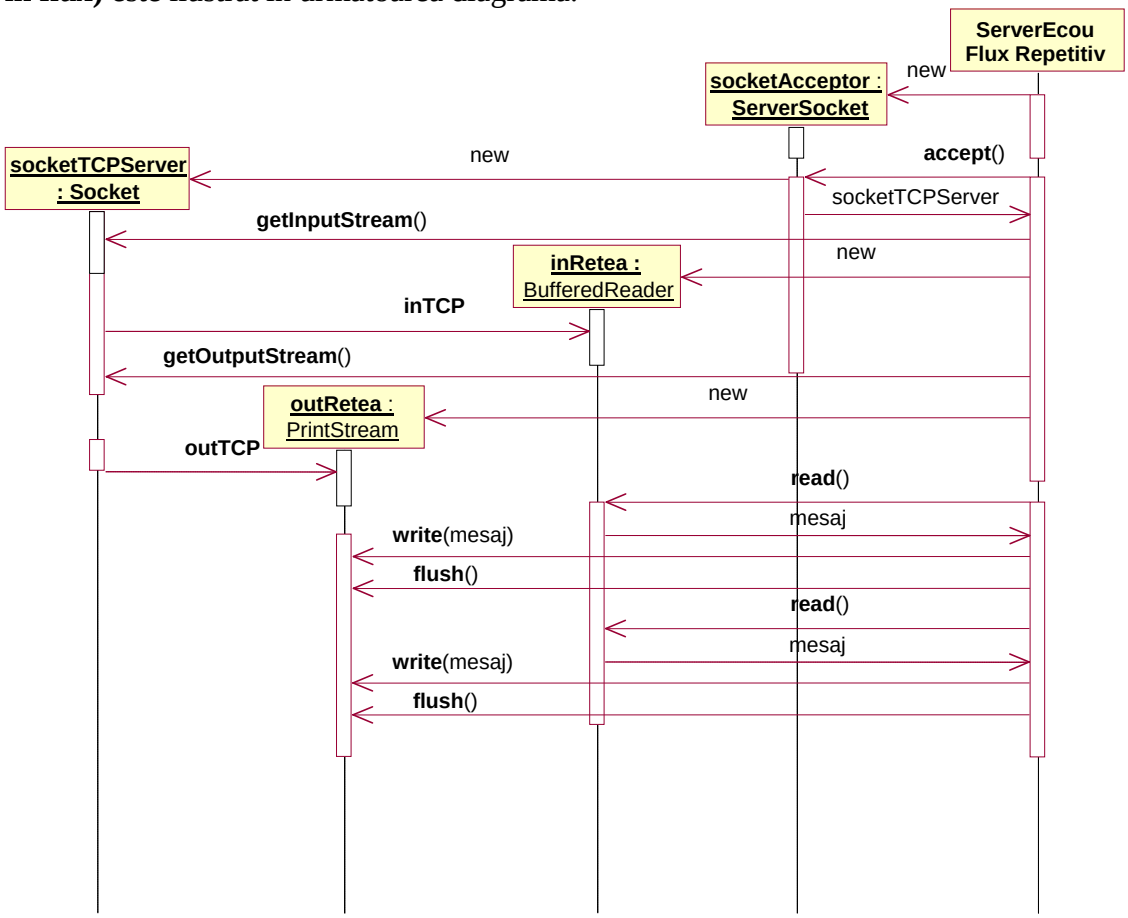


Urmatorul program este un **server ecou TCP care permite servirea unui singur client in mod repetat.**

Mesajul format dintr-un punct (".") semnaleaza serverului terminarea mesajelor de trimis, clientul urmand sa isi termine executia.

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5    * Server ecou flux pentru servirea unui client in mod repetat
 6    */
 7  public class ServerEcouFluxRepetiv {
 8
 9    public static void main (String args[]) throws IOException {
10      BufferedReader inConsola  = new BufferedReader(new
11                                      InputStreamReader(System.in));
12
13      System.out.print("Introduceti numarul de port al serverului: ");
14      int portServer = Integer.parseInt(inConsola.readLine());
15
16      // Crearea socketului server (care accepta conexiunile)
17      ServerSocket serverTCP = new ServerSocket(portServer);
18      System.out.println("Server in asteptare pe portul "+portServer+"...");
19
20      // Blocare in asteptarea cererii de conexiune - in momentul acceptarii
21      // cererii se creaza socketul care serveste conexiunea
22      Socket conexiuneTCP = serverTCP.accept();
23      System.out.println("Conexiune TCP pe portul " + portServer + "...");
24
25      // Crearea fluxurilor de caractere conectate la fluxurile de octeti
26      // obtinute de la socketul TCP
27      PrintStream outRetea = new PrintStream(conexiuneTCP.getOutputStream());
28      BufferedReader inRetea = new BufferedReader(
29                      new InputStreamReader(conexiuneTCP.getInputStream()));
30
31      while (true) {
32        // Citirea unei linii din fluxul de intrare TCP
33        String mesajPrimit = inRetea.readLine();
34
35        // Afisarea liniei citite la consola de iesire
36        System.out.println("Mesaj primit: " + mesajPrimit);
37
38        // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
39        outRetea.println(mesajPrimit);
40        outRetea.flush();
41
42        // Testarea conditiei de oprire a servirii
43        if (mesajPrimit.equals(".")) break;
44      }
45
46      // Inchiderea socketului (si implicit a fluxurilor)
47      conexiuneTCP.close();
48      System.out.println("Bye!");
49    }
50  }
```

Schimbul de mesaje (creare *socket* server, *socket* tratare client si fluxuri, citire din flux si scriere in flux) este ilustrat in urmatoarea diagrama.



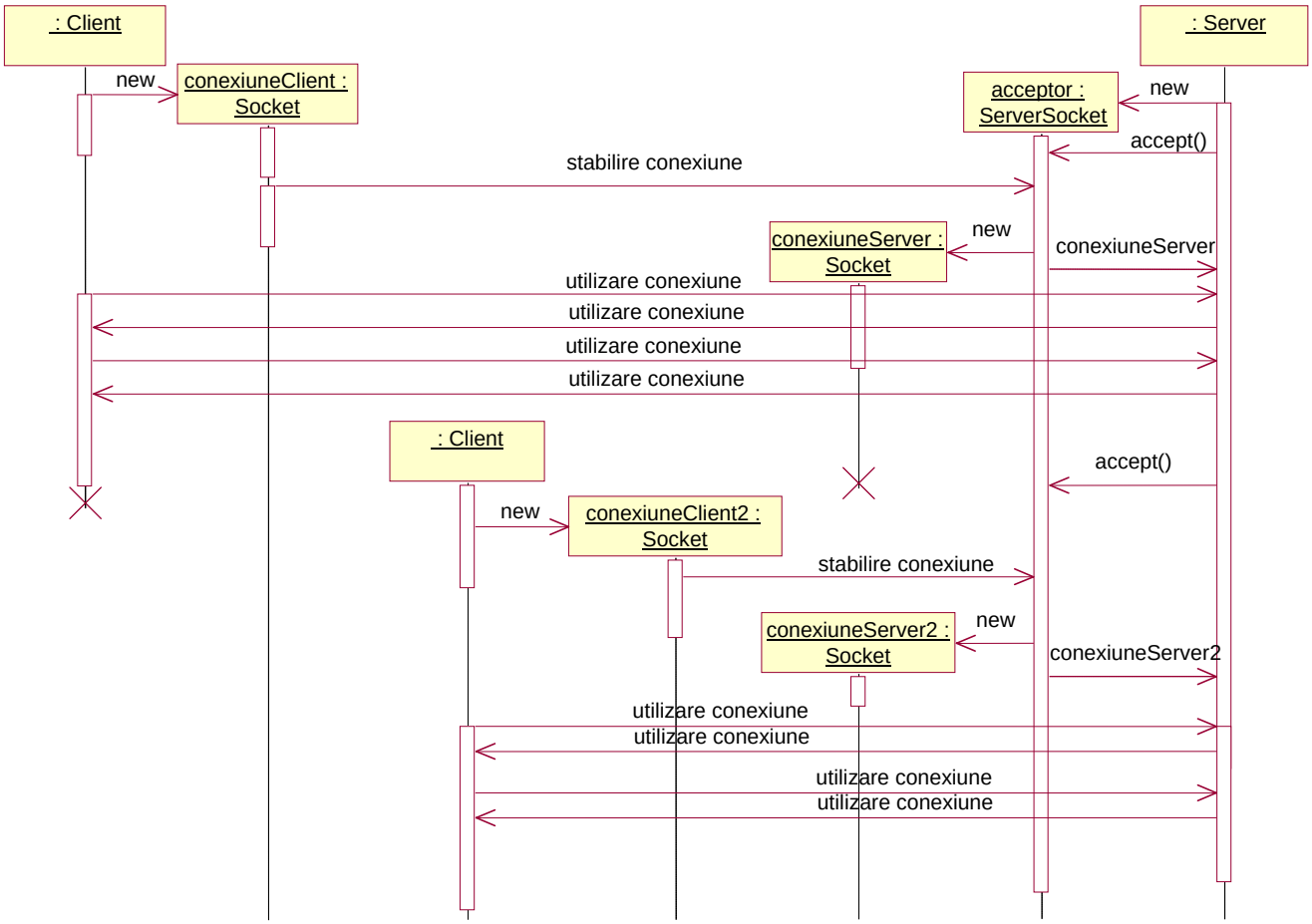
In cazul serverelor care pot trata mai multi clienti in mod secvential (succesiv), codul necesar pentru stabilirea conexiunilor si **pentru comunicatia cu clientii** este urmatorul:

```
// Initializare portServer
// Crearea socketului server (care accepta conexiunile)
ServerSocket serverTCP = new ServerSocket(portServer);

// Servirea mai multor clienti succesivi (in mod secvential)
while (true) {

  // Blocare in asteptarea cererii de conexiune

  Socket conexiuneTCP = serverTCP.accept();

  // Crearea fluxurilor conectate la fluxurile obtinute de la socketul TCP
  PrintStream out = new PrintStream(conexiuneTCP.getOutputStream());
  BufferedReader in = new BufferedReader(
              new InputStreamReader(conexiuneTCP.getInputStream()));

  // Tratarea clientului curent
  while (true) {
    // Citiri din fluxul de intrare TCP cu in.readLine();

    // Scrieri in fluxul de iesire TCP cu out.println(); out.flush();
  } // Incheierea tratarii clientului

  // Inchiderea socketului
  conexiuneTCP.close();

} // Gata pentru a servi urmatorul client
```

Server care permite receptia si trimiterea in ecou a mai multor mesaje sosite de la mai multi clienti, succesiv:

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4    * Server ecou flux pentru servirea mai multor clienti succesivi
 5    */
 6  public class ServerEcouFluxRepetivSecvential {
 7
 8    public static void main (String args[]) throws IOException {
 9      BufferedReader inConsola  = new BufferedReader(new
10                                      InputStreamReader(System.in));
11      System.out.print("Introduceti numarul de port al serverului: ");
12      int portServer = Integer.parseInt(inConsola.readLine());
13
14      // Crearea socketului server (care accepta conexiunile)
15      ServerSocket serverTCP = new ServerSocket(portServer);
16
17      // Servirea mai multor clienti succesivi (in mod secvential)
18      while (true) {
19        System.out.println("Server in asteptare pe port "+portServer+"...");
20         // Blocare in asteptarea cererii de conexiune - in momentul
21        // acceptarii cererii se creaza socketul care serveste conexiunea
22        Socket conexiuneTCP = serverTCP.accept();
23        System.out.println("Conexiune TCP pe portul " + portServer + "...");
24
25        // Crearea fluxurilor de caractere conectate la fluxurile de octeti
26        // obtinute de la socketul TCP
27        PrintStream outRetea = new
28               PrintStream(conexiuneTCP.getOutputStream());
29        BufferedReader inRetea = new BufferedReader(
30               new InputStreamReader(conexiuneTCP.getInputStream()));
31
32        // Servirea clientului curent
33        while (true) {
34          // Citirea unei linii din fluxul de intrare TCP
35          String mesajPrimit = inRetea.readLine();
36
37          // Afisarea liniei citite la consola de iesire
38          System.out.println("Mesaj primit: " + mesajPrimit);
39
40          // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
41          outRetea.println(mesajPrimit);
42          outRetea.flush();
43
44          // Testarea conditiei de oprire a servirii
45          if (mesajPrimit.equals(".")) break;
46        }
47        // Inchiderea socketului (si implicit a fluxurilor)
48        conexiuneTCP.close();
49        System.out.println("Bye!");
50      }
51    }
52  }
```

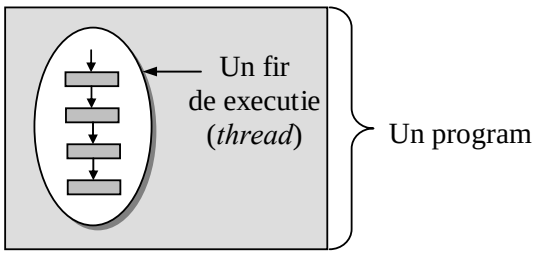
Secventa de mesaje schimbate la client, la server si intre client si server este urmatoarea:



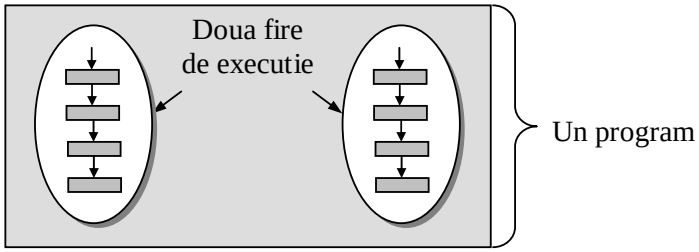
3.3.6. Fire de executie (threads)

Programele de calcul simple sunt **secventiale**, fiecare avand **un inceput, o secventa de executii si un sfarsit**. In orice moment pe durata executiei unui astfel de program **exista un singur punct de executie**.

Un fir de executie (*thread*), sau mai simplu, **un fir**, este similar acestor programe secventiale, in sensul ca are **un inceput, o secventa de executii si un sfarsit**, si in orice moment pe durata executiei firului **exista un singur punct de executie**. Totusi, **un fir nu este el insusi un program**, deoarece **nu poate fi executat de sine statator**. In schimb, **firul este executat** (ruleaza) **intr-un program**. Relatia dintre un program si un fir de executie este ilustrata mai jos.



Un fir de executie **(*thread*)** poate fi definit ca **un flux de control secvential in cadrul unui program**. **Posibilitatea utilizarii mai multor fire de executie intr-un singur program**, ruland (fiind executate) in acelasi timp si realizand diferite sarcini (nu in mod necesar diferite), este numita ***multithreading***. Un navigator (*browser*) Web este un exemplu de **aplicatie multi-filara (*multithreaded*)**. In *browser* **se poate parcurge pagina in timpul descarcarii unei miniaplicatii Java (*applet*)** a unei imagini, etc.



O denumire alternativa a firelor de executie este de ***lightweight process***, deoarece un fir este similar unui proces real in sensul ca ambele sunt fluxuri de control secventiale. Totusi, **un fir** este considerat *lightweight* deoarece el **ruleaza in contextul unui program si are la dispozitie reursele alocate pentru acel program si mediul programului**.

Ca flux de control secvential, **un fir de executie utilizeaza o parte din resursele programului in care ruleaza**. De exemplu, el trebuie sa aiba propria stiva de executie si propriul contor de program. Codul firului de executie lucreaza doar in acel context. De aceea, **o denumire alternativa** este aceea de **context de executie**.

Masina virtuala Java, JVM (*Java Virtual Machine*), **permite aplicatiilor sa aiba mai multe fire de executie care ruleaza in mod concurent** (in paralel).

Fiecare fir de executie are un anumit **nivel de prioritate**. Firele cu nivel de prioritate mai mare sunt executate inaintea celor cu nivel de prioritate mai mic. Atunci cand codul care ruleaza intr-un fir de executie creaza un nou obiect de tip `Thread`, noul fir de executie are initial acelasi nivel de prioritate cu firul de executie care l-a creat.

Fiecare fir de executie poate fi marcat ca *daemon*. Un fir de executie creat de un fir de executie *daemon* este la randul lui un *daemon*.

La lansarea masinii virtuale Java, exista in mod normal un singur fir de executie *non-daemon* (cel ce apeleaza metoda `main()` a clasei cu care incepe executia). **Masina virtuala Java continua sa execute firele pana cand**:
- **este apelata metoda `exit()`** a clasei `Runtime` si managerul de securitate permite executia operatiei exit, sau
- **toate firele care nu sunt *daemoni* au murit** (si-au incheiat executia), fie prin returnarea din apelul metodei `run()`, fie prin aruncarea unei exceptii care se propaga dincolo de metoda `run()`.

Pentru a crea un nou fir de executie exista doua modalitati.

1. Se poate declara o clasa ca subclasa a clasei **`Thread`**, subclasa care **trebuie sa rescrie codul (*override*) metodei `run()` a clasei `Thread`** (care nu contine nici un cod), noul fir de executie fiind creat prin alocarea si lansarea unei instante a subclasei.

Formatul pentru crearea unei subclase care extinde clasa `Thread` si ii reimplementeaza metoda `run()` este urmatorul:

```
class FirT extends Thread {

  public void run() {
    // codul firului de executie
  }
}
```

Formatul pentru crearea unei instante a subclasei este urmatorul:

```
FirT fir = new FirT();
```

2. Ca alternativa, se poate declara o clasa care implementeaza interfata **`Runnable`**, interfata care contine doar declaratia unei metode **`run()`** (declaratie identical celei din clasa `Thread`, deoarece clasa `Thread` implementeaza interfata `Runnable`), noul fir de executie fiind creat prin alocarea unei instante a noii clase, pasarea acestei instante ca parametru al constructorului, la crearea unei instante a clasei `Thread`, si lansarea acelei instante a clasei `Thread`.

Formatul pentru crearea unei clase care implementeaza interfata `Runnable` si ii implementeaza metoda `run()` este urmatorul:

```
class FirR implements Runnable {

  public void run() {
    // codul firului de executie
  }
}
```

Formatul pentru crearea unei instante a noii clase si apoi a unei instante a clasei `Thread` este urmatorul:

```
FirR r = new FirR();
Thread fir = new Thread(r);
```

In ambele cazuri **formatul pentru lansarea noului fir de executie,** este urmatorul:

```
fir.start();
```

Desigur, exista si **variante compacte pentru crearea si lansarea noilor fire de executie**, cum ar fi:

```
new FirT().start();                 // nu exista variabila de tip FirT
                                    // care sa refere explicit firul
```

sau

```
FirR r = new FirR();
new Thread(r).start();              // nu exista variabila de tip Thread
                                    // care sa refere explicit firul
```

sau

```
Thread fir = new Thread(new FirR());
fir.start();                        // nu exista variabila de tip FirR
```

sau

```
new Thread(new FirR()).start();     // nu exista variabila de tip Thread
                                    // care sa refere explicit firul
                                    // si nici variabila de tip FirR
```

Fiecare fir de executie are un nume, cu scopul identificarii lui. Mai multe fire de executie pot avea acelasi nume. **Daca nu este specificat** un nume in momentul constructiei (initializarii) firului, **un nou nume (aleator stabilit) este generat** pentru acesta.

3.3.7. Clasa Thread – interfata publica

1. Principalii constructori ai clasei **`Thread`:**

`Thread``()` Initializeaza un nou obiect de tip `Thread` (care trebuie sa implementeze metoda `run()`)
`Thread``(Runnable target)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta `target` (care trebuie sa implementeze metoda `run()`) dintr-o clasa care implementeaza interfata `Runnable`.
`Thread``(Runnable target, String name)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta `target` dintr-o clasa care implementeaza interfata `Runnable`, si un nume `name`.

`Thread``(String name)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un nume `name`.
`Thread``(ThreadGroup group, Runnable target)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta `target` dintr-o clasa care implementeaza interfata `Runnable`, grupul de fire de care apartine `group`.
`Thread``(ThreadGroup group, Runnable target, String name)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta `target` dintr-o clasa care implementeaza interfata `Runnable`, grupul de fire de care apartine `group`, si un nume.
`Thread``(ThreadGroup group, Runnable target, String name, long stackSize)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica un obiect tinta dintr-o clasa care implementeaza interfata `Runnable`, grupul de fire de care apartine, un nume, si dimensiunea stivei care ii va fi alocata pentru executie (*stack size*).
`Thread``(ThreadGroup group, String name)` Initializeaza un nou obiect de tip `Thread` caruia i se specifica grupul de fire de care apartine `group`, si un nume `name`.

2. Declaratiile si descrierea catorva metode ale clasei **`Thread`**:

`int`	**`activeCount`**`()` Returneaza numarul firelor de executie active din grupul curent de fire.
`void`	**`checkAccess`**`()` Determina daca firul curent executat are permisiunea de a modifica firul de executie curent (`this`).
`static Thread`	**`currentThread`**`()` Returneaza o referinta catre obiectul `Thread` curent executat.
`void`	**`destroy`**`()` Distruge firul de executie curent (`this`), fara eliberarea resurselor.
`static void`	**`dumpStack`**`()` Afiseaza informatiile din stiva ale firului de executie curent.
`static int`	**`enumerate`**`(Thread[] tarray)` Copiaza in tabloul specificat ca parametru referinte catre toate firele de executie active din grupul de fire curent si din subgrupurile sale.
`ClassLoader`	**`getContextClassLoader`**`()` Returneaza obiectul context al firului de executie curent.
`String`	**`getName`**`()` Returneaza numele firului de executie curent.
`int`	**`getPriority`**`()` Returneaza nivelul de prioritate al firului de executie curent.
`ThreadGroup`	**`getThreadGroup`**`()` Returneaza grupul de fire caruia ii apartine firul de executie curent.
`static boolean`	**`holdsLock`**`(Object obj)` Returneaza `true` daca firul de executie curent detine lock-ul monitorului creat pe obiectul specificat ca parametru.
`void`	**`interrupt`**`()` Intrerupe firul de executie curent.
`static boolean`	**`interrupted`**`()` Verifica daca firul de executie curent executat a fost intrerupt.
`boolean`	**`isAlive`**`()` Verifica daca firul de executie curent este in viata.

`boolean`	**`isDaemon`**`()` Verifica daca firul de executie curent este fir *daemon*.
`boolean`	**`isInterrupted`**`()` Verifica daca firul de executie curent a fost intrerupt.
`void`	**`join`**`()` Asteapta pana cand firul de executie curent moare.
`void`	**`join`**`(long millis)` Asteapta cel putin valoarea specificata ca parametru, in milisecunde, pana cand firul de executie curent moare.
`void`	**`join`**`(long millis, int nanos)` Asteapta cel putin `millis` milisecunde plus `nanos` nanosecunde pana cand firul de executie curent moare.
`void`	**`run`**`()` Implicit metoda nu executa nimic si returneaza.
`void`	**`setContextClassLoader`**`(ClassLoader cl)` Stabileste obiectul context al firului de executie curent.
`void`	**`setDaemon`**`(boolean on)` Marcheaza firul de executie curent ca fir *daemon*.
`void`	**`setName`**`(String name)` Schimba numele firului de executie curent cu cel specificat ca parametru.
`void`	**`setPriority`**`(int newPriority)` Stabileste nivelul de prioritate al firului de executie curent.
`static void`	**`sleep`**`(long millis)` Forteaza firul de executie curent executat sa cedeze temporar executia, pentru numarul de milisecunde specificat.
`static void`	**`sleep`**`(long millis, int nanos)` Forteaza firul de executie curent executat sa cedeze temporar executia, pentru numarul de milisecunde plus numarul de nanosecunde specificat.
`void`	**`start`**`()` Forteaza firul de executie curent sa isi inceapa executia. Masina virtuala Java apeleaza metoda `run()` a firului de executie curent.
`void`	**`stop`**`()` **Deprecated.** *Metoda este in mod inerent nesigura. In majoritatea cazurilor utilizarea `stop()` poate fi inlocuita cu un cod care modifica o anumita variabila care indica faptul ca firul de executie tinta trebuie sa isi incheie executia, firul de executie tinta urmand sa verifice variabila in mod regulat si sa returneze din metoda sa run() atunci cand variabila indica necesitatea de a se incheia executia. Daca firul de executie tinta asteapta pentru perioade lungi de timp, atunci trebuie utilizata metoda `interrupt()` pentru a intrerupe asteptarea.*
`void`	**`stop`**`(Throwable obj)` **Deprecated.** *Vezi explicatia de mai sus*
`String`	**`toString`**`()` Returneaza o reprezentare ca sir de caractere a firului de executie curent, incluzand numele, prioritatea, si grupul de fire ale firului de executie curent.
`static void`	**`yield`**`()` Forteaza firul de executie curent executat sa se opreasca temporar si sa permita altui fir de executie sa fie executat.

Urmatoarea clasa Java, **FirSimplu**, extinde clasa **Thread**, iar metoda ei principala lanseaza metoda **run**() ca nou fir de executie.

```
 1    public class FirSimplu extends Thread {
 2      public FirSimplu(String str) {
 3          super(str);
 4      }
 5      public void run() {
 6          for (int i = 0; i < 10; i++) {
 7              System.out.println(i + " " + getName());
 8              try {
 9                  sleep((long)(Math.random() * 1000));
10              } catch (InterruptedException e) {}
11          }
12          System.out.println("Gata! " + getName());
13      }
14
15      public static void main (String[] args) {
16          new FirSimplu("Unu").start();
17      }
18    }
```

Rezultatul pe ecran al executiei programului:

```
>java FirSimplu
0 Unu
1 Unu
2 Unu
3 Unu
4 Unu
Gata! Unu
```

Clasa **DemoDouaFire** lanseaza doua fire de executie **FirSimplu** care sunt executate concurent.

```
1  public class DemoDouaFire {
2
3      public static void main (String[] args) {
4          new FirSimplu("Unu").start();
5          new FirSimplu("Doi").start();
6      }
7  }
```

Rezultatele pe ecran a doua executii succesive:

```
>java DemoDouaFire
0 Unu
0 Doi
1 Unu
1 Doi
2 Unu
2 Doi
3 Doi
3 Unu
4 Doi
Gata! Doi
4 Unu
Gata! Unu
```

```
>java DemoDouaFire
0 Unu
0 Doi
1 Unu
1 Doi
2 Unu
2 Doi
3 Unu
4 Unu
Gata! Unu
3 Doi
4 Doi
Gata! Doi
```

Se observa ca firele pot avea evolutii diferite, in functie de durata intarzierii introdusa de linia de cod:

```
sleep((long)(Math.random() * 1000));
```

a programului **FirSimplu**.

3.3.8. Servere flux concurente

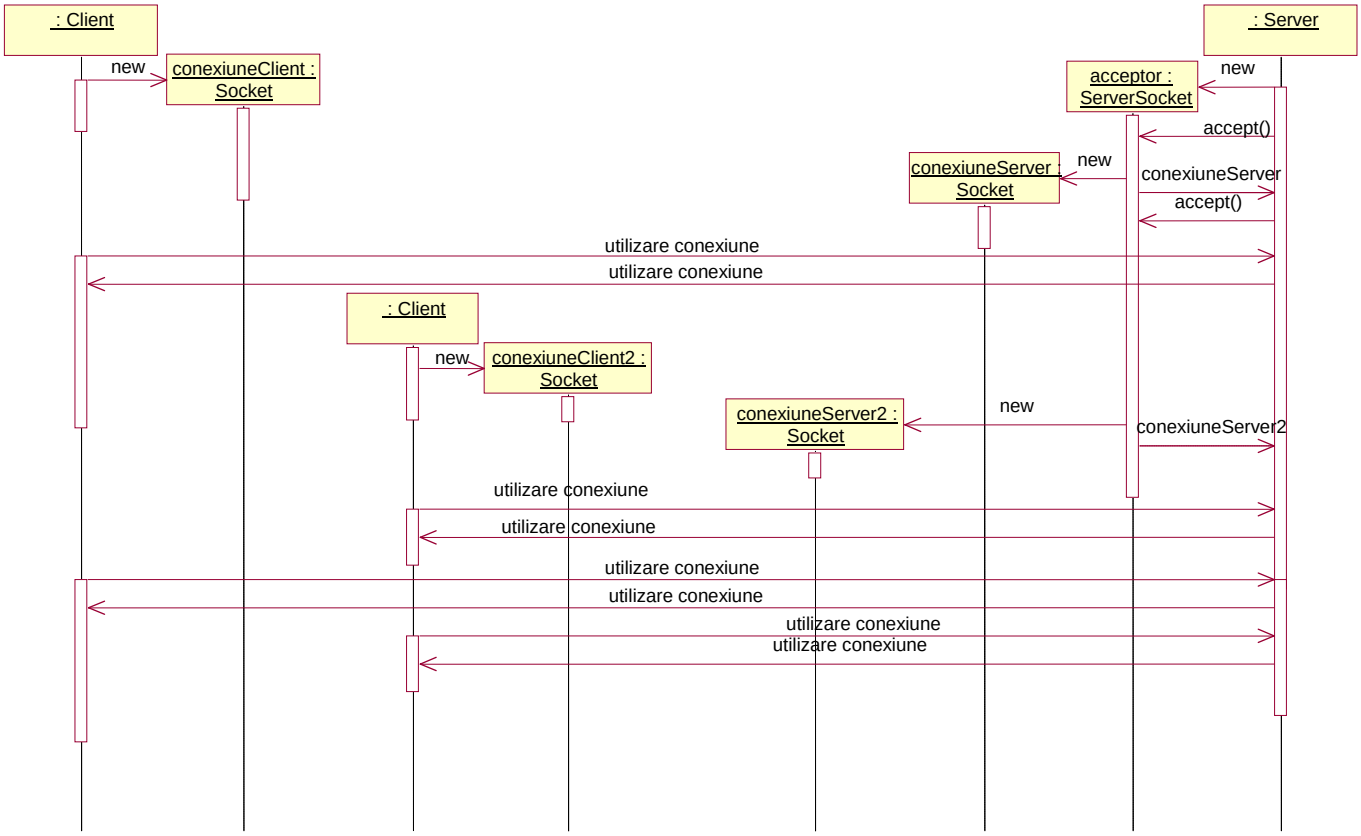
In cazul serverelor care pot trata mai multi clienti in mod concurent (in paralel), codul necesar pentru stabilirea conexiunilor cu clientii, ce urmeaza a fi tratati in clasa fir de executie pe care o vom denumi in mod generic **`ClasaFiruluiDeTratare`**, este urmatorul:

```
// initializare portServer

// Crearea socketului server (care accepta conexiunile)

ServerSocket serverTCP = new ServerSocket(portServer);

// Servirea mai multor clienti in paralel (in mod concurent)
while (true) {

  // Blocare in asteptarea cererii de conexiune

  Socket conexiuneTCP = serverTCP.accept();

  // Crearea si lansarea firului de executie pentru tratarea noului client

  ClasaFiruluiDeTratare firExecutie = new ClasaFiruluiDeTratare(conexiuneTCP);
  firExecutie.start();

} // Gata pentru a accepta urmatorul client
```

Codul necesar pentru comunicatia cu un client, din clasa fir de executie pentru tratarea clientului (denumita in mod generic **`ClasaFiruluiDeTratare`**), este urmatorul:

```
Socket socketTCP;                          // Atribut

public ClasaFiruluiDeTratare(Socket s) {   // Constructor
  socketTCP = s;                           // Initializarea atributului
}

public void run() {                       // Implementarea firului de executie

  // Crearea fluxurilor conectate la fluxurile obtinute de la socketul TCP

  PrintStream out = new PrintStream(conexiuneTCP.getOutputStream());
  BufferedReader in = new BufferedReader(
              new InputStreamReader(conexiuneTCP.getInputStream()));

  // Tratarea clientului curent
  while (true) {
    // Citiri din fluxul de intrare TCP cu in.readLine();

    // Scrieri in fluxul de iesire TCP cu out.println(); out.flush();
  } // Incheierea tratarii clientului

  // Inchiderea socketului
  conexiuneTCP.close();

}
```

Secventa de mesaje schimbate la client, la server si intre client si server este, in acest caz, urmatoarea:



Exemplu complet de server ecou care poate trata mesaje sosite de la mai multi clienti in paralel:

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5    * Server ecou flux pentru servirea mai multor clienti in acelasi timp
 6    */
 7  public class ServerEcouFluxRepetivConcurent extends Thread {
 8
 9    /**
10      * Socket de servire client
11      */
12    private Socket conexiuneTCP;
13
14    /**
15      * Initializare socket servire client
16      */
17    public ServerEcouFluxRepetivConcurent(Socket socketTCP) {
18
19      conexiuneTCP = socketTCP;
20    }
21
22    /**
23      * Punct de intrare program
24      */
25    public static void main (String args[]) throws IOException {
26
27      BufferedReader inConsola  = new BufferedReader(new
28                                      InputStreamReader(System.in));
29      System.out.print("Introduceti numarul de port al serverului: ");
```

```
30
31      int portServer = Integer.parseInt(inConsola.readLine());
32
33      // Crearea socketului server (care accepta conexiunile)
34      ServerSocket serverTCP = new ServerSocket(portServer);
35
36      System.out.println("Server in asteptare pe portul "+portServer+"...");
37
38      // Servirea mai multor clienti in acelasi timp (in mod concurent)
39      while (true) {
40
41        // Blocare in asteptarea cererii de conexiune - in momentul
42        // acceptarii cererii se creaza socketul care serveste conexiunea
43        Socket socketTCP = serverTCP.accept();
44        System.out.println("Conexiune TCP pe portul " + portServer + "...");
45
46        new ServerEcouFluxRepetivConcurent(socketTCP).start(); // run()
47
48      }
49    }
50
51    /**
52      * Fir de servire client
53      */
54    public void run() {
55
56      try {
57
58        // Crearea fluxurilor de caractere conectate la fluxurile de octeti
59        // obtinute de la socketul TCP
60        PrintStream outRetea = new
61              PrintStream(conexiuneTCP.getOutputStream());
62        BufferedReader inRetea = new BufferedReader(
63              new InputStreamReader(conexiuneTCP.getInputStream()));
64
65        BufferedReader inConsola  = new BufferedReader(new
66                                      InputStreamReader(System.in));
67
68        // Servirea clientului curent
69        while (true) {
70          // Citirea unei linii din fluxul de intrare TCP
71          String mesajPrimit = inRetea.readLine();
72
73          // Afisarea liniei citite la consola de iesire
74          System.out.println("Mesaj primit: " + mesajPrimit);
75
76          // Scrierea liniei in fluxul de iesire TCP, cu fortarea trimiterii
77          outRetea.println(mesajPrimit);
78          outRetea.flush();
79
80          // Testarea conditiei de oprire a servirii
81          if (mesajPrimit.equals(".")) break;
82        }
83
84        // Inchiderea socketului (si implicit a fluxurilor)
85        conexiuneTCP.close();
86        System.out.println("Bye!");
87      }
88      catch (IOException ex) {
89        System.err.println(ex);
90      }
91    }
92  }
```

3.4. Socketuri datagrama (UDP)

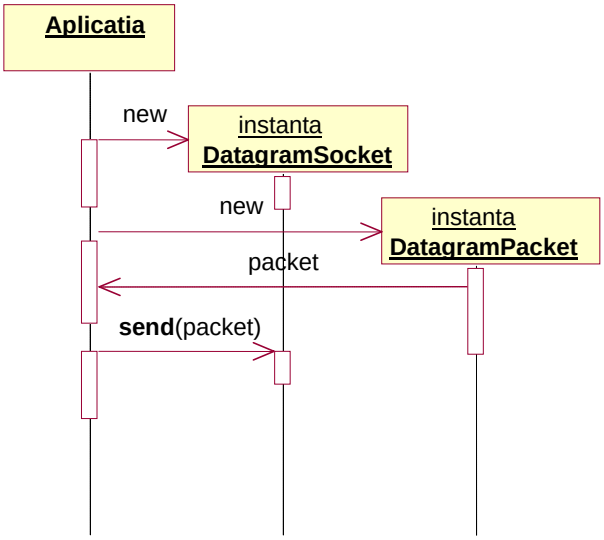
3.4.1. Lucrul cu socketuri datagrama (UDP)

Java ofera, in pachetul **`java.net`**, mai multe clase pentru lucrul cu *socket-uri* datagrame (UDP). Urmatoarele **clase Java** sunt **implicate in realizarea comunicatiilor UDP** obisnuite: **`DatagramSocket`** si **`DatagramPacket`**.

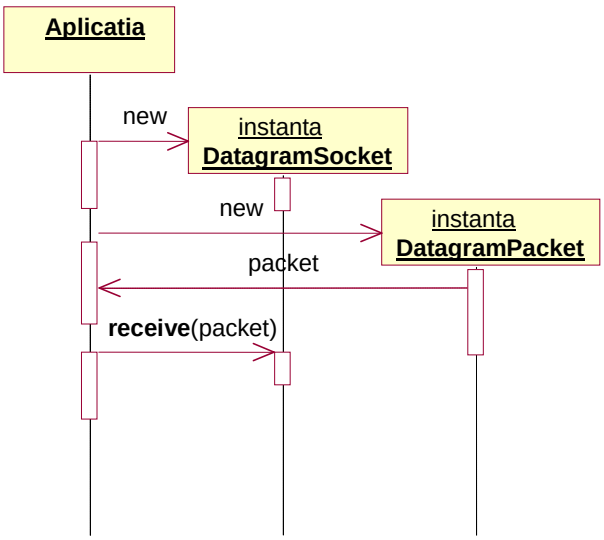
Clasa **`DatagramPacket`** reprezinta un **pachet UDP** (o **datagrama**). Pachetele datagrama sunt utilizate pentru livrare fara conexiune si includ in mod normal informatii privind adresele IP si porturile sursa si destinatie.

Clasa **`DatagramSocket`** reprezinta ***socket-ul* UDP**, **prin care se trimit sau se primesc pachete datagrama** peste retele IP **prin intermediul protocolului UDP**.

Un **`DatagramPacket` este trimis printr-un `DatagramSocket` apeland** la metoda **`send()`** a clasei **`DatagramSocket`**, **pasand** ca parametru respectivul **`DatagramPacket`.**



Un **`DatagramPacket` este primit printr-un `DatagramSocket` apeland** la metoda **`receive()`** a clasei **`DatagramSocket`**, **pasand** ca parametru un **`DatagramPacket`** pregatit pentru receptie.



Clasa **`MulticastSocket`** poate fi utilizata pentru trimiterea si receptia unui `DatagramPacket` catre, respectiv dinspre, un **grup multicast**. Clasa `MulticastSocket` este o subclasa a `DatagramSocket` care adauga functionalitati legate de multicast.

3.4.2. Clasa pachet (datagrama) UDP (DatagramPacket) – interfata publica

1. Principalii constructori ai clasei `DatagramPacket`:

`DatagramPacket``(byte[] buf, int length)` Construieste un obiect `DatagramPacket` pregatindu-l pentru **receptia** unui pachet de lungime `length`, furnizandu-i tabloul de octeti `buf` in care sa fie plasate datele pachetului (`length` trebuie sa fie mai mic sau egal cu `buf.length`).
`DatagramPacket``(byte[] buf, int offset, int length)` Construieste un obiect `DatagramPacket` pregatindu-l pentru **receptia** unui pachet de lungime `length`, furnizandu-i tabloul de octeti `buf` in care sa fie plasate datele pachetului si specificandu-i indexul `offset` de la care sa inceapa plasarea datelor pachetului in tablou (`length+offset` trebuie sa fie mai mic sau egal cu `buf.length`).
`DatagramPacket``(byte[] buf, int length, InetAddress address, int port)` Construieste un obiect `DatagramPacket` pregatindu-l pentru **trimiterea** unui pachet de lungime `length` catre numarul de port UDP specificat (`port`) al gazdei cu adresa specificata (`address`), furnizandu-i tabloul de octeti `buf` din care sa fie preluate datele pachetului (`length` trebuie sa fie mai mic sau egal cu `buf.length`).
`DatagramPacket``(byte[] buf, int offset, int length, InetAddress address, int port)` Construieste un obiect `DatagramPacket` pregatindu-l pentru **trimiterea** unui pachet de lungime `length` catre numarul de port UDP specificat (`port`) al gazdei cu adresa specificata (`address`), furnizandu-i tabloul de octeti `buf` din care sa fie preluate datele pachetului si specificandu-i indexul `offset` de la care sa inceapa preluarea datelor pachetului din tablou (`length+offset` trebuie sa fie mai mic sau egal cu `buf.length`).

2. Declaratiile si descrierea catorva metode ale clasei **`DatagramPacket`**:

`InetAddress`	**`getAddress`**`()` Returneaza **adresa IP** sub forma de obiect `InetAddress` a masinii catre care pachetul curent va fi trimis sau de la care pachetul curent va fi receptionat.
`byte[]`	**`getData`**`()` Returneaza tabloul de octeti (*bufferul*) care contine **datele** pachetului curent.
`int`	**`getLength`**`()` Returneaza numarul de octeti (**lungimea** *bufferului*) de date de trimis sau de receptionat.
`int`	**`getOffset`**`()` Returneaza indexul (***offsetul***) la care sunt plasate datele de trimis sau de receptionat in tabloul de octeti.
`int`	**`getPort`**`()` Returneaza **numarul de port UDP** al masinii catre care pachetul curent va fi trimis sau de la care pachetul curent va fi receptionat.
`void`	**`setAddress`**`(InetAddress iaddr)` Stabileste **adresa IP** sub forma de obiect `InetAddress` a masinii catre care pachetul curent va fi trimis.

void	**setData**(byte[] buf) Stabileste tabloul de octeti (*bufferul*) care contine **datele** pachetului curent (implicit indexul `offset=0`).
void	**setData**(byte[] buf, int offset, int length) Stabileste tabloul de octeti (*bufferul*) care contine **datele** pachetului curent specificandu-i indexul `offset` de la care sa inceapa plasarea datelor pachetului in tablou.
void	**setLength**(int length) Stabileste numarul de octeti (**lungimea** *bufferului*) de date de trimis sau de receptionat.
void	**setPort**(int iport) Stabileste **numarul de port UDP** al masinii catre care pachetul curent va fi trimis.

In continuare este prezentata **secventa tipica pentru crearea unui pachet datagrama (UDP) pentru timitere**:

```
// Stabilirea adresei serverului
String adresaIP = "localhost";

// Stabilirea portului serverului
int portUDP = 2000;

// Crearea tabloului de octeti (bufferului) de date
String mesaj = "mesaj de trimis";
byte[] bufferDate = mesaj.getBytes();

// Crearea pachetului de trimis
try {
  DatagramPacket pachetUDP = new DatagramPacket(bufferDate,
        bufferDate.length, InetAddress.getByName(adresaIP), portUDP);
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
```

In continuare este prezentata **secventa tipica pentru crearea unui pachet datagrama (UDP) pentru receptie**:

```
// Crearea tabloului de octeti (bufferului) de date
byte[] bufferDate = new byte[4096];

// Crearea pachetului de primit
try {
  DatagramPacket pachetUDP = new DatagramPacket(bufferDate,
                                                bufferDate.length);
}
catch (UnknownHostException ex) {
  System.err.println(ex);
}
```

ca si **secventa tipica pentru citirea din pachetul UDP a datelor primite**:

```
// Obtinerea datelor pachetului ca tablou de octeti
bufferDate = pachetUDP.getData();

// Reconstructia mesajului text
String mesajPrimit =
             new String(bufferDate, 0, pachetUDP.getLength());
```

In continuare este prezentata o **aplicatie** care permite **obtinerea si afisarea informatiilor privind un pachet creat pentru trimitere**.

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4   *  Afisare informatii privind pachet UDP (datagrama)
 5   */
 6  public class InfoPachetUDP {
 7
 8    /**
 9     *  Afiseaza informatii privind pachetul UDP specificat ca parametru
10     */
11    public static void afisareInfoPachetUDP(DatagramPacket datagrama,
12                                            boolean deTrimis) {
13      System.out.println("\n -------------------------------------");
14      System.out.print("\n Pachetul UDP ");
15
16      if (deTrimis) System.out.print("adresat " + datagrama.getAddress() +
17                       " pe portul " + datagrama.getPort());
18
19      else  System.out.print("de la adresa " + datagrama.getAddress() +
20                " si portul " + datagrama.getPort());
21
22      System.out.println(" contine " + datagrama.getLength() +
23          " octeti de date:\n" + new String(datagrama.getData()));
24    }
25
26    /**
27     *  Testarea si exemplificarea modului de utilizare
28     */
29    public static void main(String args[]) throws IOException {
30      BufferedReader inConsola  = new BufferedReader(new
31                                      InputStreamReader(System.in));
32      System.out.print("Introduceti adresa IP dorita: ");
33      String adresaIP = inConsola.readLine();
34
35      System.out.print("Introduceti numarul de port dorit: ");
36      int portUDP = Integer.parseInt(inConsola.readLine());
37
38      System.out.print("Introduceti continutul pachetului: ");
39      byte[] bufferDate = inConsola.readLine().getBytes();
40      try {
41        DatagramPacket pachetUDP = new DatagramPacket(bufferDate,
42              bufferDate.length, InetAddress.getByName(adresaIP), portUDP);
43
44        afisareInfoPachetUDP(pachetUDP, true);
45      }
46      catch (UnknownHostException ex) {
47        System.err.println(ex);
48      }
49    }
50  }
```

3.4.3. Clasa socket UDP (DatagramSocket) – interfata publica

1. Principalii constructori ai clasei `DatagramSocket`:

`DatagramSocket``()` Construieste un *socket* datagrame si il leaga la un port UDP disponibil pe masina locala.
`DatagramSocket``(int port)` Construieste un *socket* datagrame si il leaga la portul UDP specificat pe masina locala.
`DatagramSocket``(int port, InetAddress laddr)` Construieste un *socket* datagrame, legat pe masina locala la portul UDP si adresa IP specificate.

2. Declaratiile si descrierea catorva metode ale clasei **`DatagramPacket`**:

`InetAddress`	**`getLocalAddress`**`()` Obtine adresa IP locala la care este legat *socketul* curent.
`int`	**`getLocalPort`**`()` Returneaza numarul de port local la care este legat *socketul* curent.
`boolean`	**`isBound`**`()` Returneaza o valoare logica indicand daca *socketul* curent este legat cu succes la o adresa locala.
`void`	**`receive`**`(DatagramPacket p)` Receptioneaza un pachet datagrama prin *socketul* curent.
`void`	**`send`**`(DatagramPacket p)` Trimite un pachet datagrama prin *socketul* curent.
`void`	**`close`**`()` Inchide *socketul* datagrama curent.
`boolean`	**`isClosed`**`()` Returneaza o valoare logica indicand daca *socketul* curent este inchis.
`void`	**`connect`**`(InetAddress address, int port)` Conecteaza *socketul* curent la adresa IP si numarul de port specificate (acestea actioneaza ca un filtru, prin *socketul* conectat putand fi trimise sau primite pachete doar catre respectiv de la adresa IP si numarul de port specificate). Implicit *socketul* este neconectat.
`InetAddress`	**`getInetAddress`**`()` Returneaza adresa IP la care *socketul* curent este conectat (`null` daca nu este conectat).
`int`	**`getPort`**`()` Returneaza numarul de port UDP la care *socketul* curent este conectat (`-1` daca nu este conectat).
`void`	**`disconnect`**`()` Deconecteaza *socketul* curent.
`boolean`	**`isConnected`**`()` Returneaza o valoare logica indicand daca *socketul* curent este conectat.
`boolean`	**`getBroadcast`**`()` Returneaza o valoare logica indicand daca SO_BROADCAST este validat.
`void`	**`setBroadcast`**`(boolean on)` Valideaza/invalideaza SO_BROADCAST.

`int`	**`getSoTimeout`**`()` Obtine valoarea optiunii SO_TIMEOUT.
`void`	**`setSoTimeout`**`(int timeout)` Stabileste valoarea optiunii SO_TIMEOUT la un *timeout* specificat, in millisecunde.
`int`	**`getReceiveBufferSize`**`()` Returneaza valoarea optiunii SO_RCVBUF pentru *socketul* curent, adica dimensiunea *bufferului* utilizat de platforma pentru pachetele primite de *socketul* curent.
`void`	**`setReceiveBufferSize`**`(int size)` Stabileste optiunea SO_RCVBUF pentru *socketul* curent la valoarea specificata.
`int`	**`getSendBufferSize`**`()` Returneaza valoarea optiunii SO_ SNDBUF pentru *socketul* curent, adica dimensiunea *bufferului* utilizat de platforma pentru pachetele trimise de *socketul* curent.
`void`	**`setSendBufferSize`**`(int size)` Stabileste optiunea SO_SNDBUF pentru *socketul* curent la valoarea specificata.
`int`	**`getTrafficClass`**`()` Obtine valoarea octetului clasa de trafic (QoS) sau tip de serviciu (ToS) in antetul datagramei IP care incapsuleaza datagramele UDP trimise prin *socketul* curent.
`void`	**`setTrafficClass`**`(int tc)` Stabileste valoarea octetului clasa de trafic (QoS) sau tip de serviciu (ToS) in antetul datagramei IP care incapsuleaza datagramele UDP trimise prin *socketul* curent.

In continuare este prezentata **secventa tipica pentru crearea *socket-ului* unei aplicatii client**:

```
// Crearea socketului
DatagramSocket socketUDP = new DatagramSocket();
```

ca si **pentru crearea *socket-ului* unei aplicatii server**:

```
// Stabilirea portului serverului
int portServer = 2000;

// Crearea socketului
DatagramSocket socketUDP = new DatagramSocket(portServer);
```

Socket-ul UDP poate fi utilizat pentru **trimiterea de date**:

```
// Trimiterea pachetului UDP
socketUDP.send(pachetDeTrimis);
```

sau pentru **primirea de date**:

```
// Receptia unui pachet UDP - blocare in asteptare
socketUDP.receive(pachetPrimit);
```

Dupa utilizare, *socket-ul* poate fi **inchis**:

```
// Inchiderea socketului
socketUDP.close();
```

Ca si in cazul *socket-urilor* flux, metoda **`setTrafficClass`**`(int tc)` **stabileste clasa de trafic** (*traffic class*) **sau tipul de serviciu** (ToS) **din antetul IP al pachetelor trimise prin *socket-ul* curent**.

Deoarece implementarea nivelului retea poate ignora valoarea parametrului `tc` (in cazul in care **nu exista sau nu este activat controlul de trafic pentru servicii diferentiate**), **aplicatia trebuie sa interpreteze apelul ca pe o sugestie** (*hint*) data nivelului inferior.

Pentru protocolul IPv4 valoarea parametrului **`tc`** este interpretata drept **campurile precedenta si ToS**. **Campul ToS este creat ca set de biti** obtinut prin **aplicarea functiei logice OR valorilor**:

`IPTOS_LOWCOST = 0x02` - indicand cerinte de **cost redus** din partea aplicatiei
`IPTOS_RELIABILITY = 0x04` - indicand cerinte de **fiabilitate** din partea aplicatiei
`IPTOS_THROUGHPUT = 0x08` - indicand cerinte de **banda larga** din partea aplicatiei
`IPTOS_LOWDELAY = 0x10` - indicand cerinte de intarziere redusa (**timp real**) ale aplicatiei

Cel mai putin semnificativ bit al octetului **`tc`** e ignorat, el **trebuind sa fie zero** (*must be zero* – MZB).

Stabilirea bitilor in campul de precedenta (cei mai semnificativi 3 biti ai octetului **`tc`**) poate produce o **`SocketException`**, indicand imposibilitatea realizarii operatiei.

Pentru protocolul IPv6 valoarea parametrului **`tc`** este plasata in **campul `sin6_flowinfo`** al antetului IP.

3.4.4. Programe ilustrative pentru lucrul cu *socket*-uri datagrame (UDP)

Urmatorul program utilizeaza crearea de ***socket* UDP** pentru a **identifica porturile locale pe care exista conexiuni** (**pe care nu pot fi create servere**).

```
 1  import java.net.*;
 2  public class ScannerPorturiUDPLocale {
 3    public static void main (String args[]) {
 4      DatagramSocket serverUDP;
 5
 6      for (int portUDP=1; portUDP < 65536; portUDP++) {
 7        try {
 8          // Urmatoarele linii vor genera exceptie prinsa de blocul catch
 9          // in cazul in care e deja un server pe portul portUDP
10          serverUDP = new DatagramSocket(portUDP);
11          serverUDP.close();
12        }
13        catch (SocketException ex) {
14          System.err.println("Exista un server UDP pe portul " + portUDP);
15        }
16      }
17    }
18  }
```

Programul `ClientEcouDatagrameRepetitiv`, (**client pentru un server ecou UDP care permite trimiterea mai multor mesaje,** mesajul format dintr-un punct (`"."`) semnaland serverului terminarea mesajelor de trimis, clientul urmand sa isi termine executia):

```
 1  import java.net.*;
 2  import java.io.*;
 3  public class ClientEcouDatagrameRepetitiv {
 4    public static void main (String args[]) throws IOException {
 5      BufferedReader inConsola  = new BufferedReader(new
 6                                      InputStreamReader(System.in));
 7      System.out.print("Introduceti adresa IP a serverului: ");
 8      String adresaServer = inConsola.readLine();
 9      System.out.print("Introduceti numarul de port al serverului: ");
10      int portServer = Integer.parseInt(inConsola.readLine());
11
12      byte[] bufferDate = new byte[1024];
13
14      DatagramPacket pachetDeTrimis;
15      DatagramPacket pachetPrimit = new DatagramPacket(bufferDate, 1024);
16
17      // Crearea socketului
18      DatagramSocket socketUDP = new DatagramSocket();
19      System.out.println("Client pentru serverul " + adresaServer + ":" +
20                         portServer + " lansat...");
21
22      // Utilizare socket client
23      while(true) {
24        // Citirea unei linii de la consola de intrare
25        System.out.print("Mesaj de trimis: ");
26        String mesajDeTrimis = inConsola.readLine();
27
28        // Plasarea datelor mesajului de trimis in buffer
29        bufferDate = mesajDeTrimis.getBytes();
30
31        // Constructia pachetului UDP de trimis
32        pachetDeTrimis=new DatagramPacket(bufferDate, mesajDeTrimis.length(),
33               InetAddress.getByName(adresaServer), portServer);
34
35        // Trimiterea pachetului UDP
36        socketUDP.send(pachetDeTrimis);
37
38        // Receptia unui pachet UDP - blocare in asteptare
39        socketUDP.receive(pachetPrimit);
40
41        // Obtinerea datelor pachetului ca tablou de octeti
42        bufferDate = pachetPrimit.getData();
43
44        // Reconstructia mesajului text
45        String mesajPrimit =
46                   new String(bufferDate, 0, pachetPrimit.getLength());
47
48        // Afisarea informatiilor primite
49        System.out.println("Mesaj primit de la " +
50                pachetPrimit.getAddress().getHostName() + ":" +
51                pachetPrimit.getPort() + ": \n" + mesajPrimit);
52
53        // Testarea conditiei de oprire
54        if (mesajPrimit.equals(".")) break;
55      }
56      // Inchiderea socketului
57      socketUDP.close();
58    }
59  }
```

Programul `ServerEcouDatagrameRepetitiv`, (**server care permite receptia si trimiterea in ecou a mai multor mesaje sosite de la un singur client**, mesajul format dintr-un punct (".") semnaland serverului terminarea mesajelor de trimis, serverul urmand sa isi termine executia):

```
 1  import java.net.*;
 2  import java.io.*;
 3  public class ServerEcouDatagrameRepetitiv {
 4    public static void main (String args[]) throws IOException {
 5      BufferedReader inConsola  = new BufferedReader(new
 6                                      InputStreamReader(System.in));
 7      System.out.print("Introduceti numarul de port al serverului: ");
 8      int portServer = Integer.parseInt(inConsola.readLine());
 9
10      // Crearea socketului
11      DatagramSocket socketUDP = new DatagramSocket(portServer);
12      System.out.println("Server in asteptare pe portul "+portServer+ "...");
13
14      byte[] bufferDate = new byte[1024];
15
16      DatagramPacket pachetDeTrimis;
17      DatagramPacket pachetPrimit = new DatagramPacket(bufferDate, 1024);
18
19      // Utilizare socket server
20      while(true) {
21        // Receptia unui pachet UDP - blocare in asteptare
22        socketUDP.receive(pachetPrimit);
23
24        // Obtinerea datelor pachetului ca tablou de octeti
25        bufferDate = pachetPrimit.getData();
26
27        // Reconstructia mesajului text
28        String mesajPrimit =
29              new String(bufferDate, 0, pachetPrimit.getLength());
30
31        // Afisarea informatiilor primite
32        System.out.println("Mesaj primit de la " +
33                 pachetPrimit.getAddress().getHostName() + ":" +
34                 pachetPrimit.getPort() + ": \n" + mesajPrimit);
35
36        // Constructia pachetului UDP de trimis
37        pachetDeTrimis = new DatagramPacket(pachetPrimit.getData(),
38                                 pachetPrimit.getLength(),
39                                 pachetPrimit.getAddress(),
40                                 pachetPrimit.getPort());
41
42        // Trimiterea pachetului UDP
43        socketUDP.send(pachetDeTrimis);
44
45        // Testarea conditiei de oprire a servirii
46        if (mesajPrimit.equals(".")) break;
47      }
48      // Inchidere socket
49      socketUDP.close();
50    }
51  }
```

Pentru constructia datagramelor complexe pot fi utilizate **fluxurile `ByteArrayOutputStream` si `DataOutputStream`.**

Urmatoarea clasa contine metode utile crearii pachetelor UDP folosind fluxurile `ByteArrayOutputStream` si `DataOutputStream`.

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5   *  Metode utile lucrului cu pachete UDP (datagrame) de trimis
 6   */
 7  public class UtilePachetTrimitereUDP {
 8
 9    ByteArrayOutputStream outTablou;
10    DataOutputStream      outDateFormatate;
11    byte[]                bufferDate;
12
13    /**
14     *  Construieste pachet UDP de trimitere cu continut text
15     */
16    public static DatagramPacket constructiePachetString(String text,
17                         String adresa, int port) throws IOException {
18      return new DatagramPacket(text.getBytes(), text.length(),
19                                InetAddress.getByName(adresa), port);
20    }
21
22    /**
23     *  Incepe constructia unui pachet UDP de trimitere complex
24     */
25    public DataOutputStream obtinereFluxDateFormatate() throws IOException{
26      outTablou = new ByteArrayOutputStream();
27      outDateFormatate = new DataOutputStream(outTablou);
28      return outDateFormatate;
29    }
30
31    /**
32     *  Incheie constructia unui pachet UDP de trimitere complex
33     */
34    public DatagramPacket incheiereConstructiePachet(String adresa,
35                                                     int port) throws IOException {
36      outDateFormatate.flush();
37      bufferDate = outTablou.toByteArray();
38      return new DatagramPacket(bufferDate, bufferDate.length,
39                                InetAddress.getByName(adresa), port);
40    }
41  }
```

Pentru extragerea datelor din datagramele complexe pot fi utilizate **fluxurile fluxurile `ByteArrayInputStream` si `DataInputStream`.**

Urmatoarea clasa contine metode utile extragerii datelor din pachetele UDP folosind fluxurile `ByteArrayInputStream` si `DataInputStream`.

```
 1  Import java.net.*;
 2  import java.io.*;
 3  /**
 4   *  Metode utile lucrului cu pachete UDP (datagrame) de primit
 5   */
 6  public class UtilePachetPrimireUDP {
 7
 8    DatagramPacket       datagrama;
 9    ByteArrayInputStream inTablou;
10    DataInputStream      inDateFormatate;
11    byte[]               bufferDate;
12
13    /**
14     *  Obtine String din pachet UDP cu continut text
15     */
16    public static String extrageString(DatagramPacket datagrama)
17                                                    throws IOException {
18      return new String(datagrama.getData(), 0, datagrama.getLength());
19    }
20
21    /**
22     *  Pregateste extragerea dintr-un pachet UDP complex
23     */
24    public DataInputStream obtineFluxDateFormatate(DatagramPacket datagrama)
25                                                    throws IOException {
26      inTablou = new ByteArrayInputStream(datagrama.getData(),
27                       datagrama.getOffset(), datagrama.getLength());
28      inDateFormatate = new DataInputStream(inTablou);
29      return inDateFormatate;
30    }
31  }
```

Urmatorul program client (`TestUtilePachetTrimitereUDP`) ilustreaza modul de utilizare al metodelor din clasele utilitare anterioare.

```
 1  import java.net.*;
 2  import java.io.*;
 3  /**
 4   *  Client test metode utile lucrului cu pachete UDP (datagrame) de trimis
 5   */
 6  public class TestUtilePachetTrimitereUDP {
 7
 8    /**
 9     *  Testarea/exemplificarea utilizarii clasei UtilePachetTrimitereUDP
10     */
11    public static void main (String args[]) throws IOException {
12      BufferedReader inConsola  = new BufferedReader(new
13                                      InputStreamReader(System.in));
14      System.out.print("Introduceti adresa IP a serverului: ");
15      String adresaServer = inConsola.readLine();
16      System.out.print("Introduceti numarul de port al serverului: ");
17      int portServer = Integer.parseInt(inConsola.readLine());
18
19      DatagramSocket socketUDP = new DatagramSocket();
20      System.out.println("Client pentru serverul " + adresaServer + ":" +
21                         portServer + " lansat...");
22
23      byte[] bufferDate = new byte[1024];
24
25      DatagramPacket pachetDeTrimis;
26
27      System.out.print("Mesaj de trimis: ");
28      String mesajDeTrimis = inConsola.readLine();
29
30      pachetDeTrimis = UtilePachetTrimitereUDP.constructiePachetString(
31                                  mesajDeTrimis, adresaServer, portServer);
32
33      InfoPachetUDP.afisareInfoPachetUDP(pachetDeTrimis, true);
34
35      socketUDP.send(pachetDeTrimis);    // Trimiterea pachetului UDP
36
37      UtilePachetTrimitereUDP utileTrimitere = new UtilePachetTrimitereUDP();
38
39      DataOutputStream flux = utileTrimitere.obtinereFluxDateFormatate();
40
41      byte octet = 100;
42      flux.writeByte(octet);
43
44      short intregScurt = 1000;
45      flux.writeShort(intregScurt);
46
47      flux.writeInt(100000);
48
49      flux.writeLong(1000000L);
50
51      flux.writeUTF(mesajDeTrimis + " bis");
52
53      pachetDeTrimis= utileTrimitere.incheiereConstructiePachet(adresaServer,
54                                                                portServer);
55      InfoPachetUDP.afisareInfoPachetUDP(pachetDeTrimis, true);
56
57      socketUDP.send(pachetDeTrimis);    // Trimiterea pachetului UDP
58
59      socketUDP.close();
60    }
61  }
```

Urmatorul program server (`TestUtilePachetPrimireUDP`) ilustreaza modul de utilizare al metodelor din clasele utilitare anterioare.

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5   *  Server test metode utile lucrului cu pachete UDP (datagrame) de primit
 6   */
 7  public class TestUtilePachetPrimireUDP {
 8
 9    /**
10     *  Testarea/exemplificarea utilizarii clasei UtilePachetPrimireUDP
11     */
12    public static void main (String args[]) throws IOException {
13      BufferedReader inConsola  = new BufferedReader(new
14                                      InputStreamReader(System.in));
15
16      System.out.print("Introduceti numarul de port al serverului: ");
17      int portServer = Integer.parseInt(inConsola.readLine());
18
19      DatagramSocket socketUDP = new DatagramSocket(portServer);
20      System.out.println("Server in asteptare pe portul "+portServer+"...");
21
22      byte[] bufferDate = new byte[1024];
23
24      DatagramPacket pachetPrimit = new DatagramPacket(bufferDate, 1024);
25
26      // Receptia unui pachet UDP - blocare in asteptare
27      socketUDP.receive(pachetPrimit);
28
29      InfoPachetUDP.afisareInfoPachetUDP(pachetPrimit, false);
30
31      String text = UtilePachetPrimireUDP.extrageString(pachetPrimit);
32      System.out.println("Text primit: " + text);
33
34      // Receptia unui pachet UDP - blocare in asteptare
35      socketUDP.receive(pachetPrimit);
36
37      InfoPachetUDP.afisareInfoPachetUDP(pachetPrimit, false);
38
39      UtilePachetPrimireUDP utilePrimire = new UtilePachetPrimireUDP();
40
41      DataInputStream flux =
42            utilePrimire.obtineFluxDateFormatate(pachetPrimit);
43
44      byte octet = flux.readByte();
45      System.out.println("Octet primit: " + octet);
46
47      short intregScurt = flux.readShort();
48      System.out.println("Intreg scurt primit: " + intregScurt);
49
50      int intreg = flux.readInt();
51      System.out.println("Intreg primit: " + intreg);
52
53      long intregLung = flux.readLong();
54      System.out.println("Intreg lung primit: " + intregLung);
55
56      String text2 = flux.readUTF();
57      System.out.println("Text primit: " + text2);
58
59      socketUDP.close();
60    }
61  }
```

4. Elemente de programare Java pentru Web

4.1. Introducere in arhitectura si tehnologiile Web

- HTTP, HTML
- agent client HTTP (browser)
- server HTTP (Web)

4.2. Identificarea si accesul la resursele retelelor IP

4.2.1. Caracteristicile URL-*urilor*

URL e un **acronim** pentru ***U**niform **R**esource **L**ocator*. URL-ul este **referinta** (**adresa**) **unei resurse din *Internet***.

Programele Java care interactioneaza cu *Internetul pot utiliza URL-uri pentru a gasi resursele din Internet pe care vor sa le acceseze*.

Programele Java **utilizeaza o clasa** numita **URL** din pachetul `java.net` care reprezinta o adresa URL. Un URL ia forma unui ***sir de caractere*** (*string*) ***care descrie modul in care se poate gasi o resursa din Internet***.

URL-urile au **doua componente principale**:
- protocolul necesar pentru a accesa resursa (*protocol identifier*) si
- locatia resursei (*resource name*).

Un **exemplu de URL** care ***adreseaza site-ul Web Java gazduit de Sun Microsystems***:

http :// www.sun.com

Identificatorul protocolului si numele resursei sunt **separate de simbolul doua puncte si de doua *slash-uri***.

Exemplul utilizeaza protocolul *Hypertext Transfer Protocol* (HTTP), care este utilizat in general pentru a obtine documente *hypertext*. Alte protocoale posibile sunt: *File Transfer Protocol* (FTP), *Gopher*, *File*, si *News*.

Numele resursei este adresa completa a resursei. Formatul numelui resursei depinde complet de protocolul utilizat, dar pentru multe protocoale, incluzand HTTP, the numele resursei contine una sau mai multe componente listate in tabelul urmator:

Numele gazdei (*host name*)	Numele masinii (calculatorului) pe care se afla resursa.
Numele fisierului (*filename*)	Numele caii catre fisier pe respectiva masina.
Numarul portului (*port number*)	Numarul de port la care se conecteaza (in general optional).
Referinta (*reference*)	O referinta catre o ancora cu nume intr-o resursa care identifica o locatie specifica intr-un fisier (in general optional).

De exemplu, numele resusei pentru un URL HTTP **trebuie *sa specifice un server*** din retea (*host name*) ***si calea catre document*** pe acea masina (*filename*), si **poate *sa specifice un numar de port*** si ***o referinta***. In URL-ul pentru *site-ul Web* Java gazduit de *Sun Microsystems*, **java.sun.com** este numele gazdei (*host name*) iar ***slash-ul*** este o prescurtare (*shorthand*) pentru fisierul numit **/index.html**.

4.2.2. Crearea unui URL

In programele Java, se poate crea un obiect URL care reprezinta o adresa URL. Obiectul URL refera intotdeauna un URL absolut dar care poate fi construit dintr-un URL absolut, dintr-un URL, sau din componente URL.

Calea cea mai simpla de a crea un obiect `URL` este dintr-un `String` care reprezinta forma interpretabila de catre utilizator a adresei URL. Aceasta este in general forma pe care o alta persoana o va folosi pentru un URL. De exemplu, URL-ul pentru *site-ul* Gamelan, care este un server de resurse Java, ia urmatoarea forma:

```
http://www.gamelan.com/
```

In programul Java, se poate folosi un `String` continand acest text pentru a crea un obiect `URL`:

```
URL gamelan = new URL("http://www.gamelan.com/");
```

Obiectul `URL` astfel creat reprezinta un *URL absolut* (contine toate informatiile necesare pentru a referi resursa in chestiune). Se pot crea obiecte `URL` si dintr-o adresa *URL relativa*.

Un URL relativ contine doar informatiile necesare pentru a referi o resursa relativ la alt URL (sau in contextul altui URL).

Specificatiile URL relative sunt adesea utilizate in interiorul fisierelor HTML. De exemplu, presupunand faptul ca se doreste scrierea unui fisier HTML numit `JoesHomePage.html`. In interiorul acestei pagini, sunt *link-uri* catre alte pagini, `PicturesOfMe.html` si `MyKids.html`, care sunt stocate pe aceeasi masina si in acelasi director ca si `JoesHomePage.html`. *Link-urile* catre `PicturesOfMe.html` si `MyKids.html` din `JoesHomePage.html` pot fi specificate doar ca nume de fisiere, astfel:

```
<a href="PicturesOfMe.html"Pictures of Me</a
<a href="MyKids.html"Pictures of My Kids</a
```

Aceste adrese URL sunt *URL-uri relative*, adica URL-urile sunt specificate relativ la fisierul in care sunt continute (`JoesHomePage.html`).

In programele Java se poate crea un obiect `URL` dintr-o specificatie URL relativa. Presupunand de exemplu faptul ca sunt cunoscute urmatoarele doua URL-uri pe *site-ul* Gamelan:

```
http://www.gamelan.com/pages/Gamelan.game.html
http://www.gamelan.com/pages/Gamelan.net.html
```

se pot crea obiecte `URL` pentru aceste pagini relativ la partea comuna a URL-urilor (baza) care le refera (**`http://www.gamelan.com/pages/`**)astfel:

```
URL gamelan = new URL("http://www.gamelan.com/pages/");
URL gamelanGames = new URL(gamelan, "Gamelan.game.html");
URL gamelanNetwork = new URL(gamelan, "Gamelan.net.html");
```

Acest cod utilizeaza **constructorul `URL` care permite crearea unui obiect `URL` dintr-un alt obiect `URL` (baza) si o specificatie URL relativa**.

Forma generala a acestui constructor este:

```
URL(URL bazaURL, String relativURL)
```

- daca **`bazaURL`** este **null**, atunci acest constructor ***trateaza `relativURL` ca o specificatie URL absoluta***.
- daca **`relativURL`** este o **specificatie URL absoluta**, atunci constructorul ***ignora `baseURL`***.

Acest constructor e **util si pentru crearea obiectelor `URL` pentru ancore cu nume** (numite si **referinte**) **in interiorul unui fisier**. Presupunand de exemplu faptul ca fisierul `Gamelan.network.html` are o ancora cu nume numita `BOTTOM` la inceputul fisierului, se poate utiliza constructorul bazat pe URL-uri relative pentru a crea un obiect `URL` din el astfel:

```
URL gamelanNetworkBottom = new URL(gamelanNetwork, "#BOTTOM");
```

Clasa `URL` ofera doi constructori aditionali pentru a crea un obiect `URL`, utili atunci cand se lucreaza cu URL-uri, de tipul celor HTTP, care au nume gazda (*host name*), nume fisier (*filename*), numar port (*port number*), si componente referinta (*reference components*) in portiunea nume al resursei (*resource name*) a URL-ului.

Constructorii sunt utili atunci cand nu e detinut un String care sa contina specificatia completa a URL-ului, dar sunt cunoscute diverse componente ale URL-ului.

De exemplu, codul celui de-al doilea constructor pentru obiecte `URL`:

```
new URL("http", "www.gamelan.com", "/pages/Gamelan.net.html");
```

e echivalent cu:

```
new URL("http://www.gamelan.com/pages/Gamelan.net.html");
```

primul argument fiind protocolul, al doilea numele gazdei (*host name*), iar ultimul numele caii si al fisierului. Numele fisierului incepe cu un *slash* ceea ce indica specificarea de la radacina.

Ultimul constructor `URL` adauga numarul portului la lista de argumente a constructorului anterior:

```
URL gamelan = new URL("http", "www.gamelan.com", 80,
                      "pages/Gamelan.network.html");
```

ceea ce creaza un obiect `URL` pentru urmatorul URL:

```
http://www.gamelan.com:80/pages/Gamelan.network.html
```

Pentru un obiect `URL` construit utilizand unul dintre acesti constructori, se poate obtine un `String` continand adresa URL completa utilizand una dintre metodele `toString` si `toExternalForm` (echivalente) ale obiectului `URL`.

Observatie: `URL`-urile sunt obiecte nemodificabile (*immutable* sau *write-once*). Odata creat un obiect `URL`, nu poate fi schimbat nici un element al lui (*protocol, host name, filename*, sau *port number*).

4.2.3. Analiza lexicala a unui URL

In Java nu mai este necesara analiza si separarea lexicala a unui URL (*URL parsing*) pentru a gasi host name, filename, si alte informatii. Cu un obiect URL valid se pot apela toate metodele accesor pentru a obtine toate aceste informatii despre URL fara a mai face analiza lexicala !

Clasa `URL` ofera mai multe metode care permit interogarea obiectelor `URL`. Se pot obtine protocolul (*protocol*), numele gazdei (*host name*), numele fisierului (*filename*) si numarul portului (*port number*) dintr-un URL utilizand una dintre urmatoarele metode accesor:

getProtocol()

Returneaza componenta identificator de protocol a URL-ului.

getHost()

Returneaza componenta nume gazda (*host name*)a URL-ului.

getPort()

Returneaza componenta numar port (*port number*) a URL-ului. Metoda `getPort` returneaza un intreg care este numarul portului. Daca portul nu e configurat, `getPort` returneaza -1.

getFile()

Returneaza componenta nume fisier (*filename*) a URL-ului.

getRef()

Returneaza componenta referinta a URL-ului.

Observatie: Nu toate adresele URL contin aceste componente. Clasa URL ofera aceste metode deoarece URL-urile HTTP contin aceste componente ca si, probabil, cele mai multe dintre URL-urile utilizate. Clasa URL este intr-o oarecare masura centrata pe HTTP.

4.2.4. Citirea direct dintr-un URL

Programele Java pot citi dintr-un URL utilizand metoda `openStream()`.

Dupa crearea cu succes a unui obiect `URL`, se poate apela metoda sa **openStream()** pentru a obtine un flux din care se poate citi continutul URL-ului. Metoda openStream() returneaza un obiect java.io.InputStream, astfel incat citirea dintr-un URL devine la fel de usoara ca citirea dintr-un flux de intrare.

Urmatorul program Java utilizeaza metoda openStream() pentru a obtine un flux de intrare de la URL-ul http://www.yahoo.com/. El deschide un BufferedReader de la un flux de intrare si citeste din BufferedReader ca si cum ar citi din URL. Tot de este citit este copiat apoi in fluxul standard de iesire (System.out):

```
 1    import java.net.*;
 2    import java.io.*;
 3
 4    public class URLReader {
 5        public static void main(String[] args) throws Exception {
 6            URL yahoo = new URL("http://www.yahoo.com/");
 7            BufferedReader in = new BufferedReader(
 8                    new InputStreamReader(yahoo.openStream()));
 9            String inputLine;
10
11            while ((inputLine = in.readLine()) != null)
12                System.out.println(inputLine);
13
14            in.close();
15        }
16    }
```

Cand se executa programul, vor putea fi vazute in fereastra de comanda MS-DOS, comenzile HTML si continutul textual din fisierul HTML aflat la http://www.yahoo.com/.

4.2.5. Conectarea la un URL

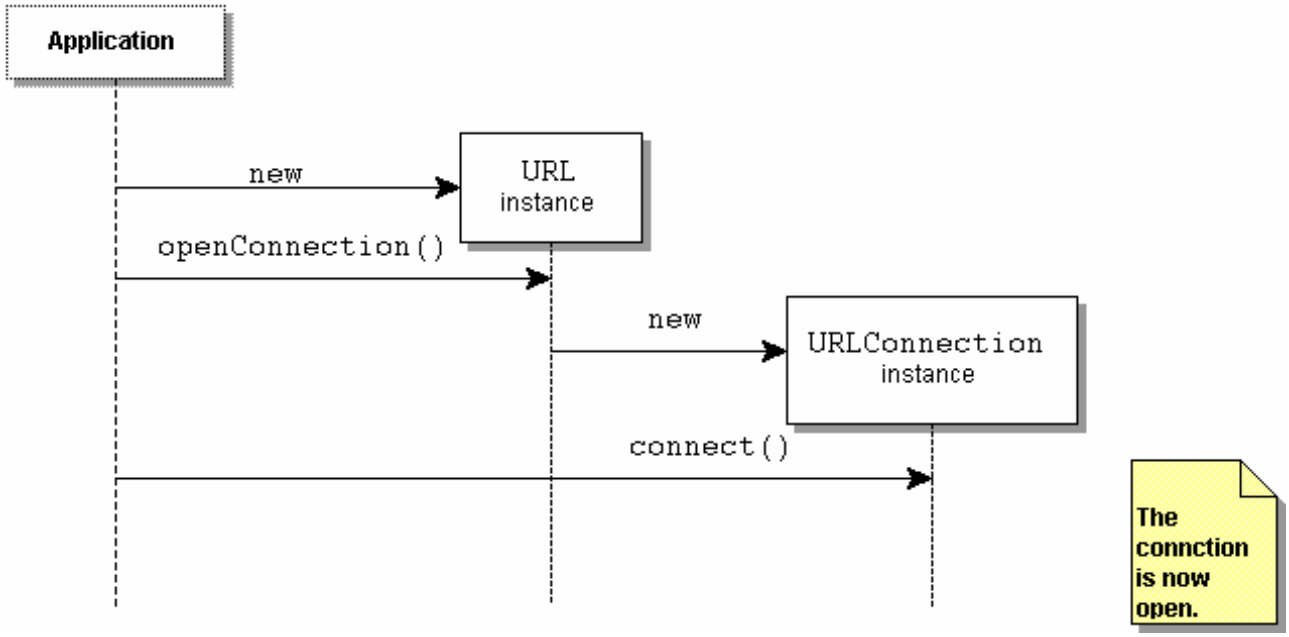
Pentru a realiza mai mult decat simpla citire dintr-un URL, poate fi realizata conectarea la el prin apelul metodei **openConnection()** a obiectului URL. Metoda openConnection() returneaza un obiect **URLConnection** care poate fi utilizat pentru o comunicatie mai generala cu URL-ul, cum ar fi citirea din el, scrierea in el, sau interogarea lui privind continutul si alte informatii.

Dupa crearea cu succes a obiectului `URL`, se poate apela metoda openConnection() a obiectului `URL` pentru conectarea la el. Prin conectarea la un `URL`, se initializeaza o legatura de comunicatie intre programul Java si URL peste retea. De exemplu, se poate deschide o conexiune catre *site-ulYahoo* cu urmatorul cod:

```
try {
    URL yahoo = new URL("http://www.yahoo.com/");
    yahoo.openConnection();
} catch (MalformedURLException e) {     // new URL() failed
    . . .
} catch (IOException e) {               // openConnection() failed
    . . .
}
```

Daca e posibil, metoda openConnection() creaza un nou obiect URLConnection (daca nu exista deja unul potrivit), il initializeaza, si se conecteaza la URL, si returneaza obiectul URLConnection. Daca, de exemplu, serverul *Yahoo* nu functioneaza, atunci metoda openConnection() arunca o exceptie IOException.

Dupa conectarea cu succes la URL, poate fi utilizat obiectul URLConnection pentru a realiza actiuni cum ar fi citirea de pe si scrierea pe o conexiune.



4.3. Applet-uri (miniaplicatii) Java

4.3.1. Caracteristicile *applet-urilor* Java

Applet-urile sau miniaplicatiile Java sunt portiuni de cod Java care mostenesc clasa **`Applet`**. Prin plasarea lor in *browser-e*, *applet-urile* devin **panouri frontale ale serviciilor distribuite oferite de *Web***.

Applet-urile sunt **mai intai incarcate in *browsere***, fiind **apoi executate in mediul de executie oferit de acesta**.

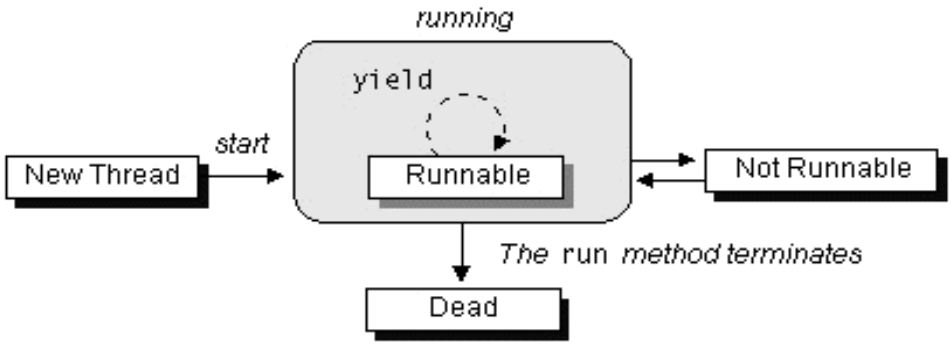
Applet-urile **nu sunt aplicatii complete**, ci **componente care ruleaza in mediul *browser-ului***.

Browser-ul* actioneaza ca un *framework pentru executia *applet-urile* (componentelor Java). ***Browser-ul* informeaza *applet-ul*** asupra evenimentelor care se petrec pe durata de viata a *applet-ului*. **Serviciile oferite de *browser*** sunt:

- **controlul total al ciclului de viata** al *applet-ului*,
- **furnizarea informatiilor privind atributele** din *tag-ul* APPLET,
- **functia de program/proces principal** din care se executa *applet-urile* (ofera functia **main()**).

4.3.2. Ciclul de viata al *applet-urilor* Java

Clasa `Applet` si interfata `Runnable` definesc metode pe care un *browser* le poate invoca pe durata ciclului de viata al unui *applet*.



Browser-ul* invoca*:

- **`init()`** cand ***incarca applet-ul prima oara***;
- **`start()`** cand ***un utilizator intra*** sau ***reintra in pagina care contine applet-ul***;
- **`stop()`** cand ***utilizatorul iese din pagina***;
- **`destroy()`** ***inaintea terminarii normale.***

Invocarea ultimelor doua metode conduce la **"omorarea" tuturor firelor de executie** ale *applet-ului* si la **eliberarea tuturor resurselor** *applet-ului*.

Pentru a avea interactivitate avansata, *applet-ul* trebuie sa implementeze metoda **`run()`** a interfetei **`Runnable`** sau metoda **`run()`** a clasei `Thread`, **care se executa in interiorul unui fir de executie** (*thread*).

Urmatorul *applet* simplu permite, **prin vizualizarea lui**, urmarirea **fazelor ciclului de viata ale unui *applet***.

```
 1  import java.applet.Applet;
 2  import java.awt.Graphics;
 3
 4  public class Simple extends Applet {
 5      StringBuffer buffer;
 6
 7      public void init() {
 8          buffer = new StringBuffer();
 9          addItem("initializing... ");
10      }
11
12      public void start() {
13          addItem("starting... ");
14      }
15
16      public void stop() {
17          addItem("stopping... ");
18      }
19
20      public void destroy() {
21          addItem("preparing for unloading...");
22      }
23
24      void addItem(String newWord) {
25          System.out.println(newWord);
26          buffer.append(newWord);
27          repaint();
28      }
29
30      public void paint(Graphics g) {
31          //Draw a Rectangle around the applet's display area.
32          g.drawRect(0, 0, size().width - 1, size().height - 1);
33          //Draw the current string inside the rectangle.
34          g.drawString(buffer.toString(), 5, 15);
35      }
36  }
```

4.3.3. Dezvoltarea unui *applet* Java

Miniaplicatia **FirstApplet** foloseste componentele multimedia integrate ale limbajului Java pentru afisarea unei imagini si redarea unui fisier de sunet.

```
1       import java.awt.*;
2       import java.applet.*;
3
4       public class FirstApplet extends Applet {
5             Image NewImage;
6
7             public void init() {
8               resize(400,400);
9               NewImage = getImage(getCodeBase(),"New.gif");
10            }
11
12            public void paint(Graphics g) {
13              g.drawString("Hello!", 200, 200);
14              g.drawImage(NewImage, 40, 40, this);
15              play(getCodeBase(),"New.au");
16            }
17      }
18
```

Instructiunea **import** permite miniaplicatiei sa foloseasca metode si clase din alte pachete. In mod prestabilit, toate programele Java importa pachetul **java.lang**, care contine functiile de baza ale limbajului Java. Asteriscul de la sfarsitul instructiunii import permite importul dinamic al claselor Java. In acest exemplu, sunt importate dinamic clasele din pachetele **java.awt** si **java.applet**.

Linia urmatoare declara o clasa numita **FirstApplet** care extinde clasa **`Applet`**:

```
public class FirstApplet extends Applet {
```

Prin extinderea clasei **`Applet`**, **FirstApplet** mosteneste functionalitatea acestei clase. Acolada deschisa marcheaza inceputul clasei **FirstApplet**.

Linia urmatoare initializeaza variabila **NewImage** si o declara de a fi de tipul Image. In acest caz, **NewImage** are rolul unui substituent al imaginii care va fi afisata:

```
Image NewImage;
```

Linia urmatoare declara o metoda numita **`init()`**, care redefineste metoda clasei **`Applet`**:

```
public void init() {
```

Metoda **`init()`** a clasei Applet este redefinita, astfel incat sa puteti redimensiona fereastra inainte de afisarea imaginii. Modificatorul public specifica faptul ca metoda este accesibila altor clase. Modificatorul void specifica faptul ca metoda nu returneaza nici o valoare. In mod normal, argumentele acceptate de o metoda sunt incadrate de paranteze rotunde. Deoarece metoda **`init()`** nu accepta argumente, intre paranteze nu apare nimic.

Folosind metoda **`resize()`**, puteti sa redimensionati zona de afisare a miniaplicatiei. In acest exemplu, dimensiunea zonei de afisare este stabilita la 400x400 pixeli:

```
resize(400,400);
```

Dupa ce ati declarat o variabila de un anumit tip, puteti sa o folositi. Linia urmatoare stabileste o valoare pentru variabila NewImage:

```
NewImage = getImage(getCodeBase(), "New.gif");
```

Pentru aceasta, este folosita metoda **getImage( )**. Primul argument al metodei este un apel al metodei **`getCodeBase()`**, care returneaza pozitia directorului de baza sau a directorului curent de pe hard-disc. Directorul de baza este directorul care contine fisierul clasei pe care o rulati. Al doilea argument este numele imaginii care poate fi gasita in pozitia specificata.

Urmatoarea linie de cod declara o metoda numita **paint( )**, care redefineste metoda **paint( )** din pachetul AWT. Metoda **paint( )** este redefinita pentru a permite miniaplicatiei sa afiseze imaginea intr-o anumita pozitie pe ecran. Modificatorul public specifica faptul ca metoda este accesibila altor clase. Modificatorul void specifica faptul ca metoda nu returneaza nici o valoare. La apelarea metodei **paint()**, trebuie folosit ca parametru un obiect al clasei **Graphics**.

```
public void paint (Graphics g) {
```

Graphics este o ***clasa de baza abstracta pentru toate obiectele grafice***. Elementul g reprezinta fereastra de tip **Graphics** specificata. Linia urmatoare apeleaza obiectul g, de tip **Graphics**, pentru afisarea imaginii **NewImage:**

```
g.drawImage(NewImage,40,40,this);
```

Metoda care realizeaza de fapt operatia se numeste **drawImage( )**. Metoda **drawImage( )** accepta argumente prin care i se precizeaza ce imagine trebuie sa afiseze si unde. In acest exemplu, obiectul **NewImage** este afisat in punctul de coordonate {40, 40}.

Asa cum sugereaza si numele sau metoda **play( )** este folosita pentru redarea fisierelor de sunet. Primul argument al metodei **play( )** este un apel al metodei `getCodeBase()`, care returneaza pozitia directorului de baza sau a directorului curent de pe *hard-disc*.

```
play(getCodeBase(), "New.au");
```

Directorul de baza este directorul care contine fisierul clasei pe care o rulati. Al doilea argument este numele fisierului de sunet care poate fi gasit in pozitia specificata.

Programul **FirstApplet** trebuie **editat** intr-un fisier numit **FirstApplet.java**. Acesta va fi salvat ca fisier de text ASCII standard.

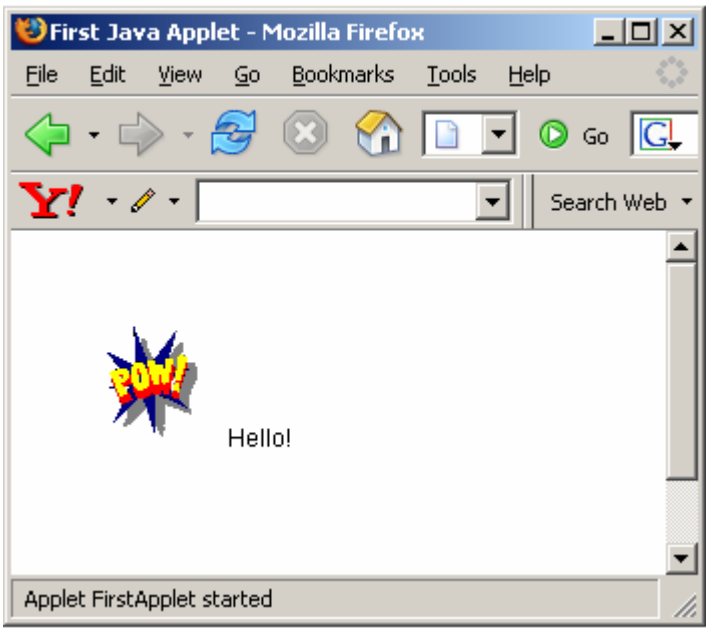
Compilarea unei miniaplicatii se realizeaza la fel ca si compilarea unei aplicatii. Pentru compilarea miniaplicatiei **FirstApplet**, se foloste compilatorul Java, **javac**. La compilarea unui fisier sursa, compilatorul creeaza un fisier separat pentru fiecare clasa din program.

Deorece **miniaplicatiile pot fi vizualizate cu ajutorul unor programe *hypertext* specializate**, cum ar fi ***browserele* Web**, trebuie creat un document HTML inainte de a putea utiliza miniaplicatia. In cadrul acestui document, pentru incarcarea si rularea miniaplicatiei specificate, se foloseste o eticheta de marcare numita APPLET. In eticheta <APPLET> se face referire la clasele Java, nu la fisierele de clasa care se termina cu extensia **.class**.

Exemplul de document HTML de mai jos contine o eticheta <APPLET> care se refera la clasa **FirstApplet**, si nu la fisierul numit **FirstApplet.class**. Cu ajutorul unui editor sau al unui procesor de texte, se creeaza un fisier de text ASCII standard, cu urmatorul continut (se salveaza acest fisier in acelasi director cu codul compilat al programului **FirstApplet**):

```
<HTML>
<HEAD>
<TITLE>First Java Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE="FirstApplet" width=400 height=400></APPLET>
</BODY>
</HTML>
```

Dupa crearea fisierelor necesare pentru programul **FirstApplet**, se poate rula miniaplicatia cu ajutorul unui **program de vizualizare a *hypertextului***. Setul de dezvoltare Java contine un astfel de program, numit ***appletviewer***. In anumite sisteme, programul ***appletviewer*** este un instrument de lucru din linia de comanda si poate fi apelat cu numele clasei pe care vreti sa o rulati.



4.4. Interfete grafice Java. Biblioteci grafice Java. Java Swing. Java Beans

4.4.1. Elementele unei aplicatii grafice Swing

Programul **`ElementeAplicatieSwing`** ilustreaza **elementele unei aplicatii grafice Swing**.

```
1    // 1. Importul pachetelor Swing si AWT necesare
2
3    import javax.swing.*;          // Numele actual al pachetului Swing
4    //import com.sun.java.swing.*; // Numele pachetului in JDK 1.2 Beta 4 si anterior
5    import java.awt.*;             // Numele pachetului AWT (necesar uneori)
6    import java.awt.event.*;       // Numele pachetului pentru tratarea
7                                   // evenimentelor, pentru interactivitate
8    public class ElementeAplicatieSwing {
9      public static void main(String[] args) {
10
11       // 2. Optional: stabilirea aspectului (Java, Windows, CDE/Motif,...)
12
13       try {
14           UIManager.setLookAndFeel(
15               UIManager.getCrossPlatformLookAndFeelClassName());
16       } catch (Exception e) { }
17
18       // 3. Stabilirea containerului de nivel maxim (JFrame, JApplet, JDialog)
19
20       JFrame frame = new JFrame("Elementele unei aplicatii Swing");
21
22       // 4. Obtinerea panoului de continut intern cadrului
23       //    (containerul in care vor fi plasate componentele ferestrei)
24
25       Container container = frame.getContentPane();
26
27       // 5. Optional: Stabilirea modului de asezare (layout-ului) a panoului
28       //              (implicit FlowLayout)
29
30       container.setLayout(new BorderLayout());
31
32       // 6. Crearea si configurarea componentelor grafice (controalelor)
33
34       // 6.1. Crearea unei etichete
35
36       final String textEticheta = "Numarul de actionari ale butonului: ";
37       final JLabel eticheta = new JLabel(textEticheta + "0    ");
38
39       // 6.2. Crearea unui buton, atasarea unei combinatii de taste echivalente
40
41       JButton buton = new JButton("Sunt un buton Swing!");
42       buton.setMnemonic(KeyEvent.VK_I);
43
44       // 6.3. Crearea si configurarea unui panou (care permite
45       //      gruparea spatiala, ierarhica, a componentelor)
46
47       JPanel panou = new JPanel();
48
49       //      - stabilirea spatiului gol care inconjoara continutul panoului
50       panou.setBorder(BorderFactory.createEmptyBorder(
51                                       30,    // sus
52                                       30,    // in stanga
53                                       10,    // jos
54                                       30));  // in dreapta
55
56       //      - stabilirea modului de asezare (layout-ului) a componentelor
57       panou.setLayout(new GridLayout(0, 1));
58
59       //      - adaugarea componentelor in panou
60       panou.add(buton);
61       panou.add(eticheta);
```

```
62        // 7. Adaugarea in fereastra a componentelor grafice (controalelor)
63
64        container.add(panou, BorderLayout.CENTER);
65
66        // 8. Crearea codului pentru tratarea evenimentelor (interactivitatii)
67
68        // 8.1. Atasarea unui obiect al unei clase anonime care implementeaza
69        //      interfata ActionListener, a carui metoda actionPerformed()
70        //      trateaza actionarea butonului
71        buton.addActionListener(new ActionListener() {
72          int numActionari = 0;
73
74          public void actionPerformed(ActionEvent e) {
75            numActionari++;
76            eticheta.setText(textEticheta + numActionari);
77          }
78        });
79
80        // 8.2. Stabilirea iesirii din program la inchiderea ferestrei
81
82        frame.addWindowListener(new WindowAdapter() {
83            public void windowClosing(WindowEvent e) {
84                System.exit(0);
85            }
86        });
87
88        //    Alternativa, incepand cu Java 2, versiunea 1.3:
89        //      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
90
91        // 9. Stabilirea dimensiunii ferestrei
92        //    - In functie de cea a componentelor:
93        frame.pack();
94
95        //    - Impunerea dimensiunii ferestrei:
96        //      frame.setSize(400,    // latime
97        //                    300);   // inaltime
98
99        // 10. Prezentarea ferestrei pe ecran
00        frame.setVisible(true);
01      }
02    }
```

Desigur, **exista si posibilitatea de a crea coduri mai simple**, de exemplu prin **compactarea etapelor 4 si 7**:

```
frame.getContentPane().add(panou, BorderLayout.CENTER);
```

Urmeaza detalierea fiecarei etape din program.

4.4.1.1. Importul pachetelor Swing si AWT necesare

```
import javax.swing.*;          // Numele actual al pachetului Swing
import java.awt.*;             // Numele pachetului AWT (necesar uneori)
import java.awt.event.*;       // Numele pachetului pentru tratarea
                               // evenimentelor, pentru interactivitate
public class ElementeAplicatieSwing {
  public static void main(String[] args) {
    // codul metodei principale
  }
}
```

4.4.1.2. Stabilirea aspectului

```
try {
    UIManager.setLookAndFeel(
        UIManager.getCrossPlatformLookAndFeelClassName());
} catch (Exception e) { }
```

4.4.1.3. Stabilirea containerului de nivel maxim

```
JFrame frame = new JFrame("Elementele unei aplicatii Swing");
```

4.4.1.4. Obtinerea containerul in care vor fi plasate componentele

```
Container container = frame.getContentPane();
```

4.4.1.5. Stabilirea modului de asezare (*layout-ului*) a panoului

```
container.setLayout(new BorderLayout());
```

4.4.1.6. Crearea si configurarea componentelor grafice

In cazul programului `ElementeAplicatieSwing`:

- crearea unei etichete:

```
final String textEticheta = "Numarul de actionari ale butonului: ";
final JLabel eticheta = new JLabel(textEticheta + "0    ");
```

- crearea unui buton, si atasarea unei combinatii de taste echivalente, [Alt] + I:

```
JButton buton = new JButton("Sunt un buton Swing!");
buton.setMnemonic(KeyEvent.VK_I);
```

- crearea si configurarea unui panou (componenta care grupeaza spatial alte componente):

```
JPanel panou = new JPanel();

// Stabilirea spatiului gol care inconjoara continutul panoului
panou.setBorder(BorderFactory.createEmptyBorder(
                                30,    // sus
                                30,    // in stanga
                                10,    // jos
                                30));  // in dreapta

// Stabilirea modului de asezare (layout-ului) a componentelor
panou.setLayout(new GridLayout(0, 1));

// Adaugarea componentelor in panou
panou.add(buton);
panou.add(eticheta);
```

4.4.1.7. Adaugarea in fereastra principala a componentelor grafice

```
container.add(panou, BorderLayout.CENTER);
```

4.4.1.8. Crearea codului pentru tratarea evenimentelor (interactivitatii)

Atasarea unui obiect al unei clase anonime care implementeaza interfata `ActionListener`, a carui metoda **`actionPerformed()` trateaza actionarea butonului:**

```
buton.addActionListener(new ActionListener() {
  int numActionari = 0;

  public void actionPerformed(ActionEvent e) {
    numActionari++;
    eticheta.setText(textEticheta + numActionari);
  }
});
```

Stabilirea iesirii din program la inchiderea ferestrei, prin **atasarea unui obiect al unei clase anonime care extinde clasa `ActionListener`**, a carui metoda **`windowClosing()` trateaza inchiderea ferestrei:**

```
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
```

Alternativa pentru stabilirea iesirii din program la inchiderea ferestrei, incepand cu versiunea 1.3 a Java 2 SE:

```
// Alternativa, incepand cu Java 2, versiunea 1.3:
//      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

4.4.1.9. Stabilirea dimensiunii ferestrei

Stabilirea dimensiunii ferestrei in functie de cea a componentelor:

```
frame.pack();
```

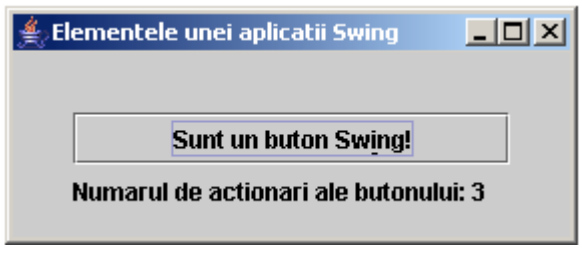
Impunerea dimensiunii ferestrei:

```
// Alternativa pentru cazul ca se doreste impunerea unei dimensiuni
//      frame.setSize(400,    // latime
//                    300);   // inaltime
```

4.4.1.10. Prezentarea ferestrei pe ecran

```
frame.setVisible(true);
```

Fereastra obtinuta prin executia programului **`ElementeAplicatieSwing.java`**:



4.4.2. Modalitati de a crea containerul de nivel maxim

Exista urmatoarele modalitati de a crea containere de nivel maxim:

1. Includerea unui obiect de tip `Frame` (fereastra principala in programele Java de sine statatoare)

Programul **`IncludereJFrame`** ilustreaza crearea unei ferestre principale prin **includerea unui obiect** de tip **`JFrame`**, fereastra in care **sunt asezate 5 butoane, folosind asezarea relativ la margini** - **`BorderLayout`**.

```
 1  import java.awt.*;
 2  import javax.swing.*;
 3  /**
 4   * Demonstreaza includerea unui obiect JFrame pentru a crea o fereastra pe ecran.
 5   */
 6  public class IncludereJFrame {
 7      public static void main(String[] args) {
 8        // Crearea obiectului cadru, cu titlu specificat
 9        JFrame cadru = new JFrame("Demo includere JFrame si asezare BorderLayout");
10
11        // Obtinerea panoului de continut intern cadrului (container de componente)
12        Container container = cadru.getContentPane();
13
14        // Asezarea componentelor in panou (la 10 pixeli de marginea panoului)
15        container.setLayout(new BorderLayout(10, 10));
16
17        // Adaugarea a 5 butoane la panoul cadrului (ferestrei)
18        container.add(new JButton("Est (Dreapta)"), BorderLayout.EAST);
19        container.add(new JButton("Sud (Jos)"), BorderLayout.SOUTH);
20        container.add(new JButton("Vest (Stanga)"), BorderLayout.WEST);
21        container.add(new JButton("Nord (Sus)"), BorderLayout.NORTH);
22        container.add(new JButton("Centru"), BorderLayout.CENTER);
23
24        // Din jdk1.3, pentru terminarea programului la inchiderea ferestrei
25        // cadru.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26
27        // Stabilirea dimensiunii ferestrei
28        // - impunand dimensiunile ferestrei (nepotrivita la BorderLayout):
29        //      cadru.setSize(100, 100);
30        // - compactand componentele adaugate:
31        cadru.pack();
32
33        // Stabilirea vizibilitatii ferestrei (Atentie: implicit e false!)
34        cadru.setVisible(true);
35    }
36  }
```

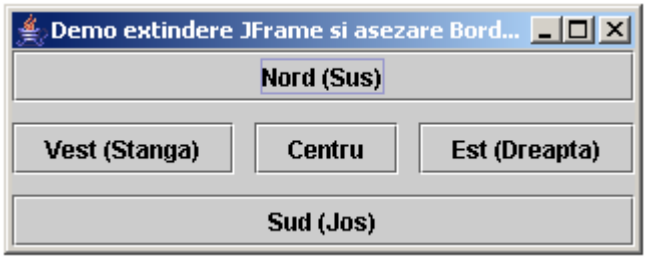
Fereastra obtinuta prin executia programului **`IncludereJFrame.java`**:

2. Extinderea clasei Frame (fereastra principala in programele Java de sine statatoare)

Programul **`ExtensieJFrame`** ilustreaza crearea unei ferestre principale prin **extinderea clasei `JFrame`**, si **asezarea relativ la margini** - **`BorderLayout`**.

```
 1  import java.awt.*;
 2  import javax.swing.*;
 3  /**
 4   * Demonstreaza extinderea JFrame pentru a crea o fereastra pe ecran.
 5   */
 6  public class ExtensieJFrame extends JFrame {
 7      public ExtensieJFrame() {
 8        // Obtinerea panoului de continut intern cadrului (container de componente)
 9        Container container = getContentPane();
10
11        // Asezarea componentelor in panou (la 10 pixeli de marginea panoului)
12        container.setLayout(new BorderLayout(10, 10));
13        // Adaugarea a 5 butoane la panoul cadrului (ferestrei)
14        container.add(new JButton("Est (Dreapta)"), BorderLayout.EAST);
15        container.add(new JButton("Sud (Jos)"), BorderLayout.SOUTH);
16        container.add(new JButton("Vest (Stanga)"), BorderLayout.WEST);
17        container.add(new JButton("Nord (Sus)"), BorderLayout.NORTH);
18        container.add(new JButton("Centru"), BorderLayout.CENTER);
19      }
20
21      public static void main(String[] args) {
22        // Crearea obiectului cadru
23        ExtensieJFrame cadru = new ExtensieJFrame();
24
25        // Adaugarea titlului ferestrei
26        cadru.setTitle("Demo extindere JFrame si asezare BorderLayout");
27
28        // Compactarea componentelor
29        cadru.pack();
30        // Stabilirea vizibilitatii ferestrei (Atentie: implicit e false!)
31        cadru.setVisible(true);
32      }
33  }
```

Fereastra obtinuta prin executia programului **`ExtensieJFrame.java`**:



3. Extinderea clasei JApplet (in cazul miniaplicatiilor Java)

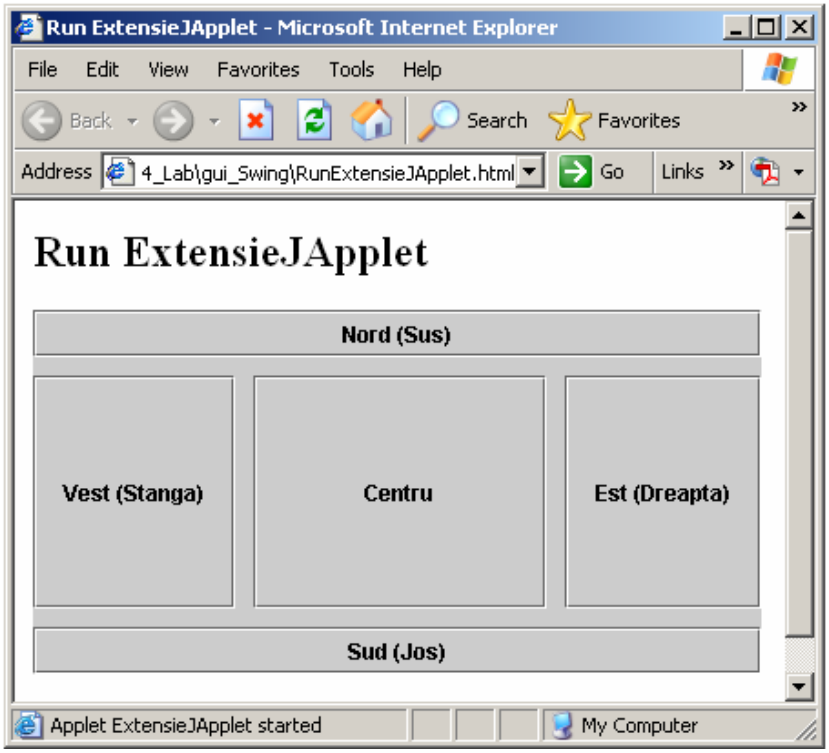
Pentru includerea unei miniaplicatii intr-o pagina HTML in vederea vizualizarii ei se va adauga in corpul fisierului .html urmatorul cod:

```
<APPLET  CODE = "ExtensieJApplet.class" WIDTH = 400 HEIGHT = 200 >
</APPLET>
```

Programul **`ExtensieJApplet`** ilustreaza **crearea unei miniaplicatii (*applet*)** prin **extinderea clasei `JApplet`**, si **asezarea relativ la margini** - **`BorderLayout`**.

```
1  import java.awt.*;
2  import javax.swing.*;
3  /**
4   * Demonstreaza extinderea JApplet pentru a o miniaplicatie Java.
5   *
6   * Se executa prin includerea intr-o pagina Web a unui tag HTML de genul:
7   *    <APPLET  CODE = "ExtensieJApplet.class" WIDTH = 400 HEIGHT = 200 >
8   *    </APPLET>
9   */
10 public class ExtensieJApplet extends JApplet {
11     /**
12      * Metoda de initializare a appletului. Apelata de browser la prima
13      * utilizare a appletului, stabileste layout-ul (modul de dispunere a
14      * componentelor in panoul de continut) si adauga componentele in panou.
15      */
16     public void init() {
17       // Obtinerea panoului de continut intern cadrului (container de componente)
18       Container container = getContentPane();
19
20       // Asezarea componentelor in panou (la 10 pixeli de marginea panoului)
21       container.setLayout(new BorderLayout(10, 10));
22
23       // Adaugarea a 5 butoane la panoul appletului
24       container.add(new JButton("Est (Dreapta)"), BorderLayout.EAST);
25       container.add(new JButton("Sud (Jos)"), BorderLayout.SOUTH);
26       container.add(new JButton("Vest (Stanga)"), BorderLayout.WEST);
27       container.add(new JButton("Nord (Sus)"), BorderLayout.NORTH);
28       container.add(new JButton("Centru"), BorderLayout.CENTER);
29     }
30 }
```

Pagina HTML **RunExtensieJApplet.html** permite vizualizarea appletului **`ExtensieJApplet`**. Iata cum poate arata pagina **RunExtensieJApplet.html** vizualizata intr-un browser:



4. Crearea unui obiect de tip `JDialog`, pentru a crea o fereastra secundara

5. Crearea ferestrelor de dialog predefinite utilizand metodele clasei `JOptionPane`

Metoda **`showInputDialog()`** a clasei **`JOptionPane`** este folosita pentru a crea **dialoguri de intrare**, metoda **`showMessageDialog()`** pentru a crea **dialoguri de informare a utilizatorului**, etc.

4.4.3. Crearea interactivitatii aplicatiilor si miniaplicatiilor grafice Swing

Programele din sectiunea anterioara creau butoane care nu reactioneaza la actionarea lor de catre utilizator.

Pentru introducerea interactivitatii, trebuie tratate evenimentele din interfata grafica. In Java exista mai multe moduri de tratare a evenimentelor.

Incepand cu versiunea initiala, JDK 1.0, interfetele grafice realizate cu biblioteca AWT au 2 moduri de tratare a evenimentelor:

1. Implementand metoda `action()`, care:

- primeste ca parametri un obiect de tip **`Event`** care **incapsuleaza evenimentul produs,** si **un obiect** de tip **`Object`** care **incapsuleaza parametrii** acestuia,

- testeaza atributele `target` si `id` ale obiectului de tip **`Event`** pentru a **identifica obiectul tinta (in care s-a produs evenimentul) si tipul de actiune** produsa, si **trateaza apoi evenimentul respectiv**

2. Implementand metoda `handleEvent()`, care:

- primeste ca parametru un obiect de tip **`Event`** care **incapsuleaza evenimentul produs,**

- testeaza atributele `target` si `id` ale obiectului de tip **`Event`** pentru a **identifica obiectul tinta (in care s-a produs evenimentul) si tipul de actiune** produsa, si **trateaza apoi evenimentul respectiv**

Incepand cu versiunea JDK 1.1, interfetele grafice realizate cu biblioteca AWT au:

3. Un nou mod de tratare a evenimentelor, utilizat si de interfetele grafice Swing, care presupune trei operatii:

- (a) declararea unei clase care implementeaza o interfata « ascultator de evenimente », (care contine metode ce trebuie implementate de utilizator pentru tratarea evenimentului respectiv), sau **(b) declararea unei clase care extinde o clasa predefinita care implementeaza o interfata « ascultator de evenimente »**

- (a) implementarea tuturor metodelor definite in interfata « ascultator de evenimente », sau **(b) re-implementarea metodelor dorite din clasa extinsa care implementeaza interfata**

- inregistrarea unui obiect din clasa « ascultator de evenimente » de catre fiecare dintre componentele grafice (numite tinta sau sursa) pentru care se doreste tratarea evenimentului respectiv

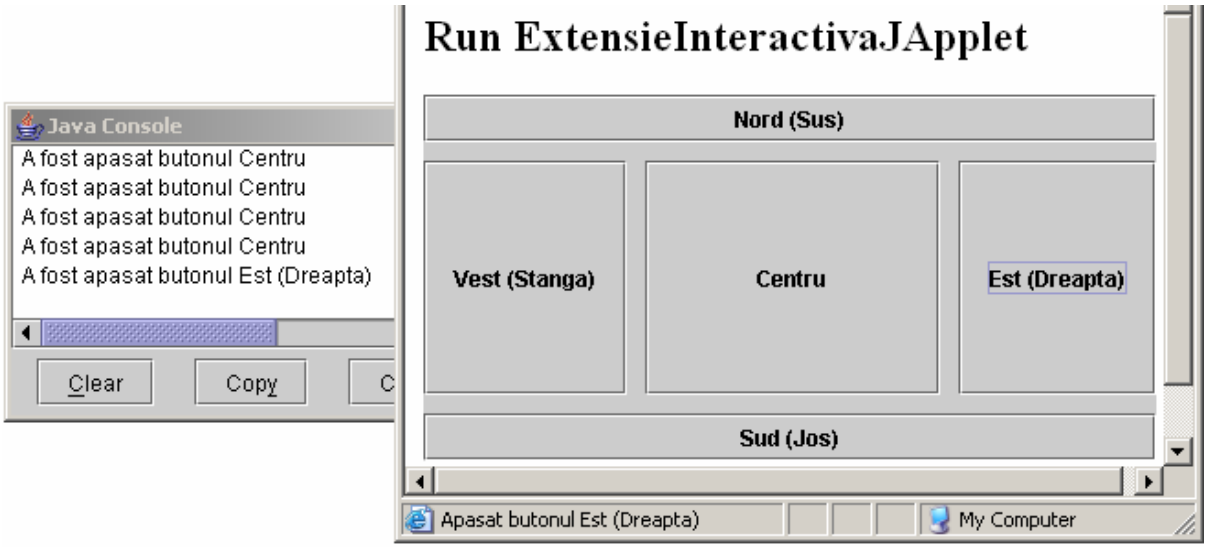
Programul **`ExtensieInteractivaJFrame`** ilustreaza crearea unei ferestre principale prin **extinderea clasei `JFrame`**, **asezarea relativ la margini - `BorderLayout`**, si **tratarea evenimentului « actionare » pentru componentele buton**.

```
 1  import java.awt.*;
 2  import java.awt.event.*;
 3  import javax.swing.*;
 4  /**
 5   * Extinderea JFrame pentru a crea o fereastra cu componente interactive pe ecran.
 6   */
 7  public class ExtensieInteractivaJFrame extends JFrame {
 8      public ExtensieInteractivaJFrame() {
 9        // Obtinerea panoului de continut intern cadrului (container de componente)
10        Container container = getContentPane();
11
12        // Asezarea componentelor in panou (la 10 pixeli de marginea panoului)
13        container.setLayout(new BorderLayout(10, 10));
14
15        // Adaugarea a 5 butoane la panoul cadrului (ferestrei)
16        JButton b1 = new JButton("Est (Dreapta)");
17        JButton b2 = new JButton("Sud (Jos)");
18        JButton b3 = new JButton("Vest (Stanga)");
19        JButton b4 = new JButton("Nord (Sus)");
20        JButton b5 = new JButton("Centru");
21        container.add(b1, BorderLayout.EAST);
22        container.add(b2, BorderLayout.SOUTH);
23        container.add(b3, BorderLayout.WEST);
24        container.add(b4, BorderLayout.NORTH);
25        container.add(b5, BorderLayout.CENTER);
26
27        // Inregistrarea "ascultatorului" de "evenim.actionare" la "sursele" evenim.
28        b1.addActionListener(obiectAscultatorActionare);
29        b2.addActionListener(obiectAscultatorActionare);
30        b3.addActionListener(obiectAscultatorActionare);
31        b4.addActionListener(obiectAscultatorActionare);
32        b5.addActionListener(obiectAscultatorActionare);
33
34        // Inregistrarea "ascultatorului" de "evenimente fereastra "
35        this.addWindowListener(ascultatorInchidereFereastra);
36      }
37
38      // Crearea unui obiect "ascultator" de "evenimente actionare"
39      ActionListener obiectAscultatorActionare = new ActionListener() {
40
41        // Tratarea actionarii unui buton
42        public void actionPerformed(ActionEvent ev) {
43          // Mesaj informare
44          System.out.println("A fost apasat un buton " + ev.getActionCommand());
45        }
46      };
47
48      // Crearea unui "adaptor pentru ascultator" de "evenimente fereastra"
49      WindowAdapter ascultatorInchidereFereastra = new WindowAdapter() {
50        // Tratarea inchiderii ferestrei curente
51        public void windowClosing(WindowEvent ev) {
52          // Terminarea programului
53          System.exit(0);
54        }
55      };
56
57      public static void main(String[] args) {
58        // Crearea obiectului cadru
59        ExtensieInteractivaJFrame cadru = new ExtensieInteractivaJFrame();
60
61        // Adaugarea titlului ferestrei
62        cadru.setTitle("Demo extindere JFrame interactiva");
63        // Compactarea componentelor
64        cadru.pack();
65        // Stabilirea vizibilitatii ferestrei (Atentie: implicit e false!)
66        cadru.setVisible(true);
67      }
68  }
```

Programul **`ExtensieInteractivaJApplet`** ilustreaza **crearea unei miniaplicatii (*applet*)** prin **extinderea clasei `JApplet`**, si **tratarea evenimentului « actionare » pentru componentele buton**.

```
 1  import java.awt.*;
 2  import java.awt.event.*;
 3  import javax.swing.*;
 4  public class ExtensieInteractivaJApplet extends JApplet {
 5
 6      public void init() {
 7        // Obtinerea panoului de continut (content pane) creat de browser pentru
 8        // executia appletului (container in care vor fi plasate componentele)
 9        Container container = getContentPane();
10
11        // Stabilirea layout-ului panoului, BorderLayout cu spatiu 10 pixeli
12        container.setLayout(new BorderLayout(10, 10));
13        // Adaugarea a 5 butoane la panoul appletului
14        JButton b1 = new JButton("Est (Dreapta)");
15        JButton b2 = new JButton("Sud (Jos)");
16        JButton b3 = new JButton("Vest (Stanga)");
17        JButton b4 = new JButton("Nord (Sus)");
18        JButton b5 = new JButton("Centru");
19        container.add(b1, BorderLayout.EAST);
20        container.add(b2, BorderLayout.SOUTH);
21        container.add(b3, BorderLayout.WEST);
22        container.add(b4, BorderLayout.NORTH);
23        container.add(b5, BorderLayout.CENTER);
24
25        // Crearea unui obiect "ascultator" de "evenimente actionare"
26        // (pe care le trateaza)
27        ActionListener obiectAscultatorActionare = new ActionListener() {
28
29          // Tratarea actionarii unui buton
30          public void actionPerformed(ActionEvent ev) {
31
32            // Mesaj informare in consola Java
33            System.out.println("A fost apasat butonul " + ev.getActionCommand());
34
35            // Mesaj informare in bara de stare
36            showStatus("Apasat butonul " + ev.getActionCommand());
37          }
38        };
39
40        // Inregistrarea "ascultatorului" de "evenimente actionare" la "sursele"
41        // de evenimente
42        b1.addActionListener(obiectAscultatorActionare);
43        b2.addActionListener(obiectAscultatorActionare);
44        b3.addActionListener(obiectAscultatorActionare);
45        b4.addActionListener(obiectAscultatorActionare);
46        b5.addActionListener(obiectAscultatorActionare);
47      }
48  }
```

Pagina **RunExtensieInteractivaJApplet.html** permite vizualizarea *applet-ului*. Iata cum poate arata pagina **RunExtensieJApplet.html** vizualizata intr-un browser:



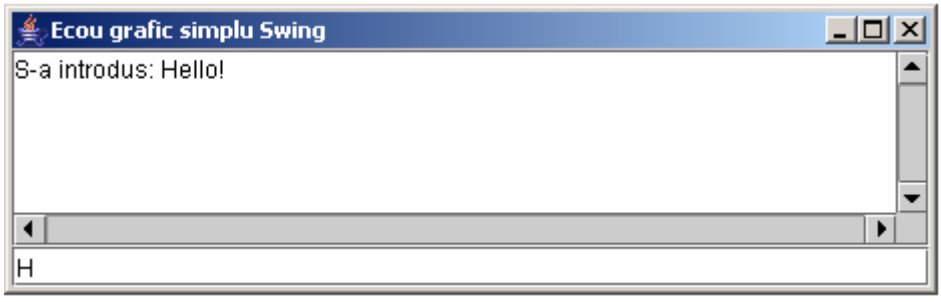
4.4.4. Utilizarea componentelor grafice Swing pentru lucrul cu text

Programul **EcouGrafic_Swing** ilustreaza utilizarea componentelor grafice Swing pentru lucrul cu text, de tip **JTextField** si **JTextArea**.

```
 1  import java.awt.*;
 2  import java.awt.event.*;
 3  import javax.swing.*;
 4
 5  /**
 6    * Ecou grafic folosind JtextField si JTextArea
 7    */
 8  public class EcouGrafic_Swing extends JFrame {
 9
10    /**
11      * Intrare - linie de text grafica (JtextField)
12      */
13    private static JTextField inTextGrafic;
14
15    /**
16      * Iesire - zona de text grafica (JtextArea)
17      */
18    private static JTextArea outTextGrafic;
19
20    /**
21      * Initializari grafice
22      */
23    public EcouGrafic_Swing() {
24
25      // Stabilire titlu fereastra (JFrame)
26      super("Ecou grafic simplu Swing");
27      Container containerCurent = this.getContentPane();
28      containerCurent.setLayout(new BorderLayout());
29
30      // Zona de text non-editabila de iesire (cu posibilitati de defilare)
31
32      outTextGrafic = new JTextArea(5, 40);
33      JScrollPane scrollPane = new JScrollPane(outTextGrafic,
34                      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
35                      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
36      containerCurent.add("Center", scrollPane);
37      outTextGrafic.setEditable(false);
38
39      // Camp de text editabil de intrare
40
41      inTextGrafic = new JTextField(40);
42      containerCurent.add("South", inTextGrafic);
43
44      System.out.println("\nPentru oprire introduceti '.' si <Enter>\n");
45
46      // Inregistrarea "ascultatorului" de "evenimente actionare" la
47      // "obiectul sursa" intrare de text
48      inTextGrafic.addActionListener(ascultatorInText);
49
50      // Inregistrarea "ascultatorului" de "evenimente fereastra" la
51      // "sursa" (fereastra curenta)
52      this.addWindowListener(ascultatorInchidere);
53
54      // Impachetarea (compactarea) componentelor in container
55      pack();
56
57      // Fereastra devine vizibila - echivalent cu frame.setVisible(true)
58      show();
59
60      // Cerere focus pe intrarea de text din fereastra curenta
61      inTextGrafic.requestFocus();
62    }
63
```

```
64    /**
65      * Crearea unui "ascultator" de "evenimente actionare", obiect al unei
66      * clase "anonime" care implementeaza interfata ActionListener
67      */
68    ActionListener ascultatorInText = new ActionListener() {
69      /**
70        * Tratarea actionarii intrarii de text (introducerii unui "Enter")
71        */
72      public void actionPerformed(ActionEvent ev) {
73        // Citirea unei linii de text din intrarea de text grafica
74        String sirCitit = inTextGrafic.getText();
75        // Pregatirea intrarii de text pentru noua intrare (golirea ei)
76        inTextGrafic.setText("");
77
78        // Scrierea liniei de text in zona de text grafica
79        outTextGrafic.append("S-a introdus: " + sirCitit + "\n");
80        // Conditie terminare program
81        if (sirCitit.equals(new String(".")))   System.exit(0);
82      }
83    };
84
85    /**
86      * Crearea unui "adaptor pentru ascultator" de "evenimente fereastra"
87      * Obiect al unei clase "anonime" care extinde clasa WindowAdapter
88      */
89    WindowAdapter ascultatorInchidere = new WindowAdapter() {
90
91      /**
92        * Tratarea inchiderii ferestrei curente
93        */
94      public void windowClosing(WindowEvent ev) {
95        // Terminarea programului
96        System.exit(0);
97      }
98    };
99
00    /**
01      * Punctul de intrare in program
02      */
03    public static void main (String args[]) {
04      EcouGrafic_Swing ecouGraficJTFJTA = new EcouGrafic_Swing();
05    }
06  }
```

Fereastra obtinuta prin executia programului **`EcouGrafic_Swing.java`**:



Programul **EcouGrafic_EventHandler** ilustreaza utilizarea metodei **eventHandler()** pentru tratarea evenimentelor grafice.

```
 1  import java.io.*;
 2  import java.awt.*;
 3
 4  public class EcouGrafic_EventHandler extends Frame {
 5    protected TextArea outTextGrafic;
 6    protected TextField inTextGrafic;
 7
 8    public EcouGrafic_EventHandler(String title) {
 9      super(title);
10
11      setLayout(new BorderLayout());
12
13      outTextGrafic = new TextArea(5, 40);
14      add("Center", outTextGrafic);
15      outTextGrafic.setEditable (false);
16
17
18      inTextGrafic = new TextField(40);
19      add("South", inTextGrafic);
20
21      pack();
22      show();
23
24      inTextGrafic.requestFocus();
25    }
26
27    // rutina de tratare a evenimentelor din interfata grafica AWT/jdk1.0
28    // (fir de executie separat)
29
30    public boolean handleEvent(Event ev) {
31
32      // tratarea unui eveniment tip ACTION_EVENT (introducerea unui "Enter")
33      // in intrarea de text tinta (inTextGrafic)
34      if ((ev.target == inTextGrafic) && (ev.id == Event.ACTION_EVENT)) {
35
36        // citirea intrarii de text si adaugarea in iesirea de text (outTextGrafic)
37
38        String intrare = inTextGrafic.getText();
39
40        outTextGrafic.appendText("S-a introdus: " + intrare + "\n");
41
42        inTextGrafic.setText("");
43        return true;
44
45      // tratarea unui eveniment tip WINDOW_DESTROY (inchiderea ferestrei)
46      // in fereastra curenta
47      } else if ((ev.target == this) && (ev.id == Event.WINDOW_DESTROY)) {
48
49        // terminare (prin inchidere fereastra)
50        System.exit(0);
51        return true;
52      }
53      return super.handleEvent(ev);
54    }
55
56    public static void main (String args[]) throws IOException {
57      EcouGrafic_EventHandler ecouGraficLocal =
58          new EcouGrafic_EventHandler ("Ecou local grafic simplu (EventHandler)");
59    }
60  }
```

Programul **`EcouGrafic_Action`** ilustreaza utilizarea metodei **`action()`** pentru tratarea evenimentelor grafice.

```
 1  import java.io.*;
 2  import java.awt.*;
 3
 4  public class EcouGrafic_Action extends Frame {
 5    protected TextArea outTextGrafic;
 6    protected TextField inTextGrafic;
 7
 8    public EcouGrafic_Action(String title) {
 9      super(title);
10
11      setLayout(new BorderLayout());
12
13      outTextGrafic = new TextArea(5, 40);
14      add("Center", outTextGrafic);
15      outTextGrafic.setEditable (false);
16
17
18      inTextGrafic = new TextField(40);
19      add("South", inTextGrafic);
20
21      pack();
22      show();
23
24      inTextGrafic.requestFocus();
25    }
26
27    // rutina de tratare a actiunilor (perechi {eveniment, obiect}
28    // din interfata grafica AWT/jdk1.0 (fir de executie separat)
29    public boolean action(Event ev, Object arg) {
30
31      // tratarea unui eveniment tip ACTION_EVENT (introducerea unui "Enter")
32      // in intrarea de text tinta (inTextGrafic)
33      if ((ev.target == inTextGrafic) && (ev.id == Event.ACTION_EVENT)) {
34
35        // citire din intrare de text
36
37        String intrare = inTextGrafic.getText();
38        outTextGrafic.appendText("S-a introdus: " + intrare + "\n");
39
40        // scriere in iesire de text
41
42        inTextGrafic.setText("");
43        return true;
44
45      // tratarea unui eveniment tip WINDOW_DESTROY (inchiderea ferestrei)
46      // in fereastra curenta
47      } else if ((ev.target == this) && (ev.id == Event.WINDOW_DESTROY)) {
48
49        // terminare (prin inchidere fereastra)
50        System.exit(0);
51        return true;
52      }
53      return super.action(ev, arg);
54    }
55
56    public static void main (String args[]) throws IOException {
57      EcouGrafic_Action ecouGraficLocal =
58          new EcouGrafic_Action ("Ecou local grafic simplu (Action)");
59    }
60  }
```

Programul **PasswordDemo** ilustreaza utilizarea unei intrari text pentru parole (*passwords*).

```
1    import javax.swing.*;
2    import java.awt.*;
3    import java.awt.event.*;
4
5    public class PasswordDemo {
6      private static boolean isPasswordCorrect(char[] input) {
7        char[] correctPassword = { 'g', 'h', 'i', 'c', 'i' };
8        if (input.length != correctPassword.length)
9          return false;
10       for (int i = 0;  i < input.length; i ++)
11         if (input[i] != correctPassword[i])
12           return false;
13       return true;
14     }
15
16     public static void main(String[] argv) {
17       final JFrame f = new JFrame("PasswordDemo");
18       JLabel label = new JLabel("Introduceti parola: ");
19
20       JPasswordField passwordField = new JPasswordField(10);
21
22       passwordField.setEchoChar('#');
23
24       // Tratarea actionarii intrarii
25       passwordField.addActionListener(new ActionListener() {
26         public void actionPerformed(ActionEvent e) {
27
28           JPasswordField input = (JPasswordField)e.getSource();
29
30           char[] password = input.getPassword();
31
32           if (isPasswordCorrect(password)) {
33             JOptionPane.showMessageDialog(f, "Succes! Password corect.");
34           }
35           else {
36             JOptionPane.showMessageDialog(f, "Password incorect. Mai incearca.",
37                         "Error Message", JOptionPane.ERROR_MESSAGE);
38           }
39         }
40       });
41
42       JPanel contentPane = new JPanel(new BorderLayout());
43       contentPane.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
44       contentPane.add(label, BorderLayout.WEST);
45       contentPane.add(passwordField, BorderLayout.CENTER);
46
47       f.setContentPane(contentPane);
48       f.addWindowListener(new WindowAdapter() {
49         public void windowClosing(WindowEvent e) { System.exit(0); }
50       });
51       f.pack();
52       f.setVisible(true);
53     }
54   }
```

Fereastrele obtinute prin executia programului **PasswordDemo.java**: