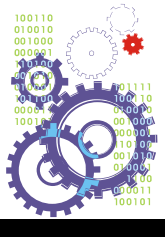
1

Introducere în Java

Ce este Java?

- **Proiectat de Sun pentru programarea aparaturii electronice**
- **Limbaj orientat pe obiect cu biblioteci de clase**
- **Folosește o mașină virtuală pentru rularea programelor**



Proiectat de Sun

Java este un limbaj de programare orientat pe obiect dezvoltat la Sun Microsystems, Inc. A fost creat de James Gosling pentru a fi utilizat la programarea aparaturii electronice. Ca urmare a robusteţii şi a independenţei de platformă, Java a trecut de bariera industriei "aparaturii electronice" la aplicaţiile de Web, apoi la aplicaţii locale şi client-server

Biblioteci de clase

Java conţine o mulţime de clase predefinite şi metode care pot trata majoritatea cerinţelor fundamentale ale unei aplicaţii. Kitul Dezvoltatorului Java (JDK - Java Developer's Kit) include clase pentru gestionarea de ferestre, intrare/ieşire și comunicaţie în reţea.

Java mai conţine un număr de utilitare care ajută la dezvoltarea aplicaţiilor. Aceste utilitare tratează operaţii cum sînt: depanarea, descărcarea şi instalarea, documentarea.

Folosește Mașina Virtuală Java

Unul dintre punctele forte ale lui Java este independenţa de platformă (maşină + sistem de operare). O aplicaţie Java scrisă pe o platformă poate fi dusă și rulată pe orice altă platformă. Facilitatea este deseori referită ca "write once, run anywhere (WORA)". Ea este dată de folosirea Mașinii Virtuale Java - Java Virtual Machine (JVM). Aceasta rulează pe o maşină locală şi interpretează codul de biţi (byte code) convertindu-l într-un cod maşină specific platformei.

Echipa lui Gosling a pornit de la C++, dar programele nu erau sigure, parte a complexităţii limbajului, dar și datorită unor facilităţi distructive cum sunt poantorii. Pentru evitarea problemelor echipa lui Gosling a inventat un limbaj nou orientat pe obiect numit Oak (în memoria unui stejar impunător vizibil din biroul lui Gosling). Pentru creșterea robusteţii au eliminat construcţiile de limbaj problematice, iar pentru a face aplicaţiile neutre arhitectural, au specificat complet semantica limbajului și au creat o mașină virtuală pentru rularea programelor. Deși Oak a fost proiectat să semene cu C++ pentru a reduce timpul de învăţare, el a fost reproiectat intern pentru a elimina pericolele și elementele redundante ale lui C++. Oak nu a reușit să se impună în domeniul pentru care a fost creat, dar, după apariţia WWW-ului și-a găsit un nou drum pe care să evolueze. Oak a fost redenumit în Java (exista deja un limbaj cu numele Oak). Robust, compact, independent de platformă și flexibil a devenit ideal pentru crearea de aplicaţii Web. Fiind interpretat este mai lent ca C++, dar scăderea vitezei este neimportantă atunci cînd domeniul de utilizare este cel de interacţiune cu utilizatorul.

Avantaje Java

- **Orientat pe obiect**
- **Interpretat și independent de platformă**
- **Dinamic și distribuit**
- **Cu fire de execuție multiple**
- **Robust și sigur**

Orientat pe obiect
Obiectul este o entitate caracterizată prin atribute şi un set de funcţii folosite la manipularea obiectului. Java este puternic tipizat, aproape totul fiind, la nivel de limbaj, un obiect. Principalele excepţii sînt tipurile primitive (întregii şi caracterele).

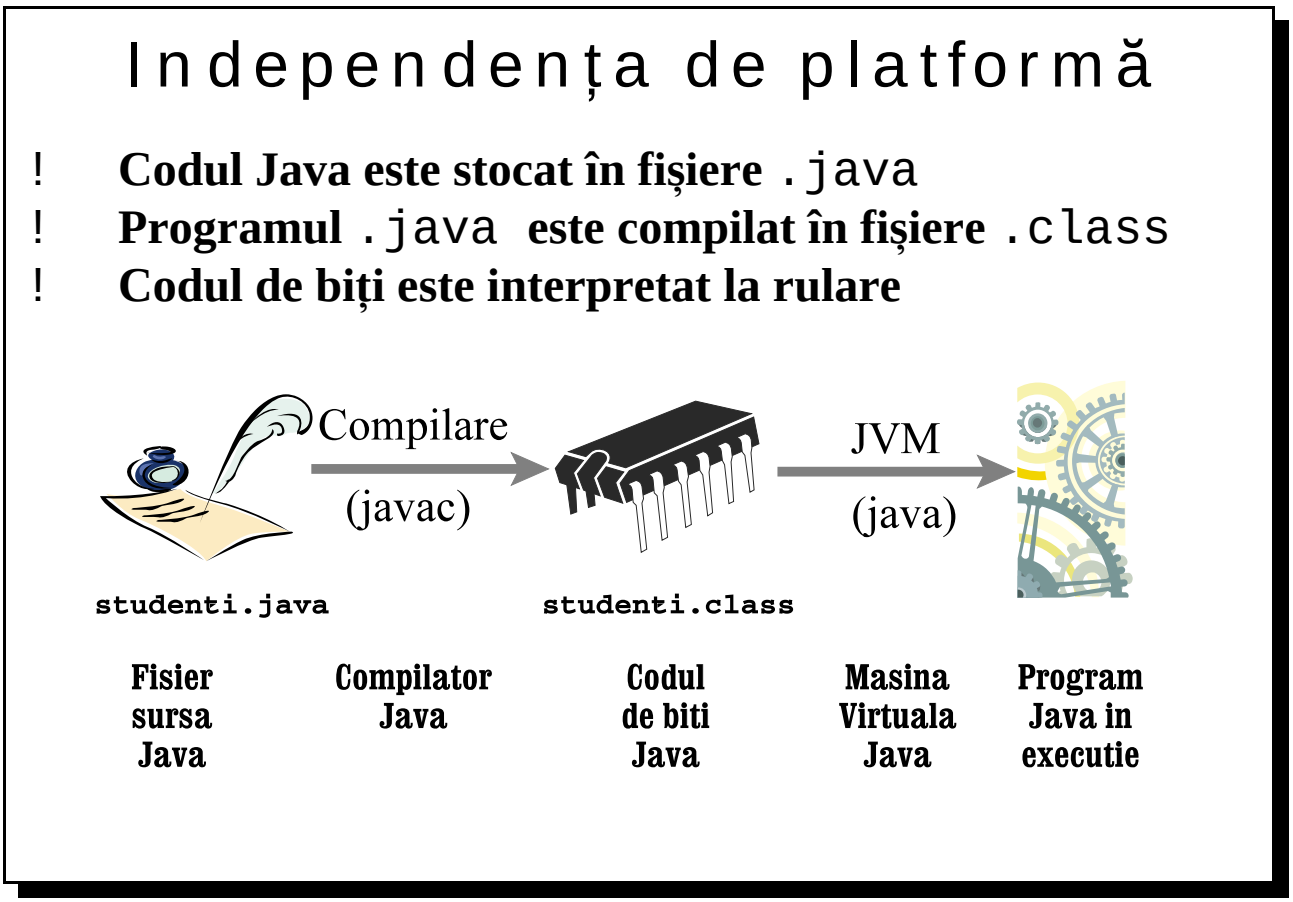
Interpretat şi independent de platformă
Programele Java sînt interpretate în setul de instrucţiuni ale maşinii native în timpul rulării. Deoarece Java se rulează sub controlul JVM, programele Java pot rula sub orice sistem de operare care dispune de JVM.

Dinamic şi distribuit
Clasele Java pot fi descărcate dinamic prin reţea atunci cînd este cazul. În plus, Java asigură un suport extins pentru programarea client-server şi distribuită.

Cu fire multiple de execuţie (Multithreaded)
Programele Java pot conţine mai multe fire în vederea execuţie concurente a mai multor sarcini. Multithreadingul este o facilitate internă lui Java şi a aflată sub controlul JVM care este dependent de platformă.

Robust şi sigur
Java are facilităţi interne de prevenire a coruperii memoriei. Java gestionează automat procesul de alocare a memoriei şi verificarea limitelor tablourilor. Interzice aritmetica poantorilor şi restricţionează obiectele la zone impuse de memorie.

Termenul de multithreading (fire de execuţie multiple) este specific calculului paralel şi apare în contextul medii de execuţie care pot intercala execuţia unor instrucţiuni din mai multe surse independente de execuţie. Diferenţa faţă de multitasking constă într-o partajare mai profundă a mediului de execuţie la nivel de thread-uri în comparaţie cu multitasking-ul. Thread-urile pot fi identificate numai prin registrul numărător de prorame (PC) și registrul poantor la stivă (SP) atunci cînd partajează un singur spațiu de adrese și o singură mulțime de variabile globale. Acesta este motivul pentru care comutarea între-thread-uri se face foarte repede deoarece informația de stare de salvat și de refăcut este scurtă.



Problema limbajelor compilate
La compliarea unei aplicaţii într-un limbaj tradiţional, cum ar fi C, codul scris de programator este convertit în limbajul maşină al platformei pe care are loc compilarea. Programul compilat poate fi rulat numai pe maşini care au acelaşi procesor (Intel, SPARC, Alpha) cu cea pe care s-a făcut compilarea.

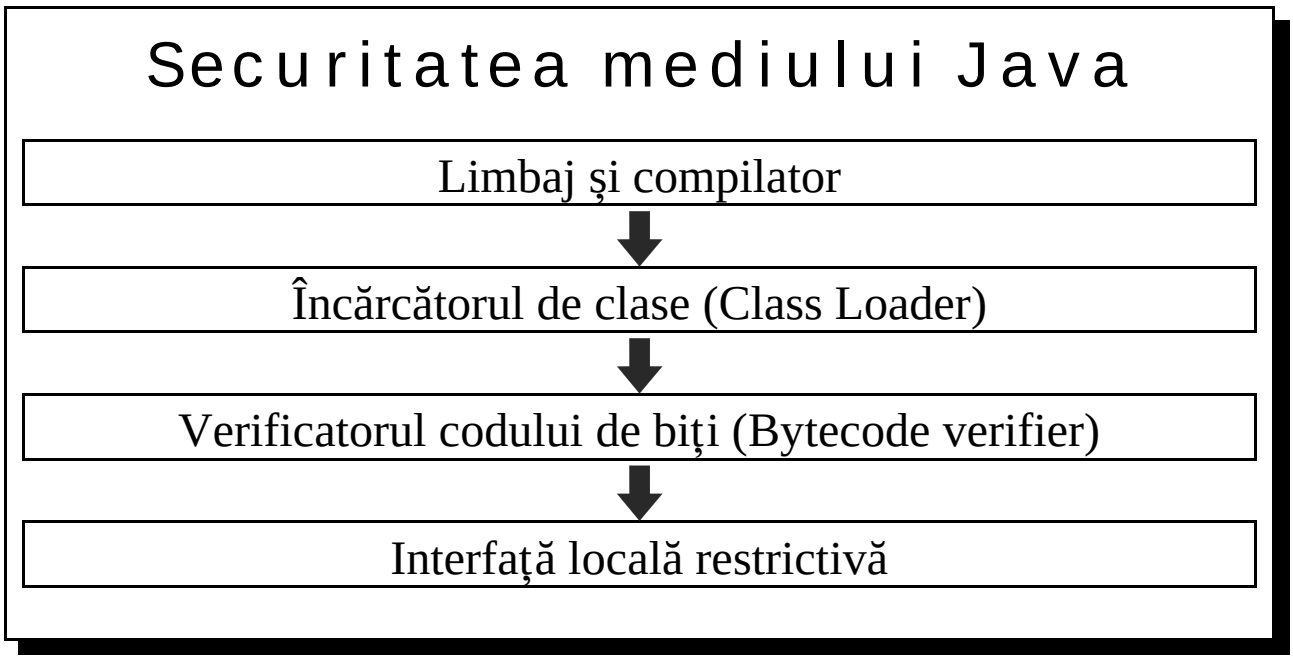
Java este un limbaj independent de platformă
Programele sursă Java sînt stocate în fişiere cu extensia `.java`. De exemplu, o aplicaţie Java care gestionează studenţii unei secţii ar putea conţine fişerele `studenti.java`, `materii.java` şi `note.java`. Fiecare fişier `.java` este compilat într-un fişier `.class` cu acelaşi nume. De exemplu, `studenti.java` se va compila în `studenti.class`. Fişierele `.class` conţin cod executabil Java numit "cod de biţi" (bytecode) care are instrucţiuni maşină independente de platformă.

Maşina Virtuală Java (JVM - Java Virtual Machine)
Maşina Virtuală Java asigură mediul de execuţie pentru programele Java. JVM interpretează codul de biţi în instrucţiuni maşină native ale maşinii pe care programul este în curs de rulare.

Aceleaşi fişiere .class pot fi rulate fără modificare pe orice platformă pentru care există o JVM. Din acest motiv JVM este uneori numită şi procesor virtual.


Se numeşte compilator un program care converteşte un alt program, dintr-un limbaj numit sursă în limbaj maşină (unicul limbaj înţeles de direct de către "calculator"). După realizarea completă a conversiei, programul în limbajul maşină va rula foarte rapid pe calculator. Procesul de compilare se realizează în mai multe faze: analiza lexicală (colectează caracterele programului sursă sub forma atomilor lexicali), analiza sintactică (preia atomii lexicali şi construieşte o reprezentare a sintactică a programului în memorie), analiza semantică (verifică existenţa erorilor de tip), generare de cod (aici se va obţine prima formă a limbajului maşină). Deşi limbajul maşină generat de compilator ar putea fi rulat direct pe calculator deseori el trebuie rulat împreună cu alte programe dependente de sistemul de operare pe care urmează să fie rulată aplicaţia finală. De

exemplu, în cazul operaţiilor de intrare/ieşire, compilatorul va genera apeluri catre funcţiile de intrare/ieşire ale sistemului de operare (în Java acesul la sistemul de opeare se face pe baza unor biblioteci numite “pachete standard”). Pentru ca aplicaţie să funcţioneze în contextul respectivului sistem de operare, compilatorul va trebui să lege respectivele porţiuni ale sistemului de operare la programul tradus. Operaţia de legare conectează programele scrise de utilizatori la programele sistemului de operare prin plasarea adreselor punctelor de intrare în sistemul de opeare în programul scris de utilizator. Se numşte interpretor un program care rulează alte programe. Interpretorul este un simulator al unei unităţi centrale care extrage şi execută instrucţiuni ale unui limbaj de nivel înalt (şi nu limbaj maşină). Din punctul de vedere al conversiei în limbajul maşină, Java a implemetat un compromis între compilatoarele şi interpretoarele pure. Aici, limbajul sursă este compilat la o formă intermediară (între limbajul Java şi limbajul maşină) foarte uşor de interpretat (deoarece a fost deja decodificat o dată) numită “cod de biţi” cu ajutorul lui javac apoi, acest cod intermediar este rulat pe interpretorul codului de biţi, java (se zice că acest limbaj maşină este pentru o maşină virtuală - Maşina Virtuala Java - care este implementată nu la nivel hardware ci la nivelui interpretorului codului de biţi). Deci, în Java programul sură trece prin analiza lexicală, sintactică generarea de cod intermediar, apoi interpretare pentru a obţine relutate.



Nivele de securitate în Java

Limbaj și compilator

Limbajul este sigur prin proiectare. Construcţiile care permit manipularea directă a poantorilor au fost eliminate pentru evitarea erorilor în execuţie şi a fragmentării memoriei.

Încărcătorul de clase

Încărcătorul de clase garantează separarea stocării claselor ce provin de la surse locale de cele de reţea. În timpul execuţiei, motorul maşinii caută mai întîi referinţele la clasele locale şi numai apoi clasele referite din interiorul acestora. Din acest motiv, clasele locale nu vor fi suprascrise de cele încărcate din reţea. Astfel, se previne modificarea comportamentului deja cunoscut al claselor locale ca urmare a unei intervenţii externe (prin reţea).

Verificatorul codului de biţi

În timpul rulării unui program Java, JVM poate importa cod de oriunde. Java trebuie să fie sigur de codul importat respectiv că el este dintr-o sursă de încredere. Pentru realizarea acestei sarcini, motorul va realiza o serie de teste denumite “verificarea codului de biţi”.

Interfaţă locală restrictivă

Accesul la sistemul de fişiere local şi la resursele de reţea este controlat prin clase locale şi metode. Clasele sunt prin definiţie restrictive. Dacă un cod importat încearcă să acceseze sistemul de fişiere local, sistemul de securitate lansează un dialog cu utilizatorul.

Miniaplicații Java (Applets)

- **Cele mai vechi tipuri de aplicații**
- **Folosite în pagini HTML**
- **Pot include conținut activ (formulare, imagini, sunet, filme)**
- **Vizualizate în navigator și pot comunica cu un server**

Miniaplicații Java

O mare parte din popularitatea lui Java pornește de la posibilitatea dezvoltării unor aplicaţii mici (applets, în engleză) care pot fi înglobate în pagini HTML şi descărcate prin rețea atunci cînd pagina este vizualizată într-un navigator de Web.

Există două tipuri de miniaplicații Java: cu şi fară încredere (trusted şi untrusted, în engleză). Miniaplicaţiile "fără încredere" sînt restricţionate din punctul de vedere al accesului local, cele "de încredere" au permisiunea accesării resuselor locale.

Caracteristicile miniaplicațiilor

Miniaplicaţiile sînt de natură grafică, tinzînd să conţină controale cum sînt butoanele, cutiile de text, listele de selecţie. Totuşi, miniaplicaţiile pot să facă mult mai mult decît simpla afişare a unor controale (obiecte de interfață grafice). Iată o listă cu cîteva dintre posibilităţile miniaplicaţiilor:

- vizualizarea de animaţii, imagini şi redare de sunete;
- conectarea la baza de date aflată pe serverul de Web de pe care s-a descărcat miniaplicația;
- comunicația cu un server de Web prin socluri (soket, în engleză);
- comunicația cu o aplicația Java rulată pe sereverul de Web;
- pot utiliza CORBA (Common Object Request Broket Architecture) pentru a comunica cu Java sau cu aplicații ne-Java de pe server-ul de Web.

OMG (Object Management Group) este un consorţiu creat în scopul dezvoltării de standare în programarea orientată pe obiect. CORBA specifică condiţiile în care un obiect respectă specificaţiile OMG, în principiu, specificaţiile descriu standardul de interfaţă pentru aceste obiecte.

Aplicații Java

- **Crearea de aplicații de sine stătătoare:**
  - **JVM rulează aplicații independente**
  - **nu se încarcă clase prin rețea**
- **Crearea de aplicații client-server:**
  - **deservește mai mulți clienți dintr-o singură sursă**
  - **se încadrează în modelul multi-nivel pentru calcule pe Internet.**

Aplicații Java de sine stătătoare

Deşi, Java şi-a câştigat popularitatea datorită miniaplicaţiilor, e posibilă şi crearea unor aplicaţii de sine stătătoare (nu este nevoie de un navigator pentru rularea lor). Acestea sunt oarecum echivalentul programelor executabile din C şi C++.

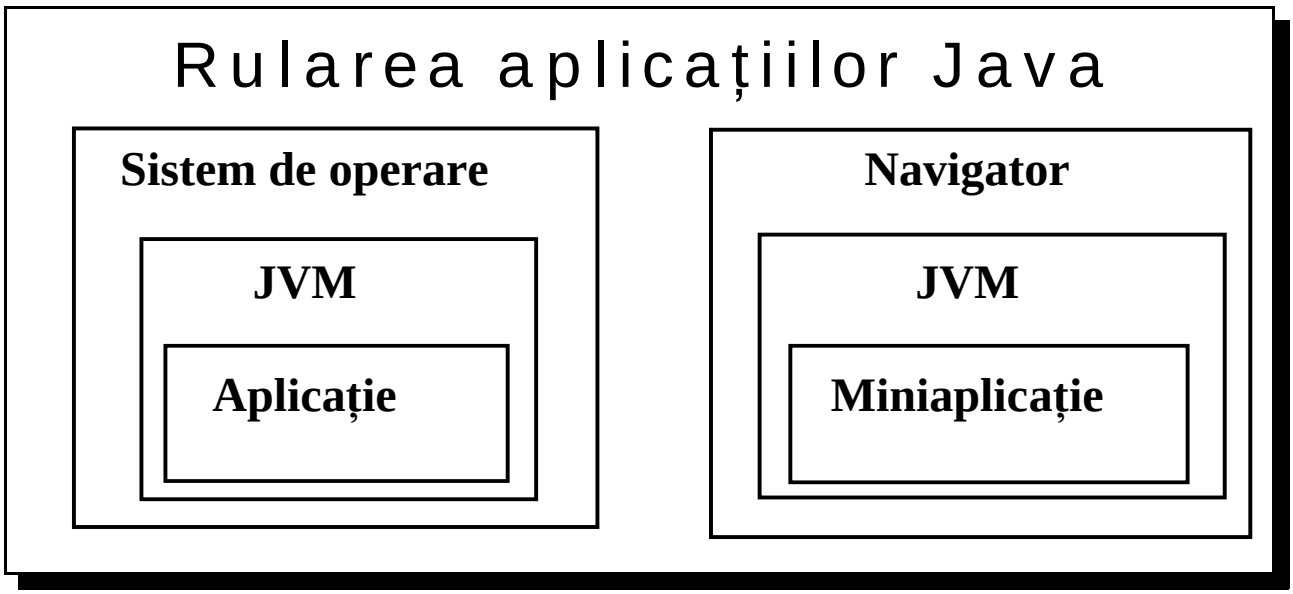
Aplicații client

Miniaplicaţiile Java pot fi rulate pe un sistem de operare local sub forma unor aplicaţii de sine stătătoare. Acest mod de lucru este specific dezvoltării de aplicaţii client (aplicaţia este descarcătă de pe un server dar rulată în navigator, care este clientul serverului de Web) şi este, uzual, fară restricţiile de securitate specifice miniaplicaţiilor. De exemplu, aplicaţiile Java pot accesa sistemul local de fişiere sau să stabilească conexiuni cu alte maşini prin reţea. Miniaplicaţiile pot beneficia de aceste facilităţi doar dacă au fost înregistrare că sînt de încredere; restricţia previne coruperea sistemului local de fişiere de către miniaplicaţii malițioase. Caracteristici adiţionale ale miniaplicaţiilor de încredere sînt:

- accesul la sistemul de fişiere local;
- accesul la o maşină distinctă de server-ul de pe care s-a descărca miniaplicaţia;
- lansarea în execuţie a unor programe noi sau încărcarea de biblioteci locale.

Aplicații sever

Aplicaţiile Java pot fi, de asemenea, rulate pe o maşină server dacă JVM este disponibilă pe platforma respectivă. Rularea pe server asigură dezvoltarea de aplicații după modelul celor 3 nivel specifice calculului pe Internet. Aplicaţia server rulează în mod continuu pe server şi trimite un răspuns unui client ca urmare a unei cerei efectuate de acesta.



Rularea aplicațiilor Java
Toate aplicaţiile Java, inclusiv miniaplicaţiile, se rulează la nivelul JVM. JVV este pornită diferit, după cu programul este o aplicație Java sau o miniaplicație.

Rularea aplicaţiilor
Aplicaţiile se rulează prin unei JVM locale direct din sistemul de operare. JVM interpretează programul Java şi îl converteşte în instrucţiuni maşină specifice platformei. Rularea programului se face plecând de la metoda statică numită `main()` pentru o aplicație de sine stătătoare sau printr-o referinţă la o clasă de miniaplicație dintr-un fişier HTML atunci cînd un program este încărcat din navigator.

Rularea miniaplicaţiilor
O miniaplicație Java este un tip special de program folost în paginile Web. Atunci cînd navigatorul de Web citeşte o pagină HTML cu marcajul de miniaplicație (<applet>) ve descărca miniaplicația prin reţea pe sistemul local şi va porni miniaplicația în JVM care vine inclusă în navigator.

Funcționarea JVM

- **Încărcătorul de clase al JVM încarcă clasele necesare fucționării aplicației**
- **Verificatorul JVM verifică secvențele ilegale de cod de biți**
- **Gestionarul de memorie al JVM eliberează memoria după terminarea aplicației**

Încărcătorul de clase al JVM

Rularea unui fișier cu extensia `.class` poate necesita şi alte clase pentru îndeplinirea scopului propus. Aceste clase sînt încărcate automat de încărcătorul (class loader) de clase în JVM. Clasele pot să fie stocate pe discul local sau pe un alt sistem, caz în care accesul se face prin rețea. JVM folosește variabila de sistem `CLASSPATH` pentru determinarea poziției fișierelor locale `.class`.

Clasele încărcate prin rețea sunt păstrate într-un spaţiu de nume separate de cel al claselor locale. Astfel se previn conflictele de nume şi înlocuirea sau suprascrierea unor clase standard prin clase malițioasă.

Verificatorul JVM

Sarcina verificatorului este să determine dacă codul de biţi interpretat nu violează regulile de bază ale limbajului Java şi că acesta este dintr-o sursă de încredere. Validarea asigură inexistenţa violării accesului la memorie sau execuția unor acțiuni ilegale.

Gestionarea memoriei

JVM urmărește toate instanțele (variabilele obiect) utilizate. Dacă o instanţă nu mai este folosită JVM are obligaţia să elibereze memoria folosită de obiect. Eliberarea memoriei se face după ce obiectul nu mai este necesar nu neapărat imediat după aceea. Procesul prin care JVM gestionează obiectele care nu mai sînt referite se numește colectare de gunoi (garbage collection).

Compilatoare JIT (Just-in-Time)

- **Cresc performanțele**
- **Utile dacă același cod de biți se execută repetat**
- **Optimizează codul repetitiv (ciclurile)**

Compilatoare JIT

JVM traduce codul de biți Java în instrucțiuni native al mașinii pe care se rulează. Ce se petrece dacă același cod trebuie executate din nou ceva mai tîrziu în program? În lipsa unui compilator JIT codul interpretat de la fiecare reluare a lui, chiar dacă a fost deja interpretata o dată ceva mai devreme deja.

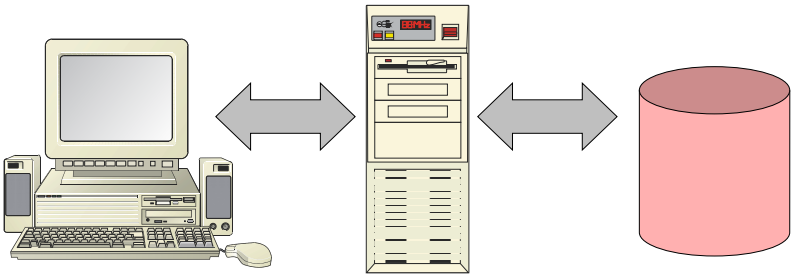
Avantajele compilatoarelor JIT

Majoritatea JVM suportă deja compilare JIT. Compilatoarele JIT traduc codul de biți numai la prima lui apartiție; dacă același cod va fi executat mai tîrziu, va fi automat asociat codului corespondent în limbaj mașină.

Compilatoatele JIT fac ca Java să funcţioneze mai repede în momentele în care se pune problema traducerii în mod repetat a unui cod de biți în instrucțiuni mașină native. Efectul este spectaculos în cazul ciclurilor sau a funcţiilor recursive. Unele compilatoare JIT sînt suficient de inteligenta ca să optimizeze traducerea unor secvențe de cod de biți în instrucţiuni native ale mașinii pe care se rulează aplicația.

Java şi calculul pe Internet

! **Calculul pe Internet are loc pe 3 nivele:**



Client **Server de aplicatii** **Server de date**

! **Java poate fi utilizat pentru oricare dintre aceste nivele**

Java şi calculul pe Internet

Produsele firmelor serioase pentru calcule pe Internet trebuie să suporte construcţia de componente Java pentru trei nivele de aplicaţie, precum şi instrumete pentru esenţiale pentru dezvoltare şi gestiune a alicaţiilor. Suportul se implementează sub forma de instrumente, aplicaţii, baze de date, servere de aplicaţii şi interfeţe pentru progrmarea aplicaţiilor.

Nivelul client

Atunci cînd Java trebuie să fie rulat pe o maşină client este tipic implementat într-o miniaplicaţie executata din navigator care poate comunica cu server-ul de pe care afost descărcat.

Nivelul serverului de aplicaţii

Java poate fi utilizate pe server-ul de aplicaţii pentru implementarea unor componente partajate şi reutilizabile ce ţin de logica aplicaţiei. Java mai poate fi utilizat la acest nivel de miloc pentru a furniza interfeţede control pentru aplicaţii generate sub forma unor pagini Web.

Nivelul serverului de date

Acest server stochează date, dar mai poate stoca şi executa cod Java, în particular acest cod manipulează intensiv date sau forţează reguli de validare specifice datelor.

Termenul de client-server apare în contextul sistemelor distribuite. Aplicația distribuită este o colecție de aplicații a căror distribuție este transparentă la nivelul utilizatorului astfel încît la aplicația pare să fie stocată pe o singură mașină locală. Într-o rețea de calculatoare utilizatorii simt că lucrează pe o mașină particulară, iar poziția lor în rețea, stocarea datelor încărcarea și funcționalitatea mașinii lor nu este transparentă. Sistemele distribite folosesc, uzual, o anumită formă de organizare cliet-server.

Clientul este o aplicație sau un proces care solicită un serviciu de la un alt calculator sau proces, denumit “server”. Server-ul este o aplicație care furnizează un anumit serviciu altor programe, numite “client”. Conexiunea între client și server se face la nivel de mesaje transmise prin rețea și folosește un protocol pentru codificareea cererilor clientului și al rapsunsurilor server-ului. Server-ul poate să ruleze continuu în așteptarea cererilor sau să fie pornit de o aplicație care controlează un grup de server-e. Acest model permite plasarea clenților și a serevr-elor independent, în nodurile rețelei, posibil pe hardware-uri și pe sisteme de operare diferite pentru a-și îndeplini funcțiile (server rapid/client lent).

Ce este JDK?

Mediul de dezvoltare JDK Sun conține:

- **Compilator**
- **Componentă de vizualizare a miniaplicațiilor**
- **Interpretor al codului de biți**
- **Generator de documentație**

Componentele Sun JDK sunt:

- **compilatorul** java `javac`, el compilează codul sursă Java în codul de biţi (bytecode) Java;
- `appletviewer` este componenta pentru **vizualizarea miniaplicațiilor**, aceasta citește un document HTML, descarcă miniaplicațiile referite în document şi le afişează într-o fereastră proprie. Se foloseşte ca o alternativă la navigatorul de Web pentru testarea miniaplicaţiilor Java;
- **interpretorul** codului de biţi Java, `java`, care este motorul ce rulează aplicațiile Java;
- **generatorul de documentație** în format HTML pe baza codului sursă Java este `javadoc`.

Alte scule JDK

- **depanatorul de clase** Java este `jdb` (asemănător cu depanatoarele dbx sau gdb din UNIX);
- `javakey` pentru generarea cheilor de **certificare** şi a codului Java de încredere (trusted);
- `jar` pentru generarea de arhive de calse şi de resurse;
- **dezasamblorul** codului de biţi Java într-un format inteligibil pentru oameni, `javap`;
- aplicaţia de **versionare** a claselor, `serialver`.

Pachete Java

- echivalentul bibliotecilor C
- pachetele standard sunt pentru lucrul cu:
  - ✓ Limbajul
  - ✓ Ferestre
  - ✓ Miniaplicaţii
  - ✓ Intrare/Ieşire
  - ✓ Comunicaţie în reţea

Pachete Java

Pachetele asigură bazele pentru funcționarea lui Java și sunt implementate sub forma unor serii de clase cu metode grupate după funcționalitate. Pachetele Java sunt echivalentul bibliotecilor din C. De exemplu, există un grup de clase care ajută la crearea și utilizarea conexiunilor de rețea, acestea sunt toate conținute în pachetul `java.net`. Pachetul de bază al limbajului Java este `classes.zip`.

Pachete Java standard

Aceste pachete conţin clasele fundamentale pentru toate aplicațiile și miniaplicațiile Java, cîteva mai importante sunt:

Pachet Java	Rol	Clase incluse
`java.lang`	Colecţia claselor de bază din Java. Conţine rădăcina ierarhiei de clase Java, `Object`, definiţiile tuturor tipurilor primitive etc.	`Object, String, Thread, Runtime System`
`javax.swing`		`Window, Button, Menu, Graphics`
`java.applet`	Clase pentru suportul scrierii miniaplicaţiilor.	`Applet, AudioClip`
`java.io`	Colecţia de clase de intrare/ieşire (inclusiv pentru accesul la fişiere)	`File, InputStream, OutputStream`
`java.net`		`Socket, DatagramPacket, URL, InetAddress`

s

Împachetări, platforme şi versiuni Java

Împachetări:

- **J2SE - Java 2 Platform, Standard Edition**
- **J2EE - Java 2 Platform, Enteprise Edition**
- **J2ME - Java 2 Micro Edition**

Platforme (SO + UCP):

- **Solaris, Windows ... + SPARC, Intel ...**

Versiuni:

- **Java release 1.0, 1.1; Java release 2, versiunea 1.2, 1.3, 1.4, 1.5, 1.6**

Împachetări Java

Tehnologia Java bazată pe JVM este strucurată pe trei tipuri de produse proiectate în funcţie de cerinţele particulare ale pieţei de software:

- J2SE - Java 2 Platform, Standard Edition: se foloseşte pentru "applet" şi aplicaţii care rulează pentru un singur calculator;
- J2EE - Java 2 Platform, Enteprise Edition: pentru aplicaţii client/server distribuite;
- J2ME - Java 2 Micro Edition: pentru crearea de aplicaţii care rulează pe un dispozitiv consumator (PDA, telefon celular etc.).

Fiecare pachet de tehnologii Java are o ediţie SDK (Software Development Kit) prin care se pot crea, compila, executa programele în tehnologie Java pentru o platformă particulară.

Platforme Java

Familia de produse a tehnologiilor Java este strâns legată de J2SE SDK deoarece majoritatea programatorilor îşi încep cariera pe PC-uri prin scrierea de applet-uri. Sun a dezvoltat Java 2 Platform, Standard Edition SDK pentru următoarele platforme:

Sistem de operare	Procesor
Solaris OE	chip SPARC pe 32 biţi
Microsoft Windows	chip Intel pe 32 de biţi
Solaris OE	chip SPARC pe 64 de biţi

Sistem de operare	Procesor
Linux	Intel

Versiuni Java

Prima versiune a lui Java a fost 1.0. Acesta a apărut pe piaţă sub denumirea de JDK 1.0, în anul 1996 şi conţinea maşina virtuală Java, bibliotecile de clase şi instrumentele specifice pentru dezvoltare (compilator etc.) aplicaţiilor. Evoluţia versiunilor de limbaj este prezentată în tabelul următor:

Versiune	An
JDK 1.0	1996
JDK 1.1	1997
JDK 1.2 (cunoscut şi sub denumirea de Java 2)	1998
JDK 1.3	2000
JDK 1.4	2002
JDK 1.5 (cunoscut şi sub denumirea de Java 5)	2004
JDK 1.6 (Java 6)	2006

Începând cu JDK 1.2 toate versiunile au fost numite, pe scurt, Java 2. În timpul evoluţiei lui Java, modificările aduse limbajului iniţial au fost puţine, schimbări majore au avut însă loc la nivelul bibliotecilor acestuia şi al integrării limbajului în contextul unor tehnologoo software strâns legate de Java. Numerotarea de 2 (din Java 2) arată că limbajul a progresat, intrând oficial în perioda lui "modernă", însă din dorinţa de a păstra continuitatea numerotării versiunilor de bibliotecă s-a acceptat că întregul limbaj să fie versionat pe baza acestora. Dincolo de modificările şi adăugirile de limbaj, în Java 2, apare pentru prima oară conectarea lui la un mediu de dezvoltare al aplicaţiilor. În afară de JDK, mai nou, Sun furnizează, şi separat componentele (JVM şi bibliotecile) necesare rulării de aplicaţii Java (fără posibilitatea dezvoltării de aplicaţii) sub denumirea de JRE (Java Runtime Environement).

2

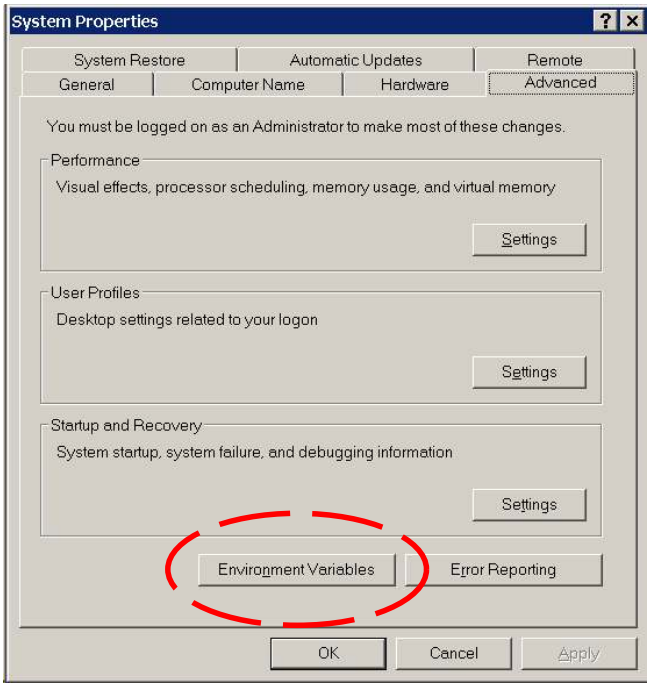
Instalarea, configurarea JDK J2SE şi rularea unei aplicații Java

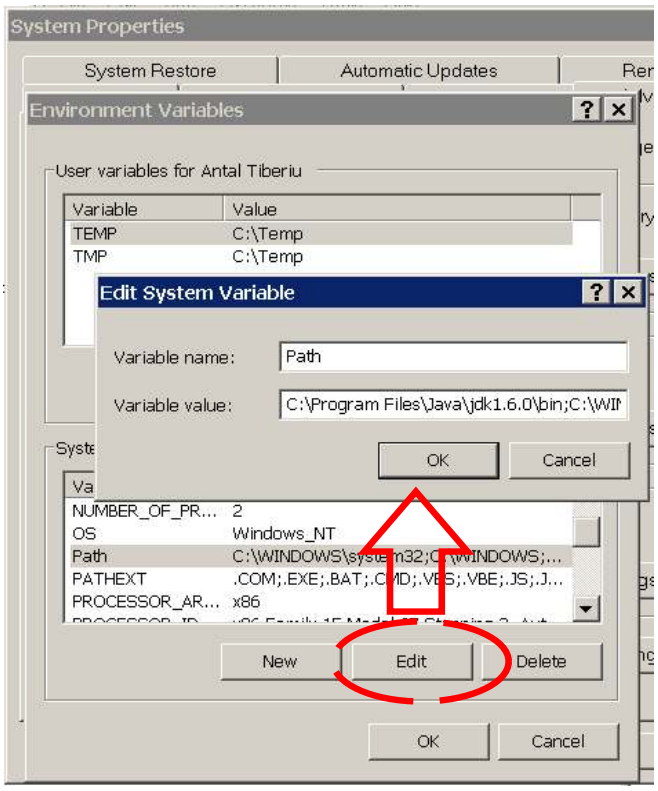
Instalrea şi configurarea JDK J2SE

1. **Descărcare J2SE de la http://java.sun.com/j2se:** `jdk-6-windows-i586.exe`
2. **Instalare JDK în C:\Program Files\Java\jdk1.6.0**
3. **Despachetare surse (**`src.zip`**) şi documentaţie (**`jdk-6-doc.zip`** de la** `http://java.sun.com/docs` **)**
4. **Configurare SO: Control Panel > System > Advanced > Environment Variables ➡ PATH**
5. **Verificarea configurării: Start > Run > cmd cu path şi java- version**

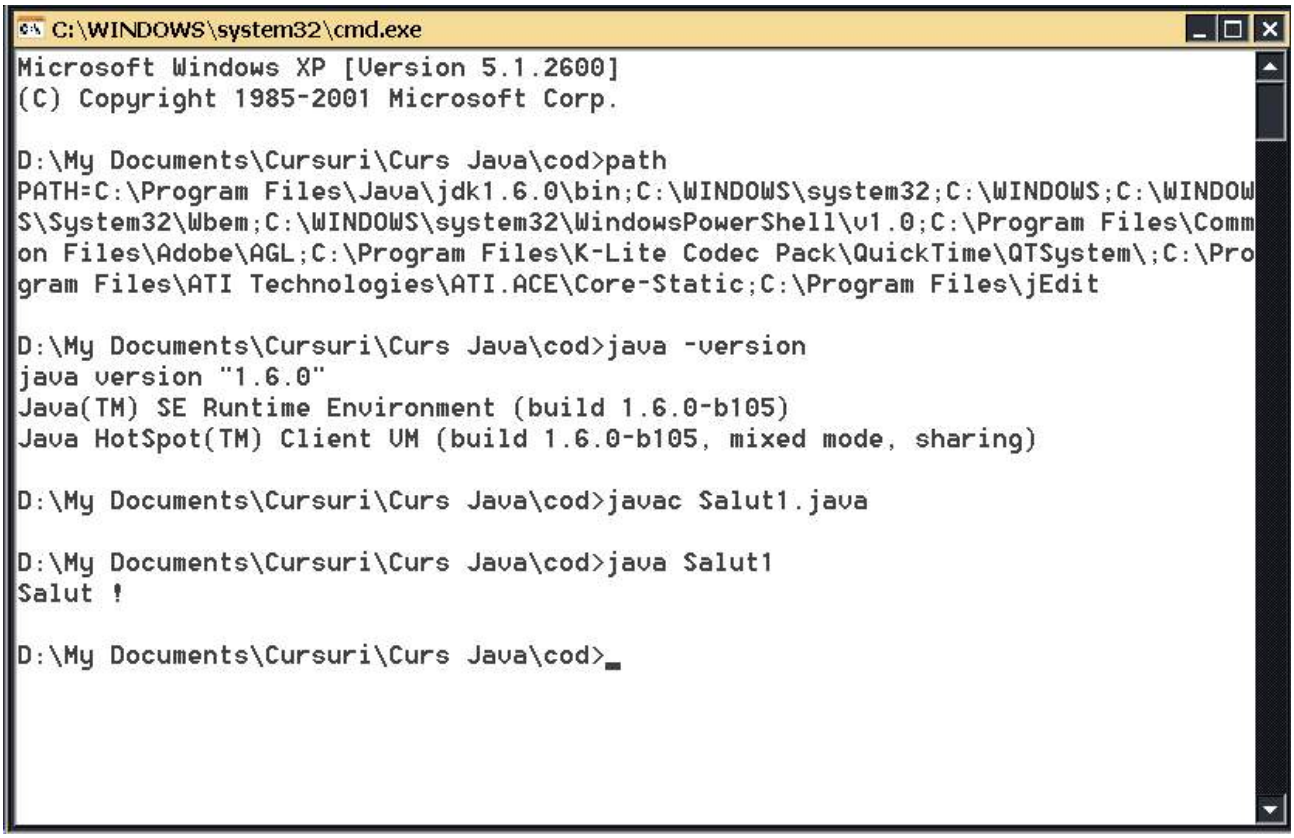
Instalarea JDK

Ultima versiune de JDK se descarcă de pe site-ul Sun-ului (http://java.sun.com/j2se), pentru Windows, împachetarea J2SE fiind sub forma fişierului executabil **jdk-6-windows-i586.exe**. Acesta conţine şi JVM-ul care se va instala automat împreună cu JDK-ul. După instalare calea jdk/bin trebuie adăugată în lista fişierelor executabile pentru ca sistemul de opearare să găsească automat pe javac, java etc. Pentru aceasta, în Windows XP, mergem la *Control Panel > System > Advanced > Environment Variables*.



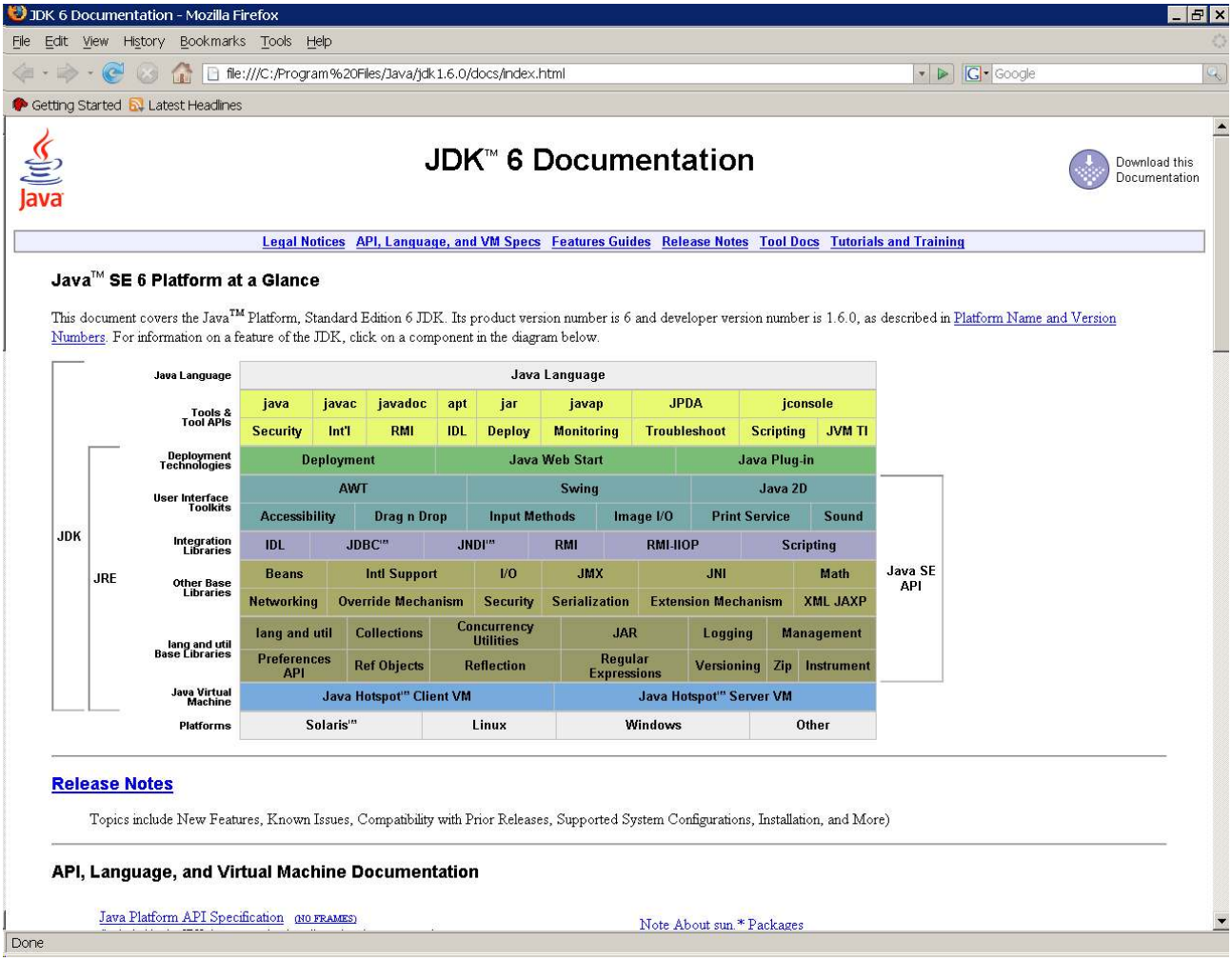


Aici, căutăm prin User Variables până găsim pe PATH, unde vom scrie pe `C:\Program Files\java\jdk1.6.0\bin`, apoi apăsam butonul **OK.** Deschideţi o consolă windows nouă (din `Start > Run > ... >cmd`) în directorul în care se află stocat fişierul sursă Java (mai jos acest director este `D:\My Documents\Cursuri\Curs Java\cod`), apoi verificaţi dacă instalarea şi configurarea este corectă prin `path` şi `java -version`. Path ne arată că am introdus corect calea, iar java -version că JVM-ul s-a instala corect.



Instalarea surselor de biblioteci şi documentaţiei Java

Fişierele care conţine sursele bibliotecilor din JDK se instalează într-o formă comprimată în fişierul `src.zip`. Este recomandată decomprimarea şi instalarea lui pentru accesul la codul sursă în directorul `src`. Documentaţia este conţinută într-un fişier separat de JDK ce trebuie descărcat de pe `http://java.sun.com/docs` , pentru J2SE, versiunea 6 acesta este `jdk-6-doc.zip`. Acest fişier se va decomprima în directorul `docs` şi aeste în format HTML. Va putea fi vizualizate cu orice navigator de Internet începând cu `index.html`. Cel mai simplu este să se pună un bookmark în navigator către acest fişier.



Structură de directori ai JDK-ului va fi în final:

jdk (numele versiunii actuale este jdk1.6.0, se modifică funcţie de versiune)
*) bin (compilator java şi alte instrumente de dezvoltare)
*) demo (exemple de aplicaţii Java)
*) docs (documentaţia bibliotecile în HTML, după decomprimarea lui `jdk-6-doc.zip`)
*) include (fişiere pentru compilarea metodelor native)
*) jre (fişierele mediului de execuţie Java)
*) lib (fişierele de bibliotecă)
*) src (fişierele sursă de bibliotecă, după decompimarea lui `src.zip`)

Etapele generării unei aplicații Java

Etapa:

1 Editarea: Salut1 ➡ salvare pe disc cu extensia .java

2 Compilarea: javac Salut1.java ➡ Salut1.class

3 Rularea: java Salut1

A. Încărcătorul de clase copiază conţinutul lui Salut1.class de pe disc în RAM

B. Verificatorul codului de biţi se asigură că sistemul de securitate Java nu este violat

C. Interpretorul citeşte din RAM şi traduce codul de biţi în limbaj maşină şi îl execută.

Etapele necesare rulării unei aplicaţii Java

Stocarea textul sursă.

Toate programele Java sunt formate din obiecte. Există două tipuri de obiecte: tipurile de date fundamentale (numite unori, primitive sau de bază) şi tipurile de date definite de utilizator, numite clase. Tipurile primitive sunt definite la nivelul limbajului Java, iar clasele sunt definite de către programator sau de către alţi dezvoltatori. În continuare se prezintă o aplicaţia Java, formată dintr-o clasa numită **`Salut1`**, care va afişa pe ecran textul **`Salut!`**.

Extensia fişierului sursă trebuie sa fie .java, iar numele acestuia Salut1. Numele clasei şi cel al fişierului sursă trebuie să fie acelaşi.

Conţinutul fişierului sursă Java Salut1.java este:

```
// Salut1.java
public class Salut1 {
   // executia unei aplicatii Java incepe cu metoda main.
   public static void main( String args[] )
   {
      System.out.println("Salut !" );
   } // terminare metoda main
} // terminare clasa Salut1
```

Explicarea liniilor codului sursă Java

Linia `// Salut1.java` reprezintă un comentariu care se întinde pe o singură linie (începând cu primul carater de după `//` şi până la trecerea la linia nouă) din textul sursă. Comentariile ajută la înţelegerea codului Java explicând rolul unor secvenţe de cod. Ele nu apar în programul executabil, liniile în cauză fiind ignorate de către compilator. Atunci când este nevoie de comentarii mai lungi, ce

trebuie să se întindă pe mai multe linii, se va folosi perechea de delimitatori `/*`, `*/`.

Limbajul Java este “case sensitive” adică face diferenţa între scrierea cu litere mari şi mici. Dacă în loc de main se va scrie Main aplicaţia nu se va rula.

`public` se numeşte **modificator de acces** , acesta permite definirea controlul accesului altor porţiuni de program la această porţiune de cod.

`class` reprezintă începutul unei declaraţii de clasă şi este urmată de numele clasei (mai sus, `Salut1`). Orice aplicaţie Java trebuie conţină cel puţin o singură declaraţie de clasă. Clasa reprezintă un şablon pentru descrierea stării şi a comportamentului asociat unui obiect din acea clasă. Procesul de creare a unei obiect pe baza unei clase se numeşte instanţiere. Starea unui obiect este stocată în variabilele membri, iar comportamentul ei se implementează prin metode.

`{` - acolada deschisă marchează începutul corpului declaraţiei de clasă şi va avea întotdeauna corespondentă o acoladă închisă - `}` ce marchează terminarea declaraţie de clasă. Porţiunea de cod sursă delimitată de aceste acolade poartă denumire de bloc.

Linia este locul de pornire a programului `public static void main( String args[] )`. Numele `main` este urmat de o paranteză rotundă `(`, un parametru şi o paranteză rotundă închisă `)`. Prezenţa parantezei spune compilatorului Java că `main` este o metodă şi nu o altă construcţie de limbaj. Atunci când dorim să rulăm o clasă pe JVM trebuie să-i specifică doar numele, interpretorul Java apelând automat metoda main a clasei. Metoda main este precedată de trei modificatori:

`public` - care permite ca orice altă clasă să poată apela metoda main;
`static` - spune că metoda main nu operează cu obiecte (deoarece în momentul pornirii aplicaţiei încă nu avem obiecte), tehnic vorbind aceasta nu-l are pe `this` (vezi referinţa this). În general, metoda main este cea care construieşte obiectele necesare programului. Dacă însă dorim să apelăm numai metode ale clasei nu mai este necesară construcţia (aceasta se face cu operatorul `new`) lui; construcţia este obligatorie dacă se doreşte accesarea de metode sau de variabile de instanţă (vezi conceptul de clasă);
`void` - indică faptul că metoda main nu întoarce o valoare.

Linia `{System.out.println("Salut !");}` este echivalentă cu liniile (în sensul că din punctul de vedere al compilatorului Java efectul compilării este acelaşi - rezultă acelaşi cod de biţi - dar pentru o citire şi înţelegre mai uşoară se preferă cea de a două scriere - cu spaţii şi pe mai multe linii):

```
{
  System.out.println(  "Salut !"  );
}
```

Aici acoladele marchează începutului şi terminarea corpului metodei main. Metoda are o singură instrucţiune care foloseşte un pachet de intrare/ieşire pentru afişarea în fereastra din care s-a rulat aplicaţia a textului `Salut !`. Textul de afişat se scrie între ghilimele şi poartă denumirea de şir de caractere sau mai pe scurt, şir. Metodele în Java pot avea parametri, metoda `System.out.println` face afişarea parametrului şir. Caracterul `;` se numeşte terminator şi face ca metoda să fie considerată o instrucţiune Java. Clasa `System` este din pachetul `java.lang` şi conţine, printre altele, metode pentru interacţiune între JMV şi mediul extern (recunoaşte carecteristicile platformei pe care rulează). Prin `System` se pot accesa fişierele standard de intrare (`in`), ieşire (`out`) şi de erorare (`err`).

Este o eroare de sintaxă dacă un şir nu este cuprins în ghilimele la nivelul programului sursă sau dacă se omite un caracter terminator de instrucţie ;.

Compilarea programului sursă Java

Pentru compilarea programului Java stocat în fişierul **`Salut1.java`** se va folosi compilatorul **`javac`** astfel: **`javac Salut1.java`**. Compilatorul Java va crea fişierul **`Salut1.class`** ce conţine codul de biţi al aplicaţiei.

Rularea aplicaţiei Java

Deoarece codul de biţi este o reprezentare intermediară programului Java el nu poate fi rulat direct de sub sistemul de operare. Rularea necesită JVM care se porneşte cu **`java Salut1`**. JVM trebuie să primească numele clasei de rulat, în exemplul prezentat acesta este **`Salut1`**. Dacă numele clasei este acelaşi cu cel al fişierului în care acesta este stocat, căutarea clasei se va face automat pe baza numelui de fişier care are extensia **`.class`**.

Erori în scrierea unei aplicații Java

1. Eroarea de sintaxă : apare atunci când compilatorul nu recunoaşte o linie de cod, reprezentând o violare a regulilor de scriere din limbaj. Compilatorul afişează un mesaj de eroare pentru ca programatorul să o poată identifica şi corecta.

2. Eroare de nerespectare a unor convenţii de limbaj:

- **O clasă public trebuie să aibă numele fişierului în care se stochează identic cu cel a numelui clasei**
- **Extensia fişierului trebuie să fie obligatorui .java pentru fişierul care conţine declaraţiile de clasă.**

Erorile de sintaxă se mai numesc şi erori de compilare deoarece compilatorul le detectează în faza de compilare. Aplicaţia nu va putea fi rulată decât după corectarea tuturor erorilor de sintaxă.

Variabile

- **Variabila = nume dat unui grup de locaţii de memorie ce la un moment dat, stochează o singură valoare**
- **Orice variabilă trebuie declarată**
- **Caraterizată prin: nume, valoare, tip, persistenţă şi vizibilitate.**
- **Variabilele pot fi iniţializate în momentul declarării**

Tip Nume Valoare

```
int i = 5;
```

La nivel conceptual, **variabila** este o abstactizare a componentei electrice numite memorie internă. La nivel de limbaj variabila reprezintă **un nume simbolic**, numit şi identificator, prin intermediul căruia vom putea accesa şi stoca valori în RAM-ul calculatorului. În Java orice variabilă trebuie declarată explicit, adică pentru fiecare nume de variabilă din program este obligatoriu să avem o line de forma `Tip Nume = [Valoare iniţială];` ce se citeşte, variabila cu numele `Nume` se declară de tipul `Tip`. Declaraţia leagă (asociază) de numele variabilei un grup de caracteristici. Variabila poate fi declarată în orice locaţie din bloc, dar este preferabil să fie scrisă la începutul acestuia. Se pot declara mai multe variabile într-o singură linie.În Java **variabilele** sunt de două feluri: **locale** (declarate într-o metodă) şi **câmp** (declarate într-o clasă). Caractersticile legate de numele variabilei prin declaraţie sunt:

Numele variabilei este formula prin care variabila este identificată în textul aplicaţiei Java.

Valoarea variabilei este conţinutul locaţiilor de memorie alocate în RAM. Modul de interpretare al locaţiilor este determinat de tipul variabilei.

Tipul asociat unei variabile prin declaraţie determină mulţimea valorilor şi a operatorilor ce pot acţiona asupra variabilei. Java este un limbaj puternic tipizat, adică tipul oricărei expresii, în particular a unei variabile, trebuie să poată fi determinat în momentul compilării.

Persistenţa defineşte intervalul de timp din execuţia aplicaţiei Java în care se zice ca variabila există (în sensul că i se alocă o anumită regiune de RAM). Variabilele locale există atât timp cât metoda în care s-au declarat este activă în sensul că aceasta se rulează. Variabilele câmp există atât timp cât obiectul al cărui membru sunt există. Dacă variabila câmp este şi statică câmpul va exista cât timp clasa respectică rămâne încărcată în JVM

Vizibilitatea defineşte porţiunea de cod din care un nume de variabilă este recunoscut de compilator. În afara acestei porţiuni de cod utilizarea respectivului nume va genera o eroare de sintaxă. Pentru variabilele declarate în interiorul unui bloc, vizibilitatea începe din locul declaraţiei şi se termină la acolada de închidere a blocului. Este posibil într-un bloc intern acelaşi numele de variabilă să fie redeclarat, situaţie în care vizibilitatea numelui extern nu o include pe cea a numelui intern.

Iniţializarea variabilelor se poate face explicit, adică ele primesc o valoarea explicită în momentul delcarării. Dacă această valoare implicită nu este dată, variabilele de tipul întreg primesc valorea 0.

Nume de variabile

- **Numele unei variabile trebuie să înceapă cu o literă din alfabet sau underscore (_) sau $**
- **Următoarele caractere pot conţine şi cifre, dar nu spaţii sau operatori (+, -, *, / etc.)**
- **Nu se poate folosi un cuvânt rezervat al limbajului Java pe post de nume de variabilă**
- **Lungimea numelui este nelimitată**
- **Este semnificativă scrierea cu litere mari sau mici**

OK - ✔	Greşit - ✘
`a x1 masa x2 masa$ideala masa_ideala`	`masa#ideala a-1 x*1 for if 1x`

Variabile locale

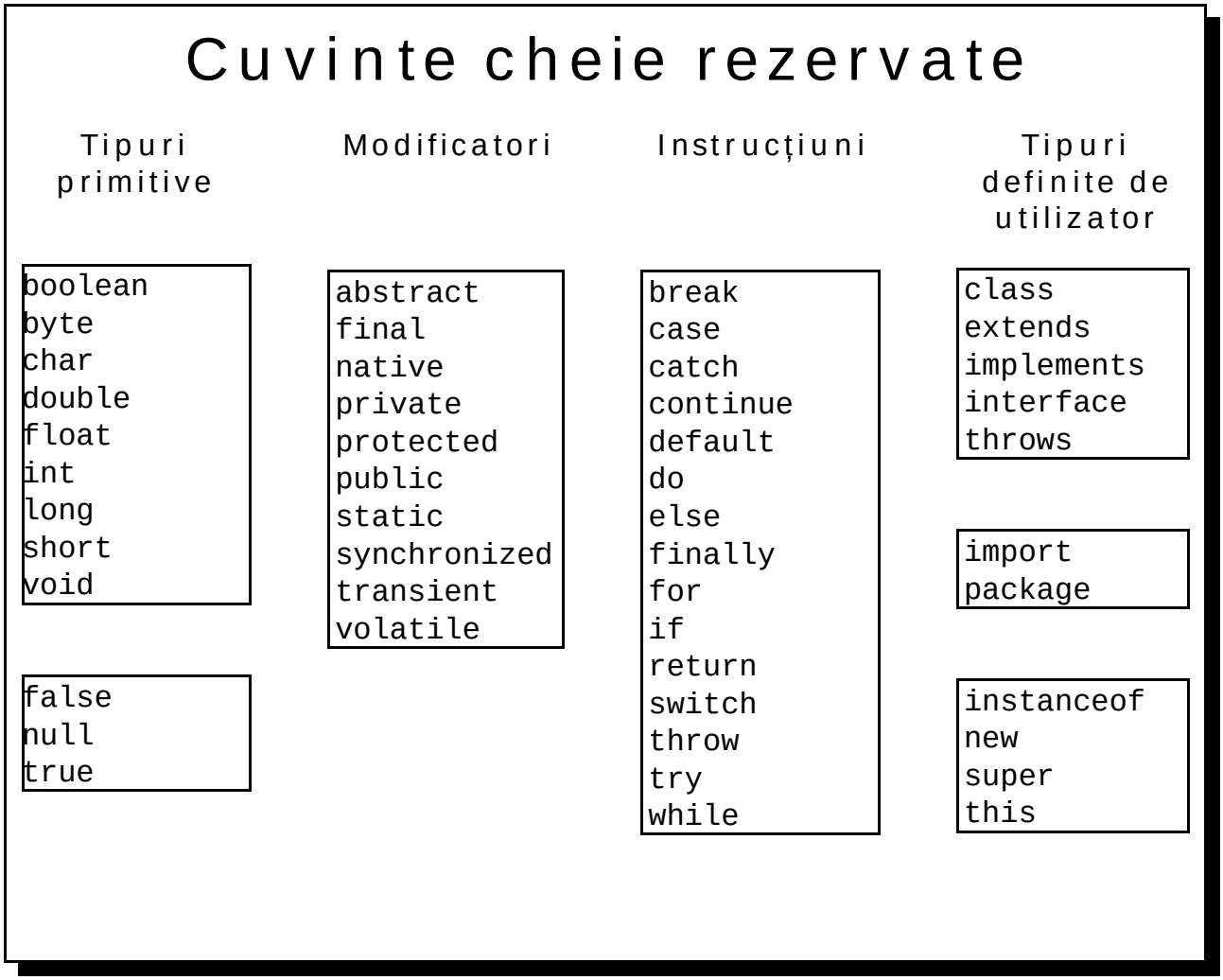
Variabilele locale sunt declarate într-o metodă sau un bloc de cod Java. Din acest motiv vizibilitatea şi persistenţa lor este restrânsă la respectiva porţiune de cod. Acestea vor putea fi acceste (sunt vizibile) numai în interiorul respectivului bloc (variabilele declarate în afara metodelor pot fi accestate din orice metodă) şi vor avea alocat un spaţiu propriu în RAM atât timp cât blocul de cod respectiv se rulează. O variabilă locală trebuie să primească valoare înainte de a fi utilizată într-o expresie, altfel compilatorul va da un mesaj de eroare.

Fie declaraţia:

```
int i = 5;
```

Aceasta declară o variabilă cu numele `i` de tipul `int` şi care are valoarea iniţială `5`.

Variabila, ca rezultat al unui proces de abstractizare, constă dintr-un grup de caracteristici (o denumire alternativă este cea de atribute). Procesul de asociere a unei valori unei caracteristici de variabilă poartă denumirea de legare (**binding**). Unele dintre carateristicile variabilei sunt cunoscute la momentul compilării, de exemplu numele şi tipul acesteia. Se zice că aceste caracteristici sunt legate static (**static binding**). Alte caracteristici, de exemplu valoarea - cu excepţia cazului când aceasta este iniţializată explicit, se leagă în timpul rulării aplicaţiei. Se zice că ele sunt legate dinamic (**dinamic binding**).



Se numeşte gramatică o mulţime de definiţii formale ce alcătuiest structura sintactică (numită, pe scurt şi sintaxă) a unui limbaj. Gramatica este definită în termenii unor reguli de generare ce descriu ordinea constituenţilor dintr-o propoziţie. Fiecare regulă are o parte stângă, numită categorie sintactică şi o parte dreaptă formată dintr-o secvenţă de simboluri. Simbolurile pot fi terminale sau neterminale. Un simbol terminal corespunde unui constituent de limbaj care nu mai are structură sintactică internă (este un element minimal de limbaj). Cuvintele cheie reprezintă o mulţime de simboluri terminale care fac parte din sintaxa limbajului Java. Restricţia de bază referitoare la acesta este cea conform căreia nu se pot da nume de variabile (sau alte elemente de limbaj definite de programator) care sa aibă aceeaşi scriere cu acestea.

Pe lângă cuvintele cheie de mai sus, numele `const` şi `goto` sunt şi ele rezervate şi nu pot fi folosite ca nume de variabile.

Tipuri de date în Java

- **Tipuri de date primitive:**
  - **6 tipuri de date numerice:**
    - **întregi:** `byte`, `short`, `int`, `long`
    - **reali:** `float`, `double`
  - **1 tip de dată pentru caractere:** `char`
  - **1 tip de dată bolean:** `boolean`
- **Tipuri de date definite de utilizator:**
  - **clasele:** `class`
  - **intefeţele:** `interface`
  - **tablourile**

Tipuri de date primitive

Tipurile de date primitive sunt implementate la nivelul JVM. Tipic pentru Java este că numele lor începe cu literă mică. Astfel, `float` este un tip de data primitiv, în timp ce `Float` este un obiect.

Tipuri întregi

Tipurile întregi se folosesc pentru manipularea numerelor ce nu au parte zecimală. Valorile numerice întregi pot fi atât negative cât şi pozitive, se mai spune că pot fi cu semn. Tipurile întregi din Java sunt:

Tip	Spaţiu alocat	Domeniu de valori
`byte`	1 octet	-128 la 127
`short`	2 octeţi	-32,768 la 32,767
`int`	4 octeţi	-2,147,483,648 la 2,147,483,647
`long`	8 octeţi	-9,223,372,036,854,775,808 la -9,223,372,036,854,775,807

Exemple de valori întregi:

- în baza 10: 1, 2, 6700000000000000000L
- în baza 8 (numărul este prefixat cu 0): 07
- în baza 16 (numărul este prefizăt cu 0x): 0xaa

Tipuri reale

Tipurile reale se folosesc pentru manipularea numerelor ce au pare zecimală. Ele se repezintă în virgulă flotantă conform standardului IEEE 745. Tipurile reale din Java sunt:

Tip	Spaţiu alocat	Domeniu de valori	Precizie
`float`	4 octeţi	aproximativ ± 3.40282347E+38F	7 zecimale
`double`	8 octeţi	aproximativ ± 1.7.9769313486231570E+308	15 zecimale

La tipul `double` precizia este dublă faţă de tipul `float`.

Exemple de valori reale:

`float`: 1.34F;
`double`: .337, 4.345345E108, 3.0, 3.46D\

Tipul caracter

Tipul `char` se foloseşte pentru stocarea caracterelor individuale, spre deosebire de tipul obiect `String` ce se foloseşte pentru stocarea şirurilor de caractere. Codificarea caractereleor în Java respectă standardul Unicode pentru reprezentarea caracterelor în orice limbă pe 16 biţi (2 octeţi). Primele 256 de caractere corespund cu mulţimea de caractere ISO Latin 1 din a cărei parte este şi codificarea ASCIII.

Exemple de valori caracter:

'A' - litera A din alfabet
'\u03C0' - valoare definită prin codificare Unicode în hexazecimal a lui π
'\n' - caracter special - line feed

Majoritatea valorilor de tipul caracter se afişează pe ecran, există totuşi un grup de caractere neafişabile acesta fiind rezevate pentru comanda dispozitivelor de afişare. Tipic, ele încep cu backslash \ urmate de un alt caracter

Tipul bolean

Tipul `boolean` are două valori distincte, `false` şi `true`. Se foloseşte pentru evaluarea unor condiţii locice. Nu se poate converti la o valoare întreagă

Tipuri de date definite de utilizator

Clasele, intefeţele, tablourile sunt tipuri de date definite de utilizator. După definirea acestora de vor putea declara variabile de acel tip după modalitatea utilizată la tipurile primitive.

Literali

Literalii reprezintă valori care nu sunt stocate în variabile.

! **numerici:**

- întregi: (Implicit int)

0	18	-23232	(`int` în baza 10)
02	077	0123	(`int` în baza 8)
0x0	0xff	0X1FF	(`int` în baza 16)
1L	022L	0x1FFFL	(`long`)

- reali: (Implicit double)

1.0	4.2	0.47	(`double`)
1.23e12		4.12E-9	(`double`)
6.3f	5.62F	4.12E9F	(`float`)

! **nenumerici:**

- booleeni:

```
true false
```

- caracter:

```
'a' '\n' '\077' '\u005F'
```

- şir:

```
"Salut/n"
```

Literalii şir se formează prin cuprinderea între ghilimele a unui grup de caractere. Pentru manipularea acestra se va folosi un tip obiect numit `String`, implementet la nivel de bibliotecă în Java şi nu tipul primitiv `char` ce poate stoca numai o singură valoare.

Erori la declararea de variabile

Fie următoarele declaraţii de variabile:

```
1 byte b = 130;
2 short s1 = 123, s2
3 int i = b*b*b*b;
4 long l = i+i+i;
5 double new=73.8;
6 boolean mergeinvacanta = true;
```

```
7 boolean max = s1>b;

8 char sal=”Salutare !”;

9 char amic = ’a’;
```

Linia 1: b este o variabilă de tipul byte, deci poate stoca valori în domeniul -128 , 127. Compilatorul va da un mesaj de eroare deoarece valoarea de iniţializare este în afara domeniului admis.

Linia 2: Este omis terminatorul de instrucţiune (;) de la sfărşitul declaraţiei.

Linia 5: Declaraţia este incorectă deoarece `new` este cuvânt rezervat în Java;

Linia 8: Variabilele de tipul `char` pot stoca o singură valoare. Dacă se foloseşte `String` în loc de `char` e Ok.

Iată două aplicaţii ce folosesc declaraţii de variabile locale şi pachete Java diferite pentru a calcula masa ideala în funcţie de vârsta şi înalţinea unei persoane:

Varianta 1:

```
import java.util.Scanner;

public class MasaIdealaV1 {
 public static void main(String[] args) {
    //declaratii de variabile locale
    double masa, varsta, inaltimea;
    Scanner intrare;

    intrare = new Scanner(System.in);


    //afisarea pe ecran a textului Ce varsta ai:
    System.out.print("Ce varsta ai: ");
    //citirea unui numar real de la tastatura
    varsta = intrare.nextFloat();

    System.out.print("Ce inaltime ai: ");
    inaltimea = intrare.nextFloat();

    //formula de calcul a masei ideale
    masa = 50 + 0.75 * (inaltimea-150) + 0.25 * (varsta - 20);

    System.out.println("Masa ideala (barbat) = " + masa + " kg");
    System.out.println("Masa ideala (femeie) = " + 0.9*masa + " kg");

 }
}
```

Varianta 2:

```
import javax.swing.JOptionPane;
public class MasaIdealaV2 {
 public static void main(String[] args) {

  //declaratii de variabile locale
  float masa, inaltimea;
```

```
  int varsta;

  //afisarea pe ecran a ferestrei de dialog Ce varsta ai:
  String intrare = JOptionPane.showInputDialog("Ce varsta ai: ");
  //citirea unui numar real de la tastatura
  varsta = Integer.parseInt(intrare);

  intrare = JOptionPane.showInputDialog("Ce inaltime ai (in cm): ");
  inaltimea = Float.parseFloat(intrare);

  masa = 50F + 0.75F * (inaltimea-150F) + 0.25F * (varsta - 20F);

  String masaideala = "Barbat = " + masa + " kg\n";
  masaideala = masaideala + "Femeie = " + 0.9*masa + " kg";

    JOptionPane.showMessageDialog(null,    masaideala,"Masa
ideala",JOptionPane.INFORMATION_MESSAGE);
 }
}
```

Constante Java

În Java cuvântul cheie `final` defineşte o constantă.

Se declară la fel ca şi o variabilă însă poate primi valoare o singură dată.

Prin convenţie, numele constantelor se scriu cu majuscule.

```
final double PI = 3.1415926535;
```

Dacă constanta este declarată într-o metodă, ea va fi vizibilă numai în interiorul acesteia. În situaţia în care se doreşte ca o constantă să fie vizibilă la nivel de clasă se zice că declarăm o constantă de clasă şi tebuie să folosim cuvintele cheie **`static final`** . Declaraţia ei se scrie în afara metodelor clasei şi va fi de forma:

```
public static final double PI = 3.1415926535;
```

Comcepte de programare orientată pe obiect

Abstractizare ☞	**Trecerea de la realitate la un model (o simplificare a realităţii)**
Clasă ☞	**Tip de dată definit de utilizator ce realizează încapsulare a datelor (câmpuri) şi a operaţiilor (metodele) ce se pot face cu datele**
Instanţiere ☞	**procesul de creare a unei variabile de tipul clasă prin folosirea lui `new`**
Obiect ☞	**denumire dată unei variabile de tip clasă**

Soluţionarea unei probleme implică izolarea ei de realitate. Se numeşte abstractizare procesul prin care se elimină detaliile realităţii fiind păstrate numai acele aspecte ce se consideră a fi relevante pentru soluţionarea problemei. Prin procesul de abstractizare se obţine un model al realităţii. Activitatea de trecere de la acest model la universul sistemului de calcul folosit pentru în scopul obţinerii rezultatelor poartă denumirea paradigmă sau metodologie (tehnologie) de programare.

În paradigma programării orientate pe obiect, modelarea realităţii de face prin conceptul de obiect. Obiectul este o abstractizare unei realităţi. Realitatea poate să fie palpabilă (ceva fizic) sau o idee (un concept) a cărei stare trebuie reprezentată. Obiectul este o unificare între datele şi mulţimea operaţiilor ce pot fi efectuate cu acestea. Utilizatorul unui obiect nu trebuie să cunoască tipurile de date reprezentate în obiect ci doar operaţiile prin care acestea se pot modifica, se zice că reprezentarea internă a datelor este încapsulată (ascunsă) de exterior, dar poate fi manipulată prin operaţii specifice.

Clasa descrie un model sau un şablon de stare şi de comportament pentru obiecte. Definiţia unei clase constă în date (câmpuri) şi metode (proceduri de calcul). Clasa este un tip de dată definit de utilizator pe baza căreia se vor crea noi obiecte din respectiva clasă. Definiţia unei clase constă în:

- **modificatori de acces**: definesc vizibilitatea clasei în raport cu alte clase (`public`);
- `class`: cuvânt cheie care anunţă Java ca urmează un bloc pentru definirea unei clase;
- **câmpuri**: variabile sau constante care sunt folosite de obiectele clasei (`x` şi `y`);
- **constructori**: metode care controlează starea iniţială a oricărui obiect din clasă(`Punct()` şi `Punct(double abscisa, double ordonata)`);
- **metode**: funcţii care controlează valorile stocate în câmpuri (`setX()`, `setY() ...`).

Exemplul următor defineşte o clasă cu numele Punct.

```
//Definitia unei clase
public class Punct {
```

```
//Campuri
    private double x;
    private double y;

//Constructori
    Punct() {
        setX(0);
        setY(0);
    }

    Punct(double abscisa, double ordonata) {
        setX(abscisa);
        setY(ordonata);
    }

// Metode
    public void setX(double abscisa) {
        x = abscisa;
    }

    public void setY(double ordonata) {
        y = ordonata;
    }

    public double x() {
        return x;
    }

    public double y() {
            return y;
    }

    public double distantaOrigine() {
        return Math.sqrt(x*x+y*y);
    }

     public String toString() {
        return "<" + x + "," + y + ">";
    }
}

public class Grafica {
    public static void main(String[] args) {
        Punct p1; //declararea var. obiect p1 de tipul clasa Punct
        Punct p2 = new Punct(-1,7); //decl. + creare + initializare

        p1 =  new Punct(); //creare + initializare obiecte p1

        System.out.println("p1 = " + p1);
        System.out.println("p2 = " + p2);

        p1.setX(12);
        p2.setY(13.345);

        System.out.println("p1 = " + p1.toString());
        System.out.println("p2 = " + p2);

    }
}
```

Rezultate:

```
p1 = <0.0,0.0>
p2 = <-1.0,7.0>
```

```
p1 = <12.0,0.0>
p2 = <-1.0,13.345>
```

3

Operatorii limbajului Java

Tipuri de operatori

În Java avem 5 tipuri de operatori.

- **atribuirea**
- **aritmetici**
- **pe biţi**
- **relaţionali**
- **booleeni**

Variabilele şi constantele se folosesc pentru stocarea datelor la nivelul aplicaţiei. Operatorii sunt caractere speciale prin care Java este anunţat despre operaţia ce trebuie să o facă cu operanzii asupra cărora acţionează. Operatorii au un efect şi întorc un rezultat. Ei combină datele în expresii pentru a produce valori noi.

Operatorul de atribuire
Operatorul de atribuire dă unei variabile o valoarea a unui literal sau o valoarea ce se obţine ca urmare a evaluării unei expresii.

Operatorii aritmetici
Operatorii aritmetici realizează operaţiile aritmetice de bază (adunare, scădere, înmulţire şi împărţire) cu operanzii. Pot fi utilizaţi pentru toate tipurile numerice.

Operatorii pe biţi
Operatorii pe biţi permit interacţiunea cu reprezentarea internă pe biţi ale tipurilor numerice întregi permiţând acţiunea la nivel de bit. Dacă operatorii aritmetici tratează o valoare numerică unitar, cei pe biţi permit modificarea individuală a fiecărui bit din valoarea întreagă.

Operatorii relaţionali
Operatorii relaţionali permit compararea a două valori. Rezultatul comparaţiei este bolean şi poate fi utilizat pentru setarea unor valori sau pentru controlul execuţiei programului.

Operatori booleeni (logici)
Operatori booleeni pot fi utilizaţi numai cu valori boleene sau ca rezultat întotdeauna o valoare boleană.

Atribuirea

Formă: `nume = expresie`

`expresie` **din stânga lui** = **se evaluează apoi valoarea ei se copiază în** `nume`

```
int i, i1=0, i2 = 1;
i1=10;
i2=15;
i1=i2=7;
i=(i1=(i2=7));
```

În limbajul Java orice expresie produce un rezultat, iar tipul rezultatului este determinat de operatorul folosit şi de către tipul operanzilor. Dacă analizăm expresia `a + b`, atunci `a` şi `b` se numesc operanzi, iar `+` operator; efectul operatorului este acela de adunare, iar rezultatul lui este suma valorilor numerice stocate în variabilele `a` şi `b`. Atribuirea, care are simbolul `=`, este un operator, asemenea lui `+` astfel, când undeva în program se scrie `a = b` el va avea un efect şi un rezultat. Efectul este cel de evaluare a expresisei din drepta lui `=`, la noi aceasta este valoarea stocată în `b` şi de copiere a valorii expresiei evaluate în `a`, iar rezultaul este valoarea copiată, adică valoarea lui `b`. Deseori, acest rezultat nu este folosit mai departe, deşi utilizarea lui ar fi corectă.

Majoritatea operatorilor Java produc un rezultat fără a modifica valorile operanzilor. Există însă operatori, asemenea celui de atriburie, care modifică valoarea unui operand. Această acţiune poartă denumirea de efect secundar, în engleză “side effect”. În cazul operatorului = , denumirea este forţată deoarece aici efectul de modificare este cel primar. Java are însă o clasă întreagă de operatori care produc efecte secundare ce vor fi discutaţi în continuare (atribuirea compusă).

Atribuirea poate fi şi multiplă, situaţie în care asociativitatea operatorului de atribuire este de la dreapta la stânga.

Operatorii aritmetici

Realizează operţiile aritmetice de bază;
Operanzii pot fi numerici (literali sau variabile)

```
int a,b,c,d,e;
a = 1 + 2; // + pentru ADUNARE
b = 1 - 2; // - pentru SCĂDERE
c = a * 2; // * pentru ÎNMULȚIRE
d = c / b; // / pentru ÎMPĂRȚIRE
e = a % 2; // % pentru RESTUL ÎMPĂRȚIRII
```

Operanzii unei operaţii aritmetice trebuie să fie numerici, iar rezultatul va fi şi el numeric. Câteva dintre problemele arimeticii simple în Java sunt:

- în expresiile în care participă operatorii aritmetici, ordinea evaluării lor este dată de prioritatea şi asociativitatea lor. `*`, `/` şi `%` au prioritatea mai mare decât `+` şi `-`, motiv pentru care aceste operaţii vor fi evaluate înainte de adunare şi scădere;
- parantezele rotunde modifică prioritatea evaluărilor aritmetice, aceasta începând de la parantezele cele mai interioare;
- împărţirea între întregi întoare rezultat întreg (eventuala parte zecimală este ignorată);
- împărţirea întreagă cu `0` generează o excepţie;
- împărţirea reală cu `0.` întoarce infinit (`Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`) sau rezultat nenumerice (Not A Number `Double.NaN`).

Aritmetica numerelor întregi

Toate operaţiile aritmetice se fac cu tipurile `int` sau `long`, valorile `byte`, `char` sau `short` fiind automat promovate la `int` înainte de efectuarea operaţiilor, rezultatul fiind şi el un `int`. Similar, dacă un operand este de tipul `long`, iar celălat nu, acesta va fi automat promovat la `long`, iar rezultatul va fi `long`. Dacă însă se încercă atribuirea unui rezultat cu reprezentare mai lungă unei variabile ce are un tip cu reprezentare (în octeţi) mai scurtă se primeşte o eroare la compilare. Terminologia folosită în situaţia în care se doreşte stocarea unei valori de un tip într-o variabilă de un alt tip se numeşte forţare de tip (type casting). Forţarea de tip poate produce rezultate ciudate, fie codul:

```
byte b1 = 1, b2 = 2, b3;
b3 = b1+b2; // eroare, rezultatul este de tip int
b3 = (byte) (b1+b2) // aici avem forțare de tip - asa merge, dar pot
                    // apărea ciudățenii
```

dacă, b1 şi b2 iau valoarea 120 fiecare, rezultaul va fi incorect. Forţarea de tip face ca reprezentarea pe biţi a valorii să fie copiată din sursă în destinaţie. Dacă rezultatul nu "încape" în destinaţie se pierd date.

Promovarea și forțarea de tip

Conversie: Trecerea de la o reprezentare internă a unui tip de date la o alta.

Promovare: conversie către un tip cu domeniu mai larg

Forţare: convesrie către un tip cu domeniu mai îngust

Ficare tip de dată are o reprezentare specifică în limbajul Java (de exemplu, numerele reale în virgulă flotantă sunt reprezentate prin M - mantisa, E - exponent şi B - bază, valoarea numărului fiind calculat prin expresia $M*B^E$). Conversia unei variabile sau expresii de la un tip la altul poate conduce la nepotriviri legate de modul de efectuare a calculelor cu respectivele tipuri de date şi de probleme legate spaţiul alocat pentru stocarea rezultatului. Cel mai des compilatorul va recunoaşte pierderea preciziei şi nu va permite compilarea programului, există însă şi cazuri când rezultatul obţinut va fi incorect. Pentru rezolvarea acestei probleme tipurile asociate variabilelor trebuie să fie promovate către tipuri cu un domeniu mai larg sau convertite forţat la tipuri cu domeniu mai mic.

Fie atriburile:

```
int n1 = 102;           //4 octeti pentru stocarea valorii
int n2 = 13;            //4 octeti pentru stocarea valorii
byte n3;                //1 octet pentru stocarea valorii
n3 = (n1 + n2);   // <--- aici compilatorul va da eroare
```

Practic, codul de mai sus ar trebui să funcţioneze deoarece un `byte`, care este un `int` mai mic, este în stare să stocheze valoarea 115. Totuşi, compilatorul nu va face atribuirea, ci da va o eroarea "possible loss of precision" pentru că o valoarea `byte` este mai mică decât una `int`. Rezolvarea problemei se face fie convertind tipul expresiei din dreapta operatorului de atribuire (=) aşa încât să se potrivească cu tipul variabilei din sânga, fie variabila din stânga (`n3`) se declară ca fiind de un tip de mai larg. Soluţia problemei, prin schimbarea tipului lui `n3`, este:

```
int n1 = 102;
int n2 = 13;
int n3;
n3 = (n1 + n2);
```

Promovarea de tip

Există situaţii în care compilatorul modifică tipul unei variabile la un tip care suportă un domeniul de valori mai larg. Această acţiune de conversie poartă denumirea de promovare. Unele promovări se fac automat de către compilator pentru evitarea pierderii preciziei rezultatelor. Situaţiile acestea apar atunci când:

- se atribuie un tip mai mic unui tip mai mare;

- se atribuie un tip întreg unui tip real (deoarece nu există zecimale care să se piardă).

Fie declaraţia:

```
long varsta = 37;
```

Valoarea întreagă (`int`) 37 este atribuită unei variabilei `varsta` de tipul `long`. Promovarea valorii întregi se va face automat înainte de atribuire ei variabilei de tipul `long` deoarece această conversie nu pune probleme.

Înainte de a fi atribuit variabilei, rezultatul unei expresii este plasat într-o locaţie de memorie temporară. Dimensiunea locaţiei va fi întotdeauna egală cu cea a unui `int` sau, în general, cu dimensiunea celui mai mare tip de dată folosit în expresie. De exemplu, dacă expresia înmulţeşte două tipuri `int`, locaţia va avea dimensiunea unui tip `int`, adică 32 de biţi (4 octeţi). Dacă cele două valori care se înmulţesc dau o valoarea care este dincolo de domeniul tipului `int` (de exemplu 77777*777778 = 6,049,339,506 nu poate fi stocată într-un `int`), valoarea va trebui trunchiată pentru ca să încapă în locaţia de memorie temporară. Acest fel de calcul va conduce, în final, la rezultat incorect pentru că variabila care va stoca rezultatul va conţine valoarea trunchiată (indiferent de tipul folosit pentru variabila ce va stoca rezultatul). Pentru rezolvarea acestei probleme, cel puţin una dintre tipurile participante în expresie trebuie să fie `long` pentru a asigura cel mai larg domeniu posibil al variabilei temporare.

Promovările atutomate sunt prezentate în continuare:

```
          char \
                --> int --> long --> float --> double
byte --> short /
```

Observaţi că nu se fac promăvări automate între tipul `boolean` şi orice alt tip de dată.

Forţarea de tip

Forţarea de tip este o conversie ce scade domeniul în care poate lua valori variabila prin modificarea tipului variabilei. Situaţia apare atunci când, de exemplu, o valoarea `long` este convertită la una `int`. Acţiunea se face pentru ca unele metode acceptă numai argumente de un anumit tip sau pentru atribuirea de valori unor variabile cu tip mai mic sau pentru economie de memorie. Sintaxa de forţare a conversiei de tip este:

```
nume = (tip_destinatie) valoare;
```

unde:

- `nume` este numele variabilei la care i se atribuie valoarea;
- `valoare` este valoarea care se atribuie lui `nume;`
- `(tip_destinatie)` este tipul la care se va converti **valoare**. Obesrvaţi că utilizarea parantezelor rotunde este obligatorie.

O soluţie pentru exemplul prezentat prin forţarea conversiei de tip este:

```
int n1 = 102;          //32 de biti pentru stocarea valorii
int n2 = 13;           //32 de biti pentru stocarea valorii
byte n3;               //8 biti de memorie rezervata
n3 = (byte)(n1 + n2); // nu exista pierdere de date
```

Forţarea conversiei de tip trebuie utilizată cu grijă. De exemplu, dacă `n1` şi `n2` ar fi conţinut valori mai mari, forţarea conversiei de tip ar fi trunchiat o parte din date rezultânt o valoarea incorectă. Iată o situaţie de evitat:

```
int i;
long l = 123456789012L;
i = (int) (l); //valoarea numarului este trunchiata
```

La forţarea conversiei de la tipul `float` sau `double`, care au parte fracţionară, la unul întreg, de exemplu, **`int`**, toate zecimalele se pierd. Totuşi, această metodă este utilă atunci când dorin ca dintr-un număr real să facem unul întreg.

Un dintre situaţiile necesare în practică este realizarea împărţirii reale între numere întregi. Codul care urmează exemplifică utilizarea forţării de tip în acest scop:

```
public class fortareConversii {

    public static void main(String args[]) {
        char c = 'a';
        int n = 1111;
        float f = 3.7F, rez;

        c = (char) n;         //fortare de conversie 'barbara'
        rez = f + (float)n / 2; //impartirea este reala

        System.out.println("a, rez: " + c + ", " + rez);
    }
}
```

Rezultate:

```
a, rez: ?, 559.2
```

Promovarea la întregi

Dacă expresia conţine tipuri întregi şi operatori aritmetici (*, /, -, +, %) valorile sunt automat promovate la tipul `int` (sau mai mare dacă este cazul) şi numai după aceea se rezolvă operatorii. Această conversie poate conduce la depăşire sau pierderea preciziei. În exemplul următor primii doi operanzi dintre cei trei cu numele de `a`, `b` şi `c` sunt automat promovaţi de la tipul `short` la tipul `int` înainte de adunare:

```
short a, b, c;
a=1;
b=2;
c=a+b;
```

În ultima linie, valorile lui a şi b sunt convertite la tipul `int`, apoi se adună şi dau un rezultat de tip `int`. Operatorul **`=`** încearcă să apoi să atribuie rezultatul `int` unei variabile `short` (`c`). Această atribuire este însă ilegală şi generează o eroare de compilare. Codul va lucra corect în condiţiile în care:

- `c` se declară de tipul (`int c`);
- se forţează conversia valorii expresiei `a+b` la `short` în linia de atribuire prin: `c=(short)(a+b);`

Promovarea la reali

Asemenea tipurilor întregi care sunt implicit `int`, în unele circumstanţe, valorile atribuite tipurilor reale

sunt implicit de tipul `double`, cu excepţia cazului în care este explicit specificat tipul `float`. Linia următoare va cauza eroare la compilare deoarece se presupune că literalul `23.5` este de tip `double`, iar aceasta nu poate fi "înghesuit" într-o variabilă de tipul `float`.

```
float f1 = 23.5;
```

Pentru ca linia să fie considerată corectă se pot folosi următoarele variante:

- se adaugă un `F` după `23.5` pentru a spune compilatorului că valoarea este `float`: `float f1 = 23.5F;`
- se forţează conversia lui `23.5` la tipul `float` prin: `float f1 = (float)23.5;`

Incrementarea și decrementarea

operatorul ++ creşte cu 1 valoarea operandului
operatorul -- scade cu 1 valoarea operandului
există două forme de scriere:

```
int i = 1, j;
j = ++i; // prefixată: i se incrementează
         // apoi de atribuie valoarea lui i
j = i++; // postfixată: se atribuie
         // valoarea lui i, apoi se
         // incrementează i
```

Una dintre necesităţile curente la nivelul aplicaţiilor este aceea de adunare sau de scădere a lui 1 din valoarea unei variabile. Rezultatul poate fi obţinut sprin folosirea operatorilor + şi - după cum urmează:

```
varsta = varsta + 1;
zileconcediu = zileconcediu - 1;
```

Totuşi, incrementarea sau decrementarea sunt operaţii atât de comune încât există opratori unari (au un singur operand) specifici în acest scop: incrementarea (++) şi decrementarea (--). Ei pot fi scrişi în faţa (pre-increment, pre-decrement) sau după (post-increment, post-decrement) o variabilă. În forma prefixată operaţia (incrementarea sau decrementarea) este realizată înainte de orice alte calcule sau atribuiri. În forma prefixată, operaţia este realizată după toate calculele sau eventualele atribuiri, astfel încât valoarea originală este cea folosită în calcule şi nu cea actualizată. Următorul tabel prezintă operatorii de incrementare şi decrementare:

Operator	Rol	Exemplu	Comentarii
++	pre-increment (**++variabila**)	int i = 5; int j = ++i;	i este 5 şi j este 6.
	post-increment (**variabila++**)	int i = 5; int j = i++;	i este 5 şi j este 5. Valoarea lui i este atribuită lui j înainte de incrementarea lui i. Dim acest motiv valoarea atribuită lui j este 5.
--	pre-decrement (--**variabila**)	int i = 5; int j = --i;	i este 5 şi j este 4.

Operator	Rol	Exemplu	Comentarii
	post-decrement (**`variabila--`**)	int i = 5; int j = i--;	i este 5 şi j este 5. Valoarea lui i este atribuită lui j înainte de decrementarea lui i. Din acest motiv valoarea atribuită lui j este 5.

Iată în comtinuare câteva exemple clasice de utilzare a celor doi operatori:

```
int contor = 10;
int a, b, c, d;

a = contor++;
b = contor;
c = ++contor;
d = contor;
System.out.println(a + " " + b + " " + c +" " + d);
```

Rezultate:

```
10 11 12 12
```

Fie codul:

```
int a, b;
a = 1;
b = ++a * 7;
```

Exisă însă o diferenţă subtilă între forma prefixată şi postfixată. Presupunând că facem o incrementare (**++**), în forma prefixată (`++a`) incrementarea se face prima oară, apoi rezultaul expresiei este chiar operand mai departe în expresie. În cazul liniei de forma:

```
b = a++ * 7;
```

operatorul de incrementare are forma postfixată (`a++`). Aici, valoarea curentă a lui `a` este valoarea expresiei, aceasta va mai fi stocată într-o locaţie temporară, însă această valoare nu va fi folosită în continuarea evaluării expresiei. După evaluarea întregii expresii valoarea din locaţia temporară este crescută cu 1 şi atribuită lui `a`.

În concluzie, incrementarea şi decrementarea sunt operatori unari (acţionează asupra unui singur operand), au efecte secundare, iar rezultaul lor poate fi folosit în continuare în expresii. Rezlultatul poate fi valoarea operandului înainte (la formele post-fixate) sau după (la formele pre-fixate) ce incrementarea sau decremantarea a avut loc.

Operatori relaționali (de comparare)

Întorc rezultate de tipul boolean

>	**mai mare**
>=	**mai mare sau egal**
<	**mai mic**
<=	**mai mic sau egal**
!=	**diferit**
= =	**egal**

```
int i = 1, j=4;
boolean rez;
rez = (i == j); // rez ia valoarea false
rez = (i < j);  // rez ia valoarea true
```

Operatorii relaţionali, unori numiţi şi de comparare a expresiilor, se folosesc la testarea unor condiţii între două expresii şi întorc un rezultat de tipul `boolean`. Sintaxa generală a unei comparaţii este:

rezultat = expresie1 `operatorrelational` *expresie2*

Operator relaţional	Denumire	`true` dacă	`false` dacă
<	Mai mic	*expresie1* **<** *expresie2*	*expresie1* **>=** *expresie2*
<=	Mai mic sau egal	*expresie1* **<=** *expresie2*	*expresie1* **>** *expresie2*
>	Mai mare	*expresie1* **>** *expresie2*	*expresie1* **<=** *expresie2*
>=	Mai mare sau egal	*expresie1* **>=** *expresie2*	*expresie1* **<** *expresie2*

Operator relaţional	Denumire	`true` dacă	`false` dacă
==	Egal	*expresie1* **==** *expresie2*	*expresie1* **!=** *expresie2*
!=	Inegal (Diferit)	*expresie1* **!=** *expresie2*	*expresie1* **==** *expresie2*

Deseori, operatorii relaţionali se folosesc împreună cu cei logici pentru a forma expresii logice complexe. În exemplul următor se testează dacă valoarea stocată în variabila y este în domeniul [x, z] definit prin variabilele x şi z.

```
public class OpRelationali {
    public static void main(String args[]) {
        double x = 11., y = 12., z = 13.;
        boolean indomeniu;

        indomeniu = x <= y && x <= z;
        System.out.println("este " + y + " in domeniul [ " + x +"," +  z + " ]?:
" + indomeniu);

    }
}
```

Rezultatul:

```
este 12.0 in domeniul [ 11.0,13.0 ]?: true
```

Una dintre aplicaţiile specifice ale operatorului **==** este evitarea erorii de împărţire cu zero. Pentru aceasta expresia cu care urmează să se împartă este testată dacă are valoarea 0, dacă este 0 împărţirea este evitată, altfel împărţirea se poate realiza. Una dintre erorile cele mai comune la testarea valorii unei expresii cu constanta 0 este folosirea lui **=** în locul lui **==**.

Operatori logici (booleeni)

Permit formarea de expresii logice pe baza rezultatelor operatorilor relaţionali

cu	fără scurtcircuitare	nume
&&	**&**	**ŞI**
||	|	**SAU INCLUSIV**
^		**SAU EXCLUSIV**
!		**NU**

```
int i=1, j=2, k=3;
boolean rez=true;
rez = (i<j) & (i==1); // false
rez = !rez;  // true
```

Operatorii logici, uneori numiţi şi booleeni, trebuie să aibă operanzi booleeni (adică de tipul `boolean`) şi generează rezultate booleene. Denumirea lor a fost dată în cinstea matematicianului britanic George Boole (1815-1864). El a pus la punct un sistem matematic ce operează valorile de adevăr: **`true`** (adevărat), `flase` (fals) şi funcţiile logice: `AND` (ŞI), `OR` (SAU) şi `NOT` (NU). Funcţiile logice au fost implementate, în Java, sub forma operatorilor logici care au scrierea **`&&`** pentru `AND` , `||` pentru `OR` şi `!` pentru `NOT`. Operatorii logici se definesc prin tabele de adevăr. Acestea reprezintă toate combinaţiile posibile ale operanzilor împreună cu rezultatele corespondente operatorului logic.

Expresiile care conţin operatori logici se evaluează cu scurtcircuitare. Evaluarea se face de la stânga la dreapta (această ordine de evaluare este garantată numai pentru operatorii logici) şi imediat ce valoarea expresiei logice devine cunoscută evaluarea celorlaţi operanzi se termină. && şi || permit utilizarea evaluării cu scurtcircuitare, dacă expresia din stânga operatorului a a determinat deja valoarea întregii expresii logice, expresia din dreapta lui nu se mai evaluează. Dacă în `e1 && e2`, e1 ia valoarea `false`, e2 nu se mai evaluează deoarece oricum rezultatul va fi `false`. Dacă în `e1 || e2`, e1 ia valoarea `true`, e2 nu se mai evaluează deoarece rezultatul este oricum `true`.

Există şi operatori logici care se evaluează fără scurtcircuitare. Ei au o scriere diferită, astfel pentru ŞI se scrie & iar pentru SAU se scrie |. Utilizarea lor se face în situaţii în care scurtcircuitarea poate produce rezultate ciudate, de exemplu fie expresia **`func1() && func2()`**, dacă funcţia **`func1()`** va întoarce rezultatul **`false`** funcţia **`func2()`** nu va mai fi apelată. Vor exista, deci, cazuri în care

`func2()` nu va fi apelată, iar dacă aceasta face ceva semnificativ în cod lipsa apelului va duce la rezultate eronare.

Tabel de adevăr pentru ȘI

&&	false	true
false	false	false
true	fals	true

Tabel de adevăr pentru SAU

||	false	true
false	false	true
true	true	true

Tabel de adevăr pentru SAU EXCLUSIV

^	false	true
false	false	true
true	true	false

Operatori pe biti

Operanzii pot fi de orice tip de întreg şi au ca rezultat o valoarea întreagă

Operează cu decompunerea binară a operanzilor

Operator	Denumire
~	NU unar pe biţi
&	ŞI pe biţi
|	SAU (INCLUSIV) pe biţi
^	SAU EXCLUSIV pe biţi
>>	deplasarea la dreapta
>>>	deplasarea la dreapta cu completare cu 0
<<	deplasare la stânga

În calculatoare, se numeşte bit (**b**inary dig**it**), cea mai mică unitate de informaţie a cărei valoare se poate stoca. Operatorii pe biţi (bitwise) tratează operanzii sub forma unui şir de biţi şi nu sub forma unui singure valori în baza 10. Şirurile se obţine prin transcrierea numerelor din baza 10 în baza 2. Operatorii pe biţi acţionează asupra celui de al N-elea bit al operanzilor folosind o funcţie booleană pentru a genera un rezultat la nivelul aceluiaşi bit N. De exemplu dacă dorim să facem & (ŞI) pe biţi între valoarea 14 şi 7 trebuie să transformăm operanzii în şiruri binare, astfel 14 în baza 10, adică $1\cdot10^1+4\cdot10^0$, devine 1110 în baza 2, adică $1\cdot2^3+1\cdot2^2+1\cdot2^1+0\cdot2^0$, iar 7 în baza 10 devine 0111 în baza 2, apoi între biţii corespunzători aceleiaşi poziţii se face **&** pe biţi după cum se vede în tabelul următor.

Valoarea în baza 10	Valorile bitului N (baza 2)			
	Bit "3" 2^3	Bit "2" 2^2	Bit "1" 2^1	Bit "0" 2^0
$14_{(10)}$	1	1	1	$0_{(2)}$
$7_{(10)}$	0	1	1	$1_{(2)}$

Valoarea în baza 10	Valorile bitului N (baza 2)			
	Bit "3" 2^3	Bit "2" 2^2	Bit "1" 2^1	Bit "0" 2^0
$14_{(10)}$ & $7_{(10)}$	**0**	**1**	**1**	**$0_{(2)}$**

$14_{(10)}$ & $7_{(10)}$ dă, în final, valoarea **$6_{(10)}$**.

Asemenea operatorilor logici şi cei pe biţi se definesc cu ajutorul tabelelor de adevăr.

Tabelele de adevăr pentru operatorii & , | , ^ şi ~:

a	b	a & b	a | b	a ^ b	~ a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

ŞI pe biţi, adică operatorul **&**, produce 1 numai dacă ambii operanzi sunt 1.

SAU pe biţi, adică operatorul **|**, produce 0 doar dacă ambii biţi sunt 0.

SAU EXCLUSIV pe biţi, adică operatorul **^**, produce 1 dacă exact unul dintre operanzi este 1.

NU pe biţi, adică operatorul **~**, inversează toţii biţii operandului. Imversarea însemnă că orice bit 1 trece în 0 şi invers.

Deplasarea la stânga, adică operatorul <<, are forma **`valoare << numar`** şi deplasează la stânga toţi biţii lui **`valoare`** cu un număr de poziţii (puteri ale lui 2) specificat în **`numar`**. Pentru fiecare poziţie deplasată bitul cel mai semnificativ se pierde, iar bitul cel mai puţin semnificativ se completează cu valoarea 0. Deplasarea la stânga cu o poziţie corespunde înmulţirii cu 2 a lui **`valoare`**.

Deplasarea la dreapta, adică operatorul >>, are forma **`valoare >> numar`** şi deplasează la dreapta toţi biţii lui **`valoare`** cu un număr de poziţii (puteri ale lui 2) specificat în **`numar`**. La fiecare deplasarea la dreapta cu o poziţie biţul cel mai puţin semnificativ se pierde, iar noua valoarea care se obţine împărţirea la 2 a valorii iniţiale (restul se pierde). Bitul cel mai semnificativ, care stochează semnul numărului, în urma deplasării la dreapta trece şi el la dreapta, iar poziţia pe care a fost se va completa cu valoarea înaintea de deplasare, adică semnul numărului se păstrează.

Deplasarea la dreapta fără semn, adică operatorul >>>, are forma **`valoare >>> numar`** şi deplasează la dreapta toţi biţii lui **`valoare`** cu un număr de poziţii (puteri ale lui 2) specificat în **`numar`** fără a păstra însă semnul valorii. În locul valorii se semn care corespunde celui mai semnificativ bit se pune valoarea 0.

În Java toate tipurile întregi, cu excepţia lui `char`, sunt cu semn. Aceasta înseamnă că ele pot reprezenta atât valori negative cât şi pozitive. Codificarea numerelor întregi cu semn se face prin

complement faţă de 2, adică numerele negative se reprezintă prin inversarea valorilor biţilor (adică 0 trece în 1 şi 1 trece în 0), după care se adună 1 la rezultat. Presupunând că se lucrează cu valori de tipul `byte` (adică pe 8 biţi), valoarea $-14_{(10)}$ se codifică prin $11110010_{(2)}$. Se pleacă de la 14 în baza 10, care în binar este 00001110, apoi se inversează biţii şi se obţine 11110001, la această valoare se adună 1 şi rezultă 11110010. Pentru a decodifica un număr negativ se vor inversa biţii lui, apoi la valoarea obţinută se adună 1. Pentru $-14_{(10)}$, care în binar este 11110010, trebuie să reţinem semnul "`-`", apoi, după inversarea biţilor se obţine 00001101, adică $+13_{(10)}$, după adunarea lui 1 vom avea 00001110, adică valoarea numerică de +14. Motivul pentru care Java foloseşte această codificare este problema reprezentării lui 0. Valoarea 0, reprezentată prin 0000000, în urma aplicării procedurii de complementare dă 11111111. Aceasta este o valoare negativă a lui 0 care în aritmetica numerelor întregi creează probleme. Pentru evitarea acestora se adună 1 la valoarea obţinută, caz în care se obţine 100000000. Bitul de 1 obţinut însă nu mai poate stocat într-un `byte` deoarece el are spaţiu numai pentru 8 biţi, iar rezultatul are 9 biţi, rezultatul final fiind primii 8 biţi, adică 00000000. Deoarece Java foloseşte complementul faţă de 2 pentru reprezentarea numerelor negative unii operatori pe biţi produc rezultate ciudate. Cel mai semnificativ bit al reprezentării (bitul corespunzători puterii celei mai mari ale lui 2) este numit şi bit de semn deoarece el defineşte semnul valorii numerice (pentru 0 - numărul este pozitiv, iar pentr 1 numărul este negativ). Operaţiile pe biţi care îl modifică sunt cele generatoare de probleme.

```
public class opBiti {
   public static void main(String args[]) {
        int a = 14, b = 4;
        int c;

        c = a & b;
        System.out.println(a + " & " + b + " = " + c);
        System.out.println(binar(a,32) + " & ");
        System.out.println(binar(b,32));
        System.out.println("--------------------------------");
        System.out.println(binar(c,32) + "\n");

        c = a | b;
        System.out.println(a + " | " + b + " = " + c);
        System.out.println(binar(a,32) + " | ");
        System.out.println(binar(b,32));
        System.out.println("--------------------------------");
        System.out.println(binar(c,32) + "\n");

        c = a ^ b;
        System.out.println(a + " ^ " + b + " = " + c);
        System.out.println(binar(a,32) + " ^ ");
        System.out.println(binar(b,32));
        System.out.println("--------------------------------");
        System.out.println(binar(c,32) + "\n");


        c = ~a;
        System.out.println("~" + a + " = " + c);
        System.out.println( binar(a,32) + " ~ ");
        System.out.println("--------------------------------");
        System.out.println( binar(c,32) + "\n");

        c = a << 2;
        System.out.println( a + " << 2 = " + c);
        System.out.println( binar(a,32) + " << 2 ");
        System.out.println("--------------------------------");
        System.out.println(  binar(c,32) + "\n");

        c = a >> 2;
```

```
        System.out.println( a + " >> 2 = " + c);
        System.out.println( binar(a,32) + " >> 2 ");
        System.out.println("--------------------------------");
        System.out.println(  binar(c,32) + "\n");


        c = a >>> 2;
        System.out.println( a + " >>> 2 = " + c);
        System.out.println( binar(a,32) + " >>> 2");
        System.out.println("--------------------------------");
        System.out.println(binar(c,32) + "\n");

        a = -14;
        c = a >> 2;
        System.out.println( a + " >> 2 = " + c);
        System.out.println( binar(a,32) + " >> 2");
        System.out.println("--------------------------------");
        System.out.println(binar(c,32) + "\n");

        c = a >>> 2;
        System.out.println( a + " >>> 2 = " + c);
        System.out.println( binar(a,32) + " >>> 2");
        System.out.println("--------------------------------");
        System.out.println(binar(c,32) + "\n");

    }

    public static String binar(int x, int n)
    {
        int c = 0;
        StringBuffer BufBinar = new StringBuffer("");

        while (x != 0 && c < n)
        {
            BufBinar.append(((x % 2 == 0) ? "0" : "1"));
            x >>>= 1;
            ++c;
        }

         while (c++ < n)
             BufBinar.append("0");

        BufBinar.reverse();
        return BufBinar.toString();
    }
}
```

Rezultate:

```
14 & 4 = 4
00000000000000000000000000001110 &
00000000000000000000000000000100
--------------------------------
00000000000000000000000000000100

14 | 4 = 14
00000000000000000000000000001110 |
00000000000000000000000000000100
--------------------------------
00000000000000000000000000001110

14 ^ 4 = 10
00000000000000000000000000001110 ^
```

```
00000000000000000000000000000100
--------------------------------
00000000000000000000000000001010

~14 = -15
00000000000000000000000000001110 ~
--------------------------------
11111111111111111111111111110001

14 << 2 = 56
00000000000000000000000000001110 << 2
--------------------------------
00000000000000000000000000111000

14 >> 2 = 3
00000000000000000000000000001110 >> 2
--------------------------------
00000000000000000000000000000011

14 >>> 2 = 3
00000000000000000000000000001110 >>> 2
--------------------------------
00000000000000000000000000000011

-14 >> 2 = -4
11111111111111111111111111110010 >> 2
--------------------------------
11111111111111111111111111111100

-14 >>> 2 = 1073741820
11111111111111111111111111110010 >>> 2
--------------------------------
00111111111111111111111111111100
```

Atribuirea compusă

Operatorul de atribuire poate fi combinat cu orice operator aritmetic binar astfel încât în loc de:

(expresie1) = (expresie1) *op* (expresie2)

se poate scrie :

expresie1 *op* = expresie2

```
double total = 0., suma =1., procent=0.5;
total = total + suma;  //1.
total+=suma;           //2.
total*=procent+1.5;    //4.
```

Java dispune de o familie întreagă de operatori ce permit scrierea scurtată a unor forme de expresii. Iniţial, aceşti operatori facilitau compilarea mai eficientă a codului. Fie secvenţa de cod:

```
int a = 1;
int b = 2;
b = b + a; //adunare traditionala
b += a; //adunare compusa
```

În varianta "tradiţională" operatorul de atribuire va face evaluarea expresiei din dreapta lui, rezultatul va fi stocat într-o zonă de memorie temporară, apoi va fi copiat în locaţia stânga egalului. Prin scrierea lui `+=` în locul lui `=` compilatorul va putea evita faza de manipulare prin zona temporară, rezultatul fiind depus direct în `b`. Azi, majoritatea compilatoarelor optimizează deja această procedură ineficientă din Java adică, deşi noi scriem `b = b + a`, compilatorul va genera codul pentru `b += a`. Scrierea poate fi utilizată cu toţi operatorii binari. Toată această familie de operatori de atribuire va produce şi efecte secundare deoarece generează un rezultat dar şi modifică valoarea opreandului din stânga. Conform celor spuse se pot scrie următoatele expresii:

```
b += a; //b = b + a
b -= a; //b = b - a
b *= a; //b = b * a
b /= a; //b = b / a
b %= a; //b = b % a
```

Prioritatea operatorilor Java

Prioritatea determină ordinea de rezolvare a operatorilor

Dacă într-o expresie avem mai mulţi operatori consecutivi de aceeaşi prioritate, atunci se aplică regula asociativitaţii

Utilizarea parantezelor rotunde redefineşte priorităţile

Prioritate	Simbol		Asociativitate
1	**++ – + - ~ ! (tip)**	operatori unari	**de la dreapta la stânga (DS)**
2	*** / %**	înmulţire împărţire rest	**de la stânga la dreapta (SD)**
3	**+ - +**	adunare scădere concatenare	**SD**
4	**<< >> >>>**	deplasări (>>> completare cu 0)	**SD**
5	**< > <= >=instanceof**	relaţionali	**SD**
6	**== !=**	Egalitate	**SD**
7	**&**	ŞI logic / pe biţi	**SD**
8	^	SAU EXCLUSIV logic / pe biţi	**SD**
9	|	SAU logic / pe biţi	**SD**
10	**&&**	ŞI logic	**SD**
11	||	SAU logic	**SD**
12	**?:**	Operatorul condiţional	**DS**
13	**= op=**	operatorii de atribuire	**DS**

În Java orice expresie are un tip (primitiv sau referinţă) determinat în faza de compilare. În cazul unor expresii complexe într-o singură linie de program, Java foloseşte un grup de reguli numite "de

precedenţă" pentru a determina ordinea de rezolvare a operatorilor. Aceste reguli asigură consistenţa operaţiilor aritmetice în cadrul programelor Java. La nivel principial, prelucrarea sau rezolvarea operatorilor se face în următoarea ordine:

- ( ): operatorii din interiorul unor perechi de paranteze; dacă perechi de paranteze sunt cuprinse în alte perechi de paranteze, evaluarea pleacă de la perechea cea mai interioară;
- ++, –: operatorii de incrementare şi decrementare;
- *, /: operatorii de multiplicare (înmulţire) şi diviziune (împărţire), evaluaţi de la stânga la dreapta;
- +, -: operatorii de adunare şi scădere, evaluaţi de la stânga la dreapta.

În tabelul anterior, precedenţa cea mai mare este notată cu 1, iar cea mai scăzută cu 13.

În cazul în care expresiile conţin operatori aritmetici, care apar alăturaţi şi au aceeaşi precedenţă, evaluarea lor se face conform regulilor de asociativitate, respectiv de la stânga la dreapta.

Fie expresia:

```
c = 23 - 6 * 4 / 3 + 12 - 31;
```

Valoarea atribuită lui `c` depinde de ordinea în care vom prelucra operatorii expresiei. De exemplu, dacă prelucrarea se face strict de la sânga la dreapta, fără a ţine cont de precedenţa operatorilor, expresia va avea valoarea `3.6(6)`. Valorea reală însă a expresiei în Java este de `-4`. Pentru a indica modul de aplicare al regulilor de precedenţă în cazul acestei expresii aceasta se va rescrie expresia folosind parantezele:

```
c = 23 - ((6 * 4) / 3) + 12 - 31;
```

Orice expresie este evaluată automat pe baza regulilor de precedenţă. Dacă acestea nu corespund ordinii de evaluare pe care o dorim este obligatorie folosirea prantezelor rotunde. Iată un exemplu în continuare:

```
c = (((23 - 6) * 4) / 3) + 12 - 31;
```

se va evalua astfel:

```
c = ((17 * 4) / 3) + 12 - 31;
c = (68 / 3) + 12 - 31;
c = 22.6(6) + 12 - 31;
c = 34.6(6) - 31;
c = 3.6(6); //3.6(6), unde (6) este perioada
```

Regulile de precedenţă se aplică nu numai în cazul operatorilor aritmetici, ci pentru toţi operatorii Java. Dacă o expresie are mai mulţi operatori consecutivi de aceeaşi precedenţă se va aplica regula asociativităţii pentru determinarea ordinii de evaluare.

```
public class Operatori {
    public static void main(String args[]) {
        int a, b, c, d, e, f, g;
        double da, db, dc, dd, de;
        boolean ba, bb, bc, bd;

        System.out.println("Aritmetica cu int");
```

```
        a=1+1;
        b=a*3;
        c=b/4;
        d=c-a;
        e=-d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);

        System.out.println("\nAritmetica cu double");
        da=1+1;
        db=da*3;
        dc=db/4;
        dd=dc-da;
        de=-dd;
        System.out.println("da = " + da);
        System.out.println("db = " + db);
        System.out.println("dc = " + dc);
        System.out.println("dd = " + dd);
        System.out.println("de = " + de);

        System.out.println("\nAritmetica cu modulo");
        a=42;
        da=42.65;
        System.out.println(a + "%10  = " + a%10);
        System.out.println(da + "%10 = " + da%10);

        System.out.println("\nConversii");
       System.out.println("(int)"+ da + " = "+ (int)da); //fortarea conversiei de
la doubela la int

        System.out.println("\nOperatori logici");
        ba = false;
        bb = true;
        bc = da > db; //? 42.5 > 6 = true
        bd = ba && bc || bc;
        System.out.println(ba + " && " + bb + " || " +bc + " = " + bd);

        /* bd = ba & 1/(e+d) < 3; //NU se face scurtcircuitare pt operandul 2
        * aplicatia va crapa */
        bd = ba && 1/(e+d) < 3; //se face scurtcircuitare pt operandul 2
        System.out.println(ba + " && 1/(" + e + d + ") < 3 = " + bd);



        System.out.println("\nOperatori pe biti");
        a = 3;  // 0011 in binar
        b = 6;  // 0110 in binar
        c = a | b;
        d = a & b;
        e = a ^ b;
        f = (~a & b) | (a & ~b);
        g = ~a & 0x0f;
        System.out.println("a = " +Integer.toBinaryString(a));  //11
        System.out.println("b = " +Integer.toBinaryString(b));  //110
        System.out.println("c = " +Integer.toBinaryString(c));  //111
        System.out.println("d = " +Integer.toBinaryString(d));  //10
        System.out.println("e = " +Integer.toBinaryString(e));  //101
        System.out.println("f = " +Integer.toBinaryString(f));  //101
        System.out.println("g = " +Integer.toBinaryString(g));  //1100
                  System.out.println(Integer.toBinaryString(a)+"  <<  2  ="  +
Integer.toBinaryString(a << 2));

        System.out.println("\nFunctii matematice si constante");
```

```
        System.out.println("sqrt(" + a + ") = " + Math.sqrt(a));    //radical de
ordinul 2
        System.out.println("sin(" + a + ") = " + Math.sin(a));      //sinus
        System.out.println("cos(" + a + ") = " + Math.cos(a));      //cosinus
        System.out.println("tan(" + a + ") = " + Math.tan(a));      //tangenta
        System.out.println("atan(" + a + ") = " + Math.atan(a));    //arctangenta
        System.out.println("exp(" + a + ") = " + Math.exp(a));      //e la a
        System.out.println("log(" + a + ") = " + Math.log(a));      //log natural
        System.out.println("pow(" + a + ",3) = " + Math.pow(a,3));  //a la 3
        System.out.println("PI = " + Math.PI);      //PI
        System.out.println("E = " + Math.E);        //E

    }
}
```

Rezultate:

```
Aritmetica cu int
a = 2
b = 6
c = 1
d = -1
e = 1

Aritmetica cu double
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5

Aritmetica cu modulo
42%10  = 2
42.65%10 = 2.6499999999999986

Conversii
(int)42.65 = 42

Operatori logici
false && true || true = true
false && 1/(1-1) < 3 = false

Operatori pe biti
a = 11
b = 110
c = 111
d = 10
e = 101
f = 101
g = 1100
11 << 2 =1100

Functii matematice si constante
sqrt(3) = 1.7320508075688772
sin(3) = 0.1411200080598672
cos(3) = -0.9899924966004454
tan(3) = -0.1425465430742778
```

```
atan(3) = 1.2490457723982544
exp(3) = 20.085536923187668
log(3) = 1.0986122886681096
pow(3,3) = 27.0
PI = 3.141592653589793
E = 2.718281828459045
```

Şiruri de caractere în Java

Şir de caractere = o secvenţa de caractere de lungime arbitrară

Java NU are implementat un tip primitiv şir de caractere

`String` **clasă Java predefinită în pachetul java.lang.String pentru manipularea şirurilor de caractere**

```
String s; //declaratia unei variabile şir
String sal = ”Salut”; //decl. cu inițial.
```

Operatori: atribuirea (=)
concatenarea (+)
atribuirea compusă (+=).

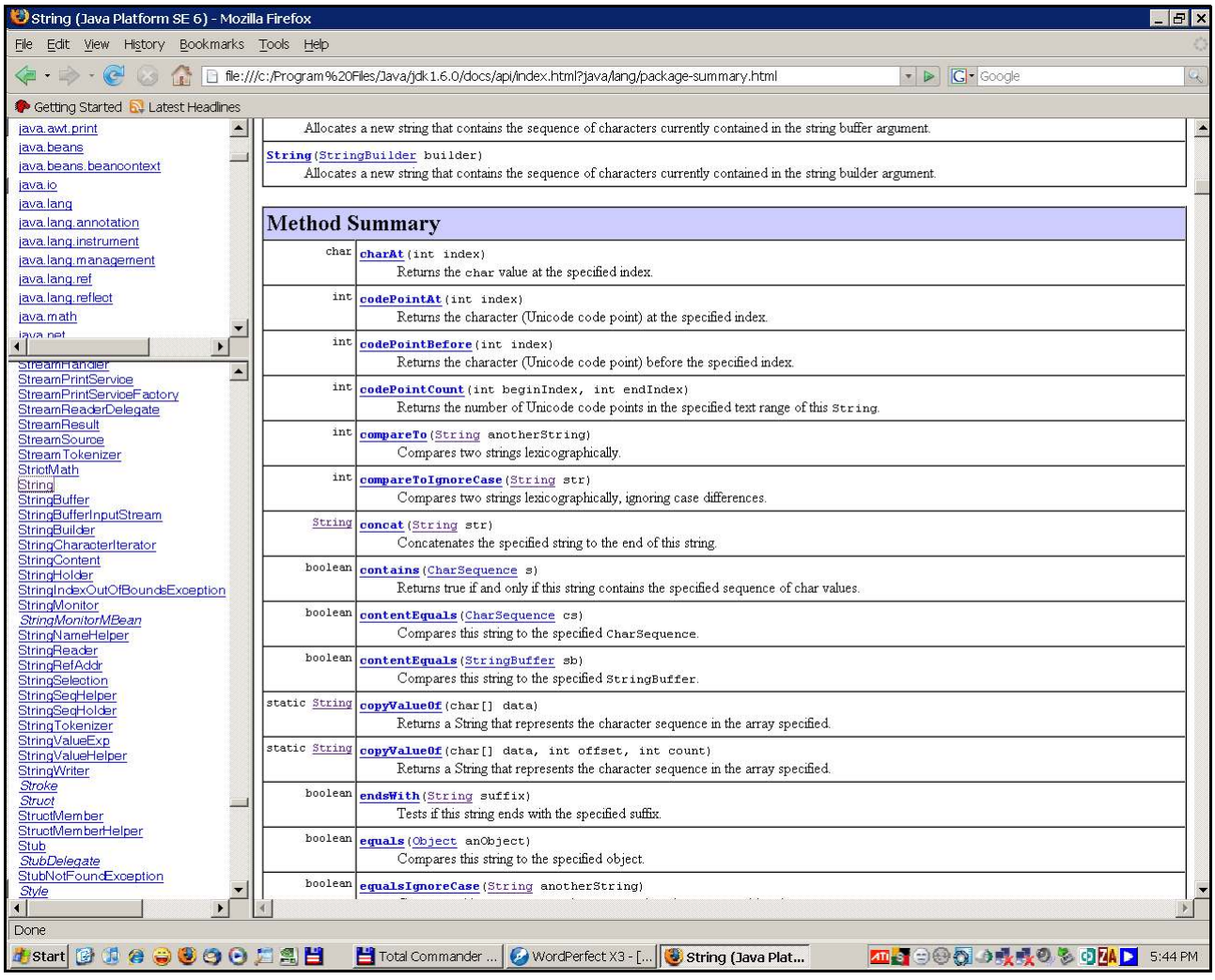
```
public class Siruri {
    public static void main(String[] args) {
        String s1, s2;
        String s3 = "Vasile"; //Vasile este un literal

        s2 ="Ion"; //Ion este un literal

        //concatenare
        s1 = s2 + s2;

        //lungimea unui sir de caractere
       System.out.println("Lungimea sirului: " +s1 +" este de " +s1.length()+"
caractere");

        //subsir
        System.out.println("Un subsir: " + s1.substring(0,3));

        //editarea sirurilor
        s1 = s1.substring(0,3) + "-" + s1.substring(3,s1.length());
        System.out.println("Editarea: " + s1);

        //testarea egalitatii sirurilor
        System.out.println("s1: " + s1+"\ns2: "+s2);
        System.out.println("Egalitatea lui s1 cu s2: " + s1.equals(s2));
        System.out.println("Egalitatea lui Ion cu s2: " + "Ion".equals(s2));

        /* comparatia sirurilor
```

```
         * intoarce < 0 daca s1 vine inainte de s2 in dictionar
         * intoarce 0 daca s1 si s2 sunt egale
         * intoarce > 0 daca s1 este dupa s2 in dictionar
         */
         System.out.println("Compararea lui s1 cu s2: " + s1.compareTo(s2));
    }
}
```

Rezultate:

```
Lungimea sirului: IonIon este de 6 caractere
Un subsir: Ion
Editarea: Ion-Ion
s1: Ion-Ion
s2: Ion
Egalitatea lui s1 cu s2: false
Egalitatea lui Ion cu s2: true
Compararea lui s1 cu s2: 4
```

Clasa `String` are mai bine de 50 de metode. Inspectarea acestora se face cel mai simplu utilizând documentaţia JDK.



Compilatorul Java aloca spaţiu pentru literalii şir în memorie iar operatorul de atribuire va stoca adresa respectivă în variabila şir. Din acest motiv, testarea egalităţii a două şiruri nu se poate face cu operatorul

==, deoarece el va compara adresele la ca sunt stocate cele două şiruri în RAM şi nu conţinutul respectivelor locaţii.

Clasa `String` nu are o metodă pentru modificarea unui caracter al şirului, se zice că obiectele clasei şir sunt imuabile (immutable) - stabile, de neschimbat. Pentru modficarea conţinutului unei variabile şir trebuie să creăm un şir nou.

4

Instrucțiunile limbajului Java

Rularea programelor

Programele sunt alcătuite din instrucţiuni.

Implicit, instrucţiunile se rulează secvenţial.

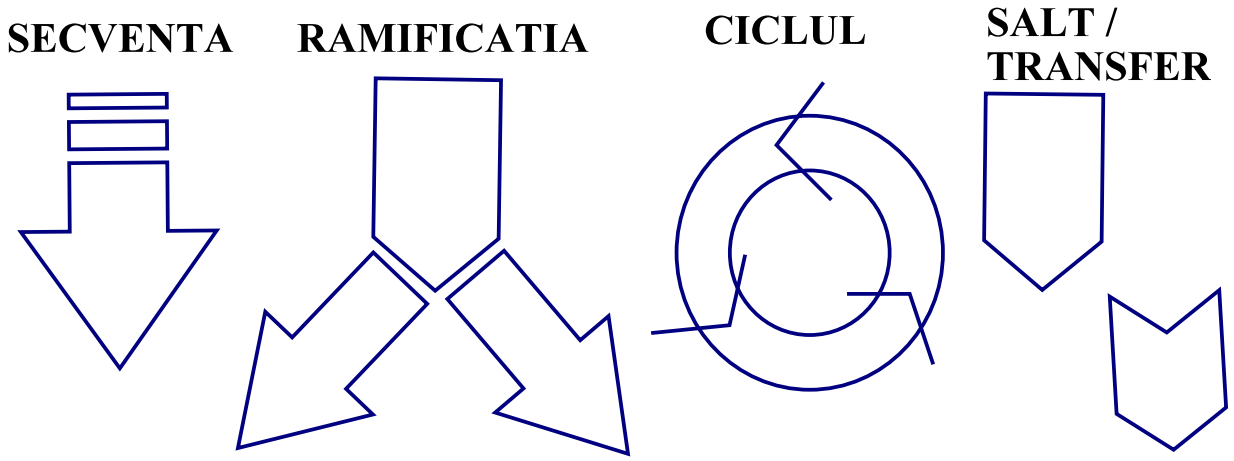
Există situaţii în care se fac devieri de la rularea secvenţială:

- Decizia sau ramificarea: if, switch
- Ciclul: do, for, while
- Saltul sau transferul: break, continue, apel de metodă

O aplicaţie Java este formată din instrucţiuni. Procesul prin care JVM îndeplineşte o instrucţiune se numeşte rulare sau execuţie. În Java terminarea rulării unei instrucţiuni se poate face cu succes, cu excepţie sau cu eroare.

O instrucţiune este formată din una sau mai multe expresii care se rulează ca o singură acţiune. De exemplu `x = 5` este expresie, în timp ce `x = 5;` este deja o instrucţiune. O expresie terminată în caraterul ; se numeşte instrucţiune, iar caracterul ; se numeşte terminator de instrucţiune. Tot instrucţiuni sunt: `x = 5.3*(4.1/Math.cos(0.2*y));` respectiv `System.out.println(x);`. Instrucţiuni sunt inclusiv declaraţiile simple sau multiple, cu sau fără iniţializare. Instrucţiunile vor fi plasate în clase pentru alcătuirea aplicaţiei funcţionale.

Implicit, aplicaţia Java începe din metoda `main`, iar instrucţiunile se rulează secvenţial, adică de sus în jos, în ordinea scrierii lor în aplicaţie pâna la terminarrea rularii tutror instrucţiunilor. În situaţia unei aplicaţii complexe rularea poate devia de la prelucrarea secvenţială astfel, există posibilitatea ramificării la nivelul rulării unor porţiuni de cod pe baza unor condiţii (`if`, `switch`), repetarea unui grup de instrucţiuni atânt timp cât o expresie de control este adevărată (`while`, `do while`, `for`), saltul de la o instrucţine la o alta în program (`break`, `continue`).



SECVENȚA

Secvenţa este alcătuită din:

- instruncţiuni simple
- blocuri numite şi instrucţiuni compuse

Instrucţiune simplă: `expresie;`

Blocul grup de instruncţiuni cuprinse între acolade `{ }`

Instruncţiunea simplă

Instrucţiune simplă este orice exprsie terminată în caracterul ;, iată câteva exemple:

```
int x = 1, y = 2 , z;
z = x+ y;
x=y++;
z++;
```

Instrucţiunea vidă

Instrucţiunea vidă constă într-un **`;`**. Ea nu are efect la nivelul execuţiei instrucţiunilor aplicaţiei. De exemplu, dacă considerăm instrucţiunea simplă **`expresie;`** iar partea de **`expresie`** lipseşte, se obţine o instrucţiune vidă. În practică, se foloseşte pentru a da posibilitatea ca aplicaţia să poată fie completată, de exemplu, cu instrucţiuni în locuri nu eram siguri că va fi nevoie de acestea. Un alt avantaj este cel de evitare al unor erori datorate scrierii "mecanice" a lui **`;`**, fie secvenţa de cod:

```
if (x < 0) { x = -x; };
```

Caracterul ; de după } este legal, însă compilatorul îl consideră o instrucţiune vidă. Un alt motiv al instrucţiunii vide este situaţia în care dorim să reprezentăm, în aplicaţie, starea de "fă mimic".

Instruncţiunea compusă sau blocul

Mai multe instrucţiuni pot fi grupate cu ajutorul acoladelor într-o instrucţiune compusă, numită şi bloc.

```
{
    int x = 1;
    System.our.println(”x = ” + x);
    ++x;
}
```

Instrucţiunea compusă este este sintactic echivalentă cu instrucţiunea simplă. Nu se scrie ; după acolada de închidere.

Orice variabilă declarată într-un bloc are presistenţa limitată la interiorul blocului în care s-a declarat. În afara blocului variabilă respectivă încetează să mai existe.

DECIZIA - if

- `if` se foloseşte pentru programarea unei decizii şi are forma generală:

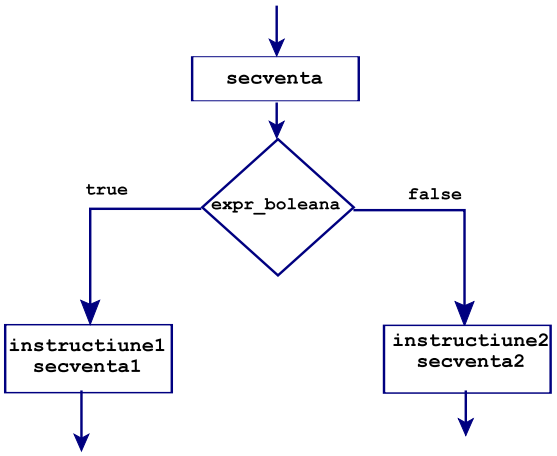
```
if (expr_boleana)
    instructiune1
[else
    instructiune2]
```

- `expr_boleana` trebuie să fie scrisă între paranteze rotunde şi să aibă un rezultat boolean
- dacă este nevoie de mai multe instrucţiuni trebuie folosită instrucţiunea compusă în locul celei simple

Instrucţiunea if

Permite ramificarea rulării în aplicaţie pe baza valorii luate de `expr_boleana`. Expresia boleană determină care dintre ramuri este rulată după cu urmează:

- dacă valoarea expresiei boleene este `true` se rulează instrucţiune simplă `instructiune1`
- dacă valoarea expresiei boleene este `false` se rulează instrucţiunea simplă `instructiune2`



Porţiune cu `else` este opţională, dacă este omisă atunci nu se va rula nimic în situaţia în care `expr_boleana` ia valoarea `false`

Dacă se doreşte ca instrucţunea `if` să aibă efect asupra unui grup de instrucţiuni acestea trebuie puse înt-un bloc. Blocul ne permite ca în locul unei instrucţiuni simple Java să avem voie să scriem un grup de instrucţiuni Java.

Exemplu:

```
if (i>0)
   System.out.println("i -= " + i);
```

```
if (i>0)
{
   System.out.println("i -= " + i);
   i=i%2;
}
```

if-uri imbricate

! apar atunci când una dintre instrucţiunile simple din if este la rândul ei un if

```
if (expr_boleana1)
    if (expr_bolenan2)
        instructiune1
    [else
        instructiune2]
[else
    instructiune3]
```

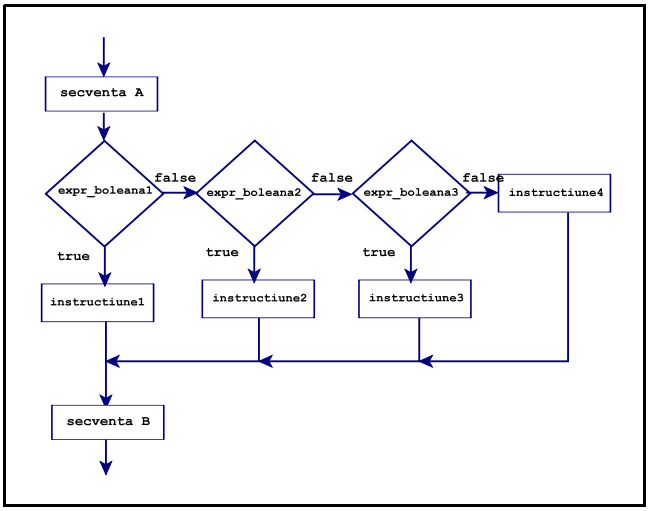
Dacă este cazul instrucţiunile `if` pot fi imbricate (cuprinse una în alta). Deoarece partea de `else` este opţională se poate ajunge la situaţia în care pune problema aparteneţei lui `else`.

```
if (expr_boleana1)
if (expr_bolenan2)
    instructiune1
else
    instructiune3
```

Regula este aceea că `else` se asociază cu cel mai apropiat `if`. Dacă acestă asociere implicită nu este convenabilă atunci se pot folosi acoladele pentru a o modifica.

Când trebuie să luam o singură decizie dintre mai multe avem de a face cu o multidecizie. Aceasta se poate programa cu o secvenţă de if-uri după cum urmează. Dacă `expr_boleanai` este `true` atunci se rulează `instructiunei` (i = 1, 2, 3), altfel se ajunge la `instructiune4`.

```
secventa A
if (expr_boleana1)
    instructiune1
else if (expr_boleana2)
    instructiune2
else if (expr_boleana3)
    instructiune3
else
    instructiune4
secventa B
```



Operatorul condițional ? :

- este o alternativă la `if else`
- este ternar (are 3 operanzi) având forma:

```
expr_boleana ? expr1 : expr2
```

- dacă `expr_boleana` este `true` ia valoarea lui `expr1`, altfel pe a lui `expr2`

Cei trei operanzi formează o expresie condiţională. Primul operand (`expr_boleana`) trebuie să fie o expresie boleană (de exemplu o condiţie), al doilea operand (`expr1`) este valoarea pe care expresia condiţională o întoarce dacă expresia boleană ia valoarea `true`. Al treilea operand (`expr2`) este valoarea expresiei condiţionale dacă expresia boleană ia valoarea `false`.

Fie linia de cod:

```
System.out.println(nota >=5 ? ”Admis.” : ”Respins.”);
```

Dacă valoarea din variabila `nota` este mai mare sau egală cu 5, atunci expresia condiţională ia valoarea `”Admis.”` , atfle ia valoarea `”Respins.”`.

O aplicaţie este determinarea minimului dintre două valori:

```
int a = 5, b =3;
int minim;

minim = (a > b) ? b : a;
System.out.println(”Minimul este: ” + minim);
```

Erori specifice lui if

❶
```
int a = 3, b = 2;
if (a > 0)
   if (b < a)
      System.out.println(”b < a”);
else
   System.out.println(”a < 0”);
```

❷
```
int a = 3;
if (a = 5)
   System.out.println(”a este 5”);
```

❸
```
int a = 3;
if (a%2 == 1) ;
   System.out.println(”a este impar”);
```

Eroarea ❶

Asocierea dintre if-uri şi else este greşită, secvenţa de cod fiind echivalentă cu:

```
if (a > 0) {
  if (b < a)
    System.out.println(”b < a”);
  else
    System.out.println(”a < 0”);
}
```

adică else-ul ţine de al doilea if. Pentru ca acesta să ţină de primul trebuie să folosim acoladele astfel:

```
if (a > 0) {
  if (b < a)
    System.out.println(”b < a”);
  }
else
  System.out.println(”a < 0”);
```

Eroarea ❷

Aici în locul opreratorului de testare a egalităţii (= =) s-a folosit cel de atribuire (=). Java va da eroare la compilare deoarece expresia de testat trebuie să fie boleană.

Eroarea ❸

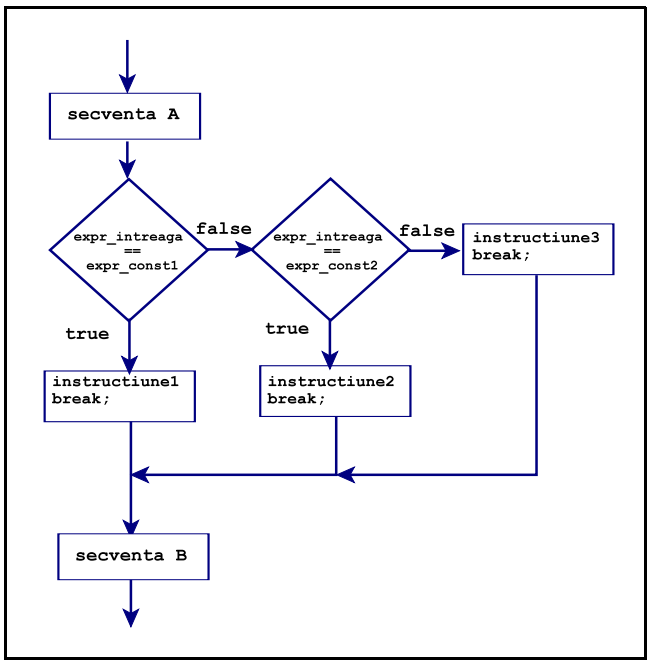
S-a pus caracterul ; după testul din if. Compilatorul nu dă eroare deoarece tratează if-ul având drept corp o instrucţiune vidă.

DECIZIA - switch

! `switch` se foloseşte pentru programarea unei multidecizii, are forma generală:

```
switch (expr_intreaga) {
   case expr_const1:
      instructiune1
      break;
   case expr_const2:
      instructiune2
      break;
   [default:]
      instructiune3
      break;]
}
```

Instrucţiunea de decizie `switch` se utilizează atunci când avem de rulat o secvenţă dintre mai multe posibile. Valoarea expresiei de test (`expr_intreaga`) trebuie să fie de tipurile: `byte`, `char`, `short` sau `int`. Tipurile `long`, `float`, `double` sau alte tipuri de obiecte (`String` etc.) sunt interzise. Valoarea expresiei de test se compară, secvenţial, cu expresiile constante (`expr_consti`) care etichetează instrucţiunile `instructiunei`. Dacă s-a găsit egalitate rularea începe de la instrucţiunea următoare (cea care vine imediat după cele `:` ale lui `case`) şi ţine până la întâlnirea unui `break`, moment în care rularea se conţinuă cu `secventa B`. Dacă nu se găseşte nici o egalitate rularea va continua cu instrucţiunile următoare etichetei `default`. Eticheta `default` este opţională, dacă aceasta lipseşte switch-ul nu va face nimic în lipsa găsirii unei egalităţi.



Etichetele trebuie să fie expresii constante (au valori cunoscute în momentul compilării). În acest scop se pot folosi literali sau variabile `final`.

```
import java.util.*;
public class Switch {
    public static void main(String[] args) {
     Scanner intrare = new Scanner(System.in);
     System.out.print("Selecteaza optiunea (1,2,...,7): ");
     int optiune = intrare.nextInt();

     switch (optiune) {
         case 1:
         case 2:
         case 3:
         case 4:
         case 5:
             System.out.println("La munca cu tine");
             break;
         case 6:
             System.out.println("La chef cu tine");
             break;
         case 7:
             System.out.println("Relaxeaza-te sau revino-ti!");
             break;
         default:
            System.out.println("Aici, pe Terra, avem numai 7 zile!");
             break;
     } //terminare switch

    } //terminare main

} // terminare clasa Switch
```

Rezultate:

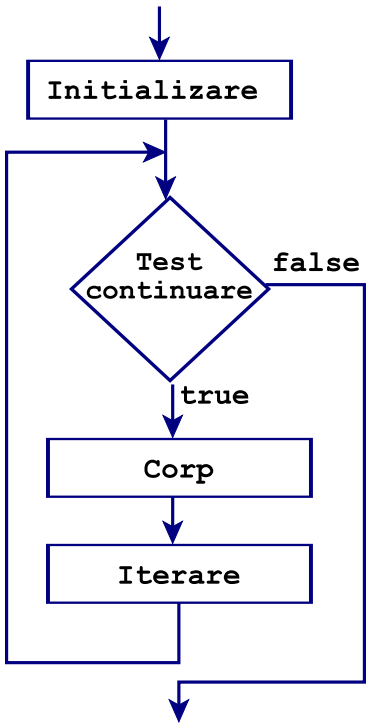
```
Selecteaza optiunea (1,2,...,7): 5
La munca cu tine
```

CICLAREA în Java

- 3 instrucţiuni de ciclare:
  - `while`
  - `do while`
  - `for`
- toate ciclurile au 4 porţiuni:
  - iniţializarea
  - test de continuare
  - corpul
  - iteraţia

Denumirea de ciclare sau iteraţie se foloseşte pentru a descrie rularea repetată a unui grup (bloc) de instrucţiuni până când o anumită condiţie ajunge să fie îndeplinită. Ciclarea este caracterizată printr-o mulţime de condiţii iniţiale, o condiţe de terminare şi un pas de iterare.

Iniţializarea
Constă în atribuirea de valori iniţiale unor variabile ce sunt modificate prin iterare şi testate pentru continuarea corpului ciclului.



Teste de continuarea
Este o valoarea de tipul `boolean` ce determină reluarea rulării instrucţinilor din corpul ciclului. Dacă expresia de terminare se evaluează la valoarea `true` corpul se reia, dacă este `false` reluarea se termină şi se continuă cu secvenţa următoare ciclului.

Corpul
Este o singură instrucţiune simplă sau una compusă care va fi reluată până la îndeplinirea condiţiilor de terminare respectiv atunci când expresia de test ia valoarea `false`.

Iterarea
Reprezintă codul ce se rulează după rularea corpului dar înainte de realizarea testuui de realuare a ciclului. Se foloseşte pentru controlul rulării instrucţiunii de ciclare. Trebuie să realizeze avansul către condiţia de terminare a ciclului.

CICLUL while

- cel mai simplu ciclul Java, are forma:

```
while (conditie) instructiune
```

- **conditie** se scrie între paranteze
- corpul este format dintr-o **instructiune** simplă sau un bloc

```
int i = 5; //initializare
while (i >= 0) //test
   System.out.println("i: " + i--);

while (i < 5) {
   System.out.println("i: " + i);
   ++i;
}
```

```
i:5
i:4
i:3
i:2
i:1
i:0
i:-1
i:0
i:1
i:2
i:3
i:4
```

Ciclul `while` reia rularea lui `instructiune` atât timp cât `conditie` are valorea `true`. Corpul lui `while` nu se va rula dacă `conditie` este `false`. Dacă `conditie` are valoarea `false` de la început, corpul ciclului nu se va rula.

Aplicația 1 - while

Cunoscându-se valoarea depunerii iniţiale şi rata anuală se cere să se determine numărul de luni necesar pentru a cumula în cont o sumă impusă de bani. Dobânda se calculează lunar şi se cumulează (se adună) cu valoarea din luna anterioară din cont.

Spaţiul datelor
Estre mulţimea variabilelor foloşite pentru reprezentarea şi soluţionarea cerinţei impuse.

Numele variabilei	Semnificaţie	Tipul	Valoarea iniţială
`depunere`	valoarea depunerii iniţiale	`double`	dată de intrare
`dobandaan`	valoarea dobânazii anuale	`double`	data de intrare
`scop`	valoarea ce se doreşte a fi atinsă	`double`	dată de intrare
`cont`	valoarea curentă în cont	`double`	`depunere`
`luni`	rezultat: numărul de luni necesar atingeriui scopului	`int`	0

Spaţiul transformărilor
Este dat de mulţimea operaţiunilor efectuate asupra spaţiunlui de date pentru găsirea soluţiei.

Datele de intrare se modifică ca urmare a citirii valorilor corepunzătoare de la tastatură.

Calculul dobânzii pe lună:
dobânda pe lună = dobandaan / 100 / 12

Se împarte cu 100 deoarece dobânda pe an se dă în procente şi cu 12 deoarece anul are 12 luni. Dacă suma existenă în cont se înmulţeşte cu această valoare se obţine câştigul lunar pe suma totală din cont.

Cumularea lunară a dobânzii:
cont = cont + cont * dobânda pe lună

Valoarea nouă a contului va fi cea veche la care se mai adaugă câştigul ca urmăre a dobânzii lunare.

Actualizarea lunii
luni = luni + 1

Variabila se actualizrează prin creştere cu o unitate până când scopul nu s-a atins. Atingerea scopului duce la afişarea valorii acestei variabile şi la terminarea aplicaţiei

Descrierea în “cuvinte” a soluţiei:

Citeşte scop
Citeşte depunere

Citeşte dobandaan in %

// iniţializări
cont = depunere;
luni = 0;

// actualizarea contului pe luna
// pana cand scopul este atins
Repetă cât timp (cont <= scop) {
dobandalunara = cont * dobandaan / 100 / 12; //calcul dobanda lunara
cont = cont + dobandalunara; //cumulare dabanda lunara
luni = luni + 1 //crestere luna
}

//afisare rezultat(e)
Afisare luni

Codul java:

```
import java.util.*;
public class Dobanda {
    public static void main(String[] args) {
        // citirea datelor de intrare
        Scanner in = new Scanner(System.in);
        System.out.print("De ce suma ai nevoie? ");
        double scop = in.nextDouble();
        System.out.print("Care e suma initiala? ");
        double depunere = in.nextDouble();
        System.out.print("Dobanda anuala %: ");
        double dobandaan = in.nextDouble();

        // initializare
        double cont = depunere;
        int luni = 0;

        // actualizarea contului pe luna
        // pana cand scopul este atins
        while (cont <= scop) {
            // dobanda pe luna se calculeaza din cont
            double dobandalunara = cont * dobandaan / 100 / 12;
            cont += dobandalunara;
            luni++;
        }

        // afisarea rezultatului
        System.out.println("Ajungi la suma dorita in " + luni + " luni.");
    }
}
```

Rezultate:

```
De ce suma ai nevoie? 10000
Care e suma initiala? 7800
Dobanda anuala %: 5
Ajungi la suma dorita in 60 luni.
```

Aplicația 2 - while

Să se calculeze şi să se afişeze media aritmetică a unei clase formate din 5 elevi. Media fiecărui elev este un număr întreg care se citeşte de la tastatură.

Spaţiul datelor

Numele variabilei	Semnificaţie	Tipul	Valoarea iniţială
`notaElev`	media unui elev	`int`	data de intrare obţinută prin conversie
`contorElev`	numără elevii pentru care s-au citi mediile	`int`	1
`total`	suma mediilor elevilor citite de la tastatură	`int`	0
`mediaClasa`	rezultat: media aritmetică a clasei	`double`	
`sirNota`	nota introdusă de la tastatură sub formă unui şir de caractere	`String`	data de intrare

Spaţiul transformărilor

Totalizarea mediilor individuale

total = total + notaElev

Suma mediilor individuale presupune o adunare repetată ce se termină atunci când s-au prelucrat toti elevii. Această sumă se realizează pe măsură ce se citeşte câte o medie de elev. Valoarea nouă a totalului este cea anterioară (veche) la care se adună ultima medie citită.

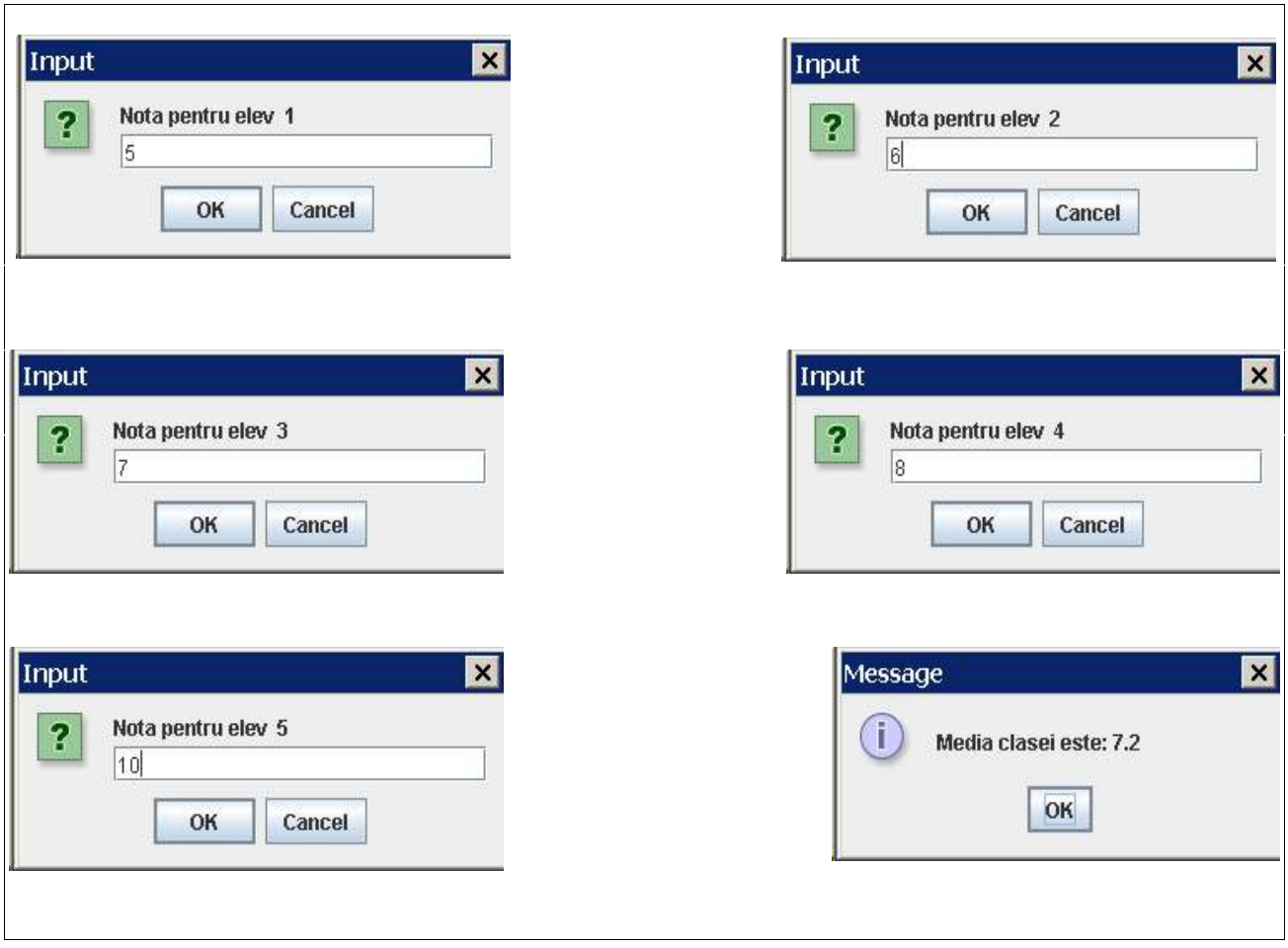
Descrierea în “cuvinte” a soluţiei:

```
//initializari
total = 0;
contorElev = 1;

//totalul mediilor individuale
Repetă cât timp (contorElev <= 5) {
        Citeste notaElev
        total = total + notaElev   //insumeaza media individuala citita
        contorElev = contorElev +1 //creste contor elev
      }

//calcul medie clasa
mediaClasa = total / 5.

//afisare rezultat
Afisare  mediaClasa)
```

Codul Java:

```
import javax.swing.JOptionPane;

public class Media {
    public static void main(String[] args) {
      int total;
      int contorElev;
      int notaElev;
      double mediaClasa;
      String sirNota;

      //initializari
      total = 0;
      contorElev = 1;
      //totalul mediilor individuale
      while (contorElev <= 5) {
         sirNota = JOptionPane.showInputDialog("Nota pentru elev  " + contorElev);
        notaElev = Integer.parseInt(sirNota);
        // sunt scrieri alternative ale instructiunilor
        total = total + notaElev;   //total+=notaElev;
        contorElev= contorElev +1;  //++contorElev;
      }
      //calcul medie clasa
      mediaClasa = total / 5.;
      //afisare rezultat
      JOptionPane.showMessageDialog(null, "Media clasei este: " + mediaClasa);
    }
}
```

Rezultate:



CICLUL do while

! are plasată condiţia de realuare a ciclului la capăt:

```
do
    instructiune
while (conditie);
```

! **conditie** se scrie între paranteze

! corpul este format dintr-o **instructiune** simplă sau un bloc

```
int i = 5;
do {
    System.out.println("i = " + i);
    i--;
} while (i > 0);
```

```
i=5
i=4
i=3
i=2
i=1
```

Ciclul `do while` lucrează după acelaşi pricipiu cu ciclul `while` cu excepţia că expresia de reluare a ciclului `conditie` se evaluează după ce corpul a fost rulat o dată. Instrucţiunea se utilizează atunci când corpul ciclului trebuie rulat cel puţin o singură dată.

Aplicația 1 - do while

Acesta este o implementare folosind ciclul `do while` a aplicaţie 1 de la ciclul `while`.

```
import java.util.Scanner;

public class DobandaV1 {
    public static void main(String[] args) {
     // citirea datelor de intrare
     double scop, depunere, cont, dobandaan, dobandalunara;
     int luni;
     String raspuns;

     Scanner in = new Scanner(System.in);
     System.out.print("De ce suma ai nevoie? ");
     scop = in.nextDouble();
     System.out.print("Care e suma initiala? ");
     depunere = in.nextDouble();
     System.out.print("Dobanda anuala %: ");
     dobandaan = in.nextDouble();

     // initializarei
     cont = depunere;
     luni = 0;

     // actualizarea contului pe luna
     // pana cand scopul este atins
     do {
       // dobanda pe luna se calculeaza din cont
       dobandalunara = cont * dobandaan / 100 / 12;
       cont += dobandalunara;
       luni++;
       System.out.println("Dupa  " + luni + " luni ai acumulat " + cont + " RON");
       System.out.print("Iti ajunge suma acumulata? (D/N): ");
       raspuns = in.next();
      } while (raspuns.equals("D")) ;
    }
}
```

Rezultate:

```
De ce suma ai nevoie? 1000
Care e suma initiala? 970
Dobanda anuala %: 12
Dupa  1 luni ai acumulat 979.7 RON
Iti ajunge suma acumulata? (D/N): D
Dupa  2 luni ai acumulat 989.4970000000001 RON
Iti ajunge suma acumulata? (D/N): D
Dupa  3 luni ai acumulat 999.39197 RON
Iti ajunge suma acumulata? (D/N): D
Dupa  4 luni ai acumulat 1009.3858897 RON
Iti ajunge suma acumulata? (D/N): N
Process exited with exit code 0.
```

CICLUL for

! realizează gruparea celor 4 porţiuni tipice ale unui ciclu:

```
for(initializare;conditie;iteratie)
    instructiune
```

```
for (int i = 0; i < 5; ++i)
    System.out.println(i);
```

```
i=0
i=1
i=2
i=3
i=4
```

Semantica lui for poate fi cuprinsă în următorii paşi: se evaluează `iniţializare`, aceasta se face o singură dată la intrarea în ciclu (nu şi la reluarea acestuia); se trece, apoi, la evaluatea lui `conditie`, această condiţie de reluarea a ciclului este indentică cu cea de la `while`, ea se evaluează la fiecare reluare, inclusiv după iniţalizarea lui; dacă `conditie` la evaluare dă `true` se trece la rularea lui `instructiune` după terminarea execuţiei corpului se trece la evaluarea lui `iteratie` din acest moment reluarea se face de la evaluarea lui `conditie` după scenariul deja descris. Partea de iniţializare poate conţine şi declaraţii, din acest punct de vedere următoarele cicluri sunt echivalente ca şi rezultate:

```
int i;
for (i = 0; i < 5; ++i)
     System.out.println(i);
```

```
for (int i = 0; i < 5; ++i)
     System.out.println(i);
```

Vizibilitatea variabilelor declarate în `iniţializare` se termină odată cu terminarea ciclului `for`.

Componentele `iniţializare` , `conditie` şi `iteratie` sunt opţionale, dar caracterele `;` care le separă sunt obligatorii. `iniţializare` şi `iteratie` pot să conţină o listă de expresii separate prin virgulă care se evaluează de la stânga la dreapta. Valorile acestor expresii se pierd dacă nu sunt stocate în variabile. `conditie` trebuie să fie de tipul logic, iar dacă este absentă se consideră că are valoarea `true`. În baza celor scrise, câteva forma valide `for` sunt:

```
//① - aceasta secventa de cod
// realizeaza calculul factorialului de la 1 la 7
int i, j;
for (i=1, j=i; i < 8; ++i, j*=i)
     System.out.println(i + "! = " + j);

//② - afiseaza cifrele de la 0 la 4
int i = 0;
```

```
for ( ; i < 5; ++i)
     System.out.println(i);

//③ - acelasi efect cu secventa de cod ②
int i = 0;
for ( ; i < 5;)
     System.out.println(i++);

//④ - ciclu for infinit
for ( ; ;) {
   . . .
}
```

În primul exemplu (①) iniţializarea conţine două expresii prin care `i` ia valoarea `1` şi `j` valoarea lui `i`, iar iteraţia (avansul) constă în incrementarea lui `i` şi modificarea lui `j` la produsul dintre valoarea lui curentă şi ce a lui `i`.

În exemplul al doilea (②) expresia de iniţializare a fost omisă şi mutată înainte de începerea ciclului `for`. Observaţi că prezenţa lui `;` este obligatorie.

În cel de al treilea (③) exemplu este omisă şi expresia de iteraţie, dar ea a fost inclusă în corpul `for`-ului. Observaţi că, şi în acest caz, prezenţa lui `;` este obligatorie.

În cel de al patrulea (④) exemplu toate cele trei expresii au fost omise. Lipsa lui `conditie` fiind interpretată ca şi `true` rezultatul este un ciclu infinit. Crearea unor astfel de cicluri ar fi absurdă dacă nu ar exista o posibilitate de a le părăsi (instrucţiunea `break` este cea care permite ieşirea forţată unui ciclu şi va fi prezentată în continuare).

Erori specifice ciclurlor

❶
```
int i = 3;
while (i < 7);
   System.out.println(i--);
System.out.println(”gata.”);
```

❷
```
int i = 3;
while (i < 7)
   System.out.println(i);
   --i;
```

❸
```
int suma = 0;
for( ;i<10; suma+=i++);
   System.out.println(i);
```

Eroarea ❶
Ciclul `while` are un ; în plus după condiţia de testat pentru reluarea ciclului fiind echivelant cu:

```
. . .
while (i <7)
      ;
. . .
```

Java nu dă eroare la compilare considerând că are de a face cu o instrucţiune vidă.

Eroarea ❷
`while` nu are, asemenea lui for, `interaţia` (expresia de avans către terminarea ciclului) inclusă în corpul ciclului. Expresia `--i` nu face pare din corpul `while`-ului deoarece nu s-au folosit acolade, respectiv valoarea lui `i` rămâne neschimbată. Se obţine un ciclu `while` infinit deorece valoarea expresiei ce ar trebuie să asigure părăsirea ciclului rămâne pe veci neschimbată.

Eroarea ❸
Aici valorea lui `i` nu s-a iniţializat. Iteraţia este continuată cât timp `i` este mai mic ca 10. Caracterul `;` face ca instrucţiunea vidă să fie corpul ciclului `for`. Expresia de iterare adună pe `i` la `suma` şi apoi incrementează pe `i`.

Saltul cu break

- realizează părăsirea unui switch sau ciclu
- părăsirea se face transferându-se rularea la prima instrucţiune următoare;
- permite ieşire prematură, deci şi forţată din cicluri

```
while (true) {
  System.out.println(i);
    if (i >= 20) //iesire pt. i>=20
       break;
       ++i;
}
. . .
```

Instrucţiunile de ciclare prezentate (`while`, `do while`, `for`) realizau testul pentru reluare sau terminare la începutul sau la sfârşitul ciclurilor. Uneori, prin natura problemei de rezolvat, acest test trebuie realizat undeva prin mijlocul corpului ciclului sau trebuie realizat în mai multe puncte ale corpului ciclului. Java furnizează o metodă generală de părăsire a unui ciclu prin instrucţiunea `break`. Sintaxa instrucţiunii este:

```
break;
```

Părăsirea ciclului, prin combinarea lui `if` cu `break`, se va putea face atunci când o anumită condiţie este îndeplinită. Controlul va fi predat imediat instrucţiunii care care urmează, secvenţial, ciclului.

Există grupuri de programatori care evită folosirea lui `break` deoarece instrucţiunea nu este stric necesară la scrierea aplicaţiilor. Aplicaţia anterioară ar putea fi modificată asttfel:

```
while (i <= 20) {
  . . .
}
```

pentru evitarea scrierii lui `break`.

O instrucţiune `break` termină instrucţiunea de ciclare în corpul căreia este scrisă. Există însă posibilitatea realizării unor cicluri imbricate (un ciclu este în corpul altui ciclu). Utilizarea unui `break` în această situaţie va duce la terminarea unui singur ciclu, şi anume, a celui din corpul căruia face parte `break`-ul. Pentru părăsirea mai multor cicluri se poate folosi instrcţiunea `break` etichetată care va fi prezentată imediat.

Saltul cu continue

- se poate utiliza numai în cicluri
- abandonează interaţia curentă şi transferă controlul la sfârşitul corpului ciclului

```
for(int an = 1990; an <= 2020; ++an) {
  if ((an % 100 == 0) && (an % 400 != 0))
    continue; //saltul se face la ++an
  if (an % 4 == 0)
    System.out.println(an);
}
```

În cadrul ciclurilor, atunci când corpul acestora are mai multe instrucţiuni, există situaţii în care un grup de instrucţiuni se doreşte a fi sărit, fără însă a părăsi ciclurile. În acest scop Java pune la dispoziţia programatorului instrucţiunea `continue`. Aceasta poate fi folosită numai în cicluri şi are ca efect continuarea execuţiei unui ciclu începând cu etapa următoare execuţiei complete a corpului acestuia (expresia de test pentru reluare corpului la `while` şi `do while` sau interaţie la `for`). Iteraţia curentă este abandonată şi se trece la interaţia următoare. Secvenţa de cod prezentată permite determinarea anilor bisecţi. Iteraţia în cazul în care anul este divizibil cu 100 dar nu este cu 400 este sărită folosind `continue` după cu se vede mai sus.

break și continue etichetate

- se utilizează în cazul ciclurilor imbricate
- permit saltul în afara ciclului curent
- eticheta este un nume dat instrucţiunii în care se va realiza saltul

```
afara1:
for (j=0; j<=3; ++j) {
  afara2:
  for (k=0; k<=3; ++k) {
    if (j+k <= 1) {
      System.out.println(j + "   " + k + " > break");
      break afara2;
    }
    if (j+k >= 5) {
      System.out.println(j + "   " + k +" > continue");
      continue afara1;
    }
    if (k == 0)
      continue afara2;
    System.out.println(j + " / " + k + " = " + (double)(j/k));
  } //for(k= ...
}//for(j=...
```

Posibilitatea scrierii unor cicluri imbricate pune problema ieşirii definitive dintr-o astfel de situaţie sau de continuare a ciclului în care ciclul curent este imbricat. `break` şi `continue` pot fi utilizate cu etichete. Acestea pot marca punctele de părăsire sau de reluare la nivelul ciclurilor. În cazul lui `break` saltul se va face la prima instrucţiune următoare etichetei. Pentru `continue` saltul se face la următoarea iteraţie a ciclului marcat prin etichetă.

Rezultate:

```
0   0 > break
1   0 > break
2 / 1 = 2.0
2 / 2 = 1.0
2   3 > continue
3 / 1 = 3.0
3   2 > continue
```

Linia `break afara2;` produce saltul la instrucţiunea marcată cu `afara2:`. Acesta este ciclul `for (k=0; k<=3; ++k) ...`. Pentru acest `break` valoarea lui `k` pleacă întotdeauna de la 0. Linia `continue afara1;` facă ca linia de afişare a împărţirii să fie omisă, iar continuarea se face cu un `j` actualizat. Deşi `break` şi `continue` sunt utile în cazul ciclurilor imbricate, folosirea lor este de evitat prin regândirea logicii ciclurilor în care participă. Folosirea intensivă a acestora va conduce la un cod neclar şi greu de depanat.

Aplicația 1 - for

Să se scrie o aplicaţie pentru calculul lui n! = 1*2*3* ... *n, cu 1! = 1.

```
import java.util.*;
public class factorial {
    public static void main(String[] args) {
        Scanner in =  new Scanner(System.in);

        System.out.print("n: ");
        int n = in.nextInt();

        long fact = 1;
        for(int i = 1; i<=n ; ++i)
            fact=fact*i; //sau fact*=i;

        System.out.println(n+"! = "+fact);
    }
}
```

Rezultate:

```
n: 5
5! = 120
n: 555
555! = 0
```

Observaţi că valoarea lui `555!` nu se mai poate reprezenta folosind tipul `long`. Dacă precizia sau domeniul de valori a tipurilor întregi sau reale nu sunt suficiente în Java se pot folosi clasele `BigInteger` sau `BigDecimal` din pachetul `java.math`. Aceste sunt create pentru manipularea unor valori numerice arbitrar de mari. Aplicaţia anterioară a fost rescrisă folosind în locul lui `long` pe `BigInteger`. Aici, crearea unei valori de tipul `BigInteger` se face cu `BigInteger.valueOf(valoare)`. Operatorii matematici sunt înlocuiţi cu metode, pentru * se va folosi `multiply`.

```
import java.math.BigInteger;
import java.util.*;

public class factorialB {
    public static void main(String[] args) {
        Scanner in =  new Scanner(System.in);

        System.out.print("n: ");
        int n = in.nextInt();

        BigInteger fact = BigInteger.valueOf(1);
        for(int i=1; i<=n ; ++i)
            fact = fact.multiply(BigInteger.valueOf(i)); //fact = fact * i;

        System.out.println(n + "! = " + fact);

    }
}
```

Rezultate:

```
n: 555
555! =
661408560927794670909833167124276990212353194561078966630610091508066518398462938708570165931453818774346806677937487622941296716409901122180791183381615199180133649323135568584492485536333258769584469786383591661922104266566863913614070698
```

```
1388815455308085223461560550531157622626126794762564813226882035671711110382549
6285768948868390683387427561794062346854491689633073215348773710363218016157511
8186305792613457707073122170130115259282176086845492519990350538601778719955400
6953007367145481629866478860197713791440756421726194493558859063114909315620185
9832173006150698910081357711177369686310362939324425024584999311539904643730800
8914727291891591177025127637515245902602746246400206381390239568453765537479100
2706998231913706076316552578696345155065900890139743142693816783198887138924073
5906053693865079154285101747723299382026182512365914527438847783156831674629869
3321947504594772835660860407072517172711559986446972230134870005688809278734282
6891132360146797709297008349134757097268075117261106076588747857118235528967700
8837953463376048502815279955957922924689302538415337162205637471098765281762231
1757186764471193697842626560000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000
```

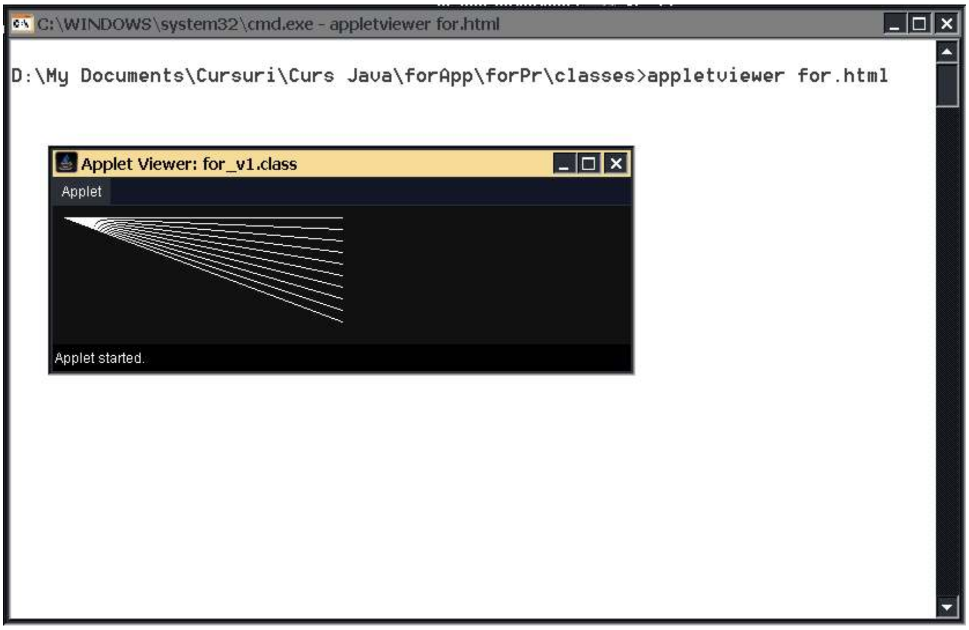
Aplicația 2 - for

Aplicaţia constă într-un applet (o aplicaţie ce se rulează în navigator). Codul Java al aplicaţiei.

```
import java.awt.Graphics;
import javax.swing.JApplet;

public class for_v1 extends JApplet {
   public void paint( Graphics g )
   {
       // apel metoda paint mostenita din JApplet
      super.paint( g );

      for ( int c = 1; cr <= 10; c++ )
         g.drawLine( 10, 10, 250, c * 10 );
   } // terminare paint
} // terminare clasa for_v1
```

Codul HTML al aplicaţiei ce va fi stocat în fişierul text `for.html`:

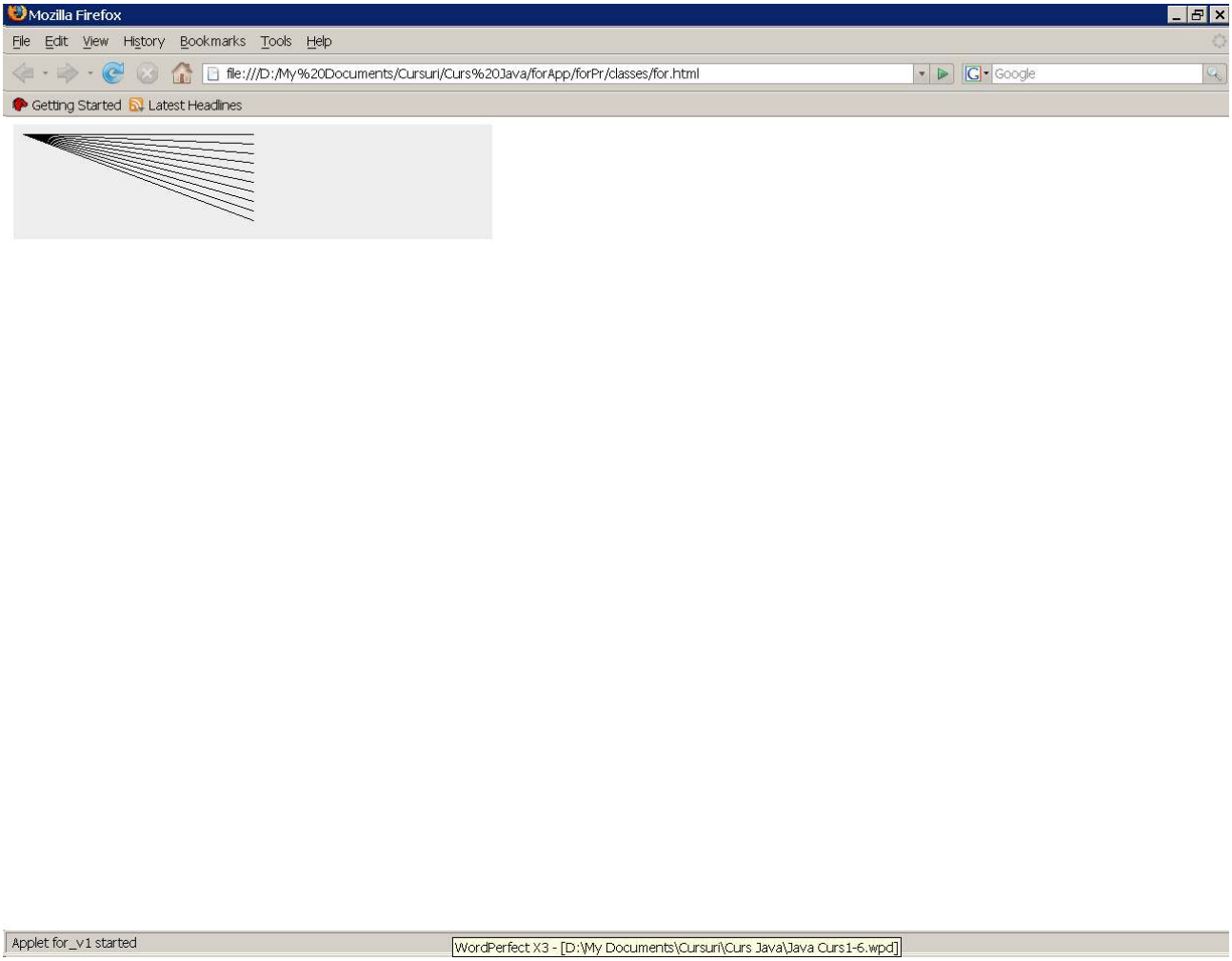
```
<HEAD>
<APPLET CODE="for_v1.class" WIDTH=500 HEIGHT=120>
  </APPLET>
</HEAD>
```



Rularea aplicaţiei se poate cu aplicaţia appletviewer.exe sau din navigatorul de Internet. Pentru rularea din appletviewer dechidem o consolă şi scriem: `appletviewer for.html`

Acelaşi fişier `for.html` va putea fi deschis

în IE sau Mozilla Firefox după cum se vede în continuare.



5

Concepte de programare orientată pe obiect în Java

Ce este programarea orientată pe obiect (POO)

- este o metodologie (paradigmă) de proiectare a programelor
- este inspirată din realitate şi se bazează pe conceptul de modelare pe obiect
- trecerea de la realitate la model se face prin abstractizare
- câteva caracteristici:
  - asigură reutilizarea şi adaptarea simplă a codului;
  - conduce la un cod mai puţin vulnerabil la manipulări eronate.

Programarea orientată pe obiect (object oriented programming, în engleză) este o metodologie de scriere a programelor. În programare, modul de abordare a scrierii aplicaţiei, adică metoda, mai poartă şi denumirea de paradigmă (vine din grecescul “paradeigma” care înseamnă “a demonstra”). Fiecare paradigmă a fost pusă la punct cu un scop precis, câteva dintre caracteristicile celei obiectuale sunt:

- reutilizarea codului;
- întreţinere uşoară;
- adaptarea codului cât mai simplă la situaţii noi;
- minimizarea vulnerabiltăţii codului la utilizarea greşită din partea programatorilor care în refolosesc.

Într-o descriere foarte generală, în programarea orientată pe obiect, aplicaţia de realizat este privită în termenii manipulării unor entităţi numite obiecte şi nu prin prisma acţiunilor care trebuiesc implementate (cum se face în programarea structurată).

POO are drept sursă lumea înconjurătoare. Omul, în viaţa de toate zilele, îşi propune să fie cât mai concret în descrierea lumii care îl înconjoară. El, dă nume scurte obiectelor înconjurătoare pentru ca să le poată identifica cât mai exact. De exemplu, spunem LOGAN, în loc de automobilul produs la Uzina de Automobile din Piteşti, în colaborare cu Renault. Toate aceste nume particulare ascund însă în spatele lor denumiri mai generale cum sunt: automobile, atomi, stele, oceane, oameni etc. Deseori, un obiect are la bază altele. Trăim, astfel, într-o lume orientată pe obiect. O descriere a termenului de obiect este dată de Booch prin "Un obiect are stare, comportament şi identitate; structura şi comportamentul similar al obiectelor este definit prin clasele lor comune; termenii de instanţă şi obiect sunt echivalenţi". Mai general, un obiect este ceva căruia i se poate aplica un concept. Conceptul este o idee sau o notaţie comună, mai multor obiecte înconjurătoare.

Obiectul, din viaţa de toate zilele, în programarea orientată pe obiect este implementat ca o structură de dată (abstractă) încapsulată cu un grup de subrutine denumite metode, care acţionează asupra datelor. Structura obiectului rezultă ca urmare a abstractizării. Abstractizarea implică o generalizare, prin

ignorarea sau ascunderea detaliilor, în vederea separării unor proprietăţi sau caracteristici ale unei entităţi de obiectul fizic sau de conceptul pe care îl descrie. Ca urmare, în programarea orientată pe obiect, căutarea şi definirea proprietăţilor obiectelor din lumea reală definesc specificul acestei paradigme.

Proiectarea orientată pe obiect conduce la o aplicaţie formată din colecţii de obiecte care cooperează. Între obiectele aplicaţiei se stabilesc relaţii de legatură, iar cooperarea între obiecte se face prin mesaje transmise între acestea.

Etapele tipice acestei paradigme de proiectare sunt: identificarea claselor şi a obiectelor, identificarea semanticii acestora, identificarea relaţiilor şi descrierea interfeţelor şi a implementărilor de clase şi de obiecte.

Câteva nume care au avut contribuţii importante în crearea şi dervoltarea POO sunt:

- **Grady Booch**, de la Rational, care a dezvoltat renumitul instrument de modelare numit "*Rose*";
- **Kirsten Nygaard** şi **Ole-Johan Dahl**, inventatorii limbajului *Simula*, primul limbaj de programare orientat pe obiect şi inventatorii proiectării orientate pe obiect;
- **Adelle Goldberg**, **Kay Alan** şi **Dan Ingallas** fondatorii limbajului obiectual *Smalltalk* şi coautori ai lui "*Smalltalk*-80";
- **Brax Cox**, creatorul limbajului *Objective-C*, care implementează în limbajul C facilităţile legate de identificarea obiectelor şi mecanismul de mesaje din Smalltalk;
- **Meyer Bertrand** creatorul limbajului *Eiffel*;
- **Bjarne Stroustrup**, inventatorul limbajului *C++*, o extensie a limbajului C, probabil, unul dintre cele mai populare limbaje de programare orientate pe obiect.

Conceptul de obiect - definiții

- depinde de contextul în care se lucrează:
  - **filozofic**: o entitate ce se poate recunoaşte ca fiind distinctă de altele - are identitae proprie
  - **proiectarea oo**: o abstractizare a unui obiect din lumea reală
  - **teoria tipurilor**: o structură de date împreună cu funcţiile de prelucrare a lor
- se defineşte prin atribute şi operaţii
- atributele definesc starea, operaţiile permit modificarea stării.

Din punctul de vedere al aplicaţie din care face parte obiectul el trebuie să asigură o parte din funcţiile necesare funcţionării întregii aplicaţii. Termenul de **modelare a obiectelor** se foloseşte pentru a descrie procedura de căutare a obiectelor prin care să descriem problema de rezolvat.

Obiectele sunt descrise prin **atribute** şi **operații**. **Atributele** sunt caracteristici care se modifică, iar **operațiile** sunt acţiuni pe care obiectul le poate face. De exemplu, pisica are culoare, rasă şi masă (greutate), acestea ar fi câteva dintre atribute ei, ea pot prinde şoareci, mânca, dormi sau mieuna, acestea ar fi câteva dintre acţiunile specifice pisicii.

Într-un **model de obiecte** toate datele sunt stocate sub formă de atribute ale obiectelor. Atributele unui obiect vor putea fi manipulate prin acţiuni sau operaţii. Singura modalitate de modificare a atributelor este prin utilizarea operaţiilor. Atributele pot fi uneori obiecte. Funcţionarea la nivelul unui model de obiecte este definită prin operații. Un obiect poate accesa şi utiliza operaţiile unui al obiect. Operaţiile sunt cele ce modifică starea obiectului. Modelarea orientată pe obiect este despre găsirea obiectelor şi a dependenţelor între acestea. Dependenţele între obiecte reprezintă modul în care ele se aranjează, în sensul legăturilor - a relaţiilor, a asocieriilor - care se formează, pentru a ne soluţiona problema. De regulă, indentificarea obiectelor unui model de obiecte se face pe bază de substantive şi verbe. Sunbstativele vor fi obiecte, iar verbele operaţii.

Relaţii între obiecte

Agregarea, cunoscută în literatura de specialitate sub prescurtarea de relaţie “has-a”, apare atunci când un obiect este compus din mai multe sub-obiecte. Comportamentul obiectului complex este definit prin comportamentul părţilor componente, distincte şi aflate în interacţiune. În procesul de descompunere al unui obiect în obiecte mai simple, acestea, deseori, vor putea fi reutilizate în alte aplicaţii. Agregarea ne permite deci, reutilizarea componentelor unei aplicaţii într-o altă aplicaţie.

Delegarea, cunoscută în literatura de specialitate sub prescurtarea de relaţie “uses-a”, apare atunci când un obiect, format total sau parţial din alte obiecte, lasă obiectele componente să îi definească comportamentul. Obiectul nu are un comportament implicit, la nivel de interacţiune cu alte obiecte, acesta fiind moştenit de la sub-obiectele lui.

Conceptul de clasă

- clasa este o generalizare a unei mulţimi de obiecte
- definiţia unei clase se face prin specificarea:
  - variabilelelor care îi definesc starea - variabile de instanţă;
  - metodele prin care se modifică starea - metode de instanţă;
  - relaţiile cu alte clase.

```
class nume_clasa extends nume_parinte {
//STARE - ATRIBUTE
    tip v_1;
    . . .
    tip v_n;
//METODE - OPERATII
    tip metoda_1(lista parametri) {
        corp_1
    }
    . . .
    tip metoda_k(lista parametri) {
        corp_k
    }
} //TERMINARE definitie de clasa
```

Clasa reprezintă un termen general asociat procesului de clasificare. Scopul clasificării este cel de încadrare a unui obiect într-o colecţie de obiecte similare ce poartă denumirea de clasă. Toate obiectele clasei au aceleaşi atribute şi fac aceleaşi operaţii, dar starea lor diferă şi au o identitate proprie. Un obiect particulat poartă denumirea de instantă. Din punctul de vedere al programatorului clasa reprezintă un nou tip de date. Clasa este o definiţie statică, prin care descriem un grup de obiecte. Clasa este un şablon (un tip), un concept, în timp de obiectele există în realitate (în timpul rulării programului). De exemplu, `String` reprezintă o clasă, iar variabilele de tipul `String` reprezintă obiecte ale aplicaţiei. Clasa `String` este una singură, în timp ce numărul obiectelor de tipul `String` nu este limitat. În procesul de proiectare a claselor apare situaţia în care mai multe clase distincte au părţi comune. Părţile comune pot fi cuprinse într-o singură clasă şi moştenite la nivelul unor clase ulterioare. **Moştenirea**, cunoscută în literatura de specialitate sub prescurtarea de relaţie “is-a”, apare atunci când o clasă are ca părinte o altă clasă. Prin moştenire, noua clasă preia toate caracteristicile clasei moştenite, la care mai poate adăuga unele noi. Astfel, noua clasă o extinde pe cea moştenită, fiind o specializare a acesteia. Moştenirea creşte claritatea proiectului şi productivitatea, deoarece noi clase pot fi create pe baza unora existente.

Definiţia unei clase presupune descrierea precisă a formei şi a naturii acesteia. Pentru aceasta trebuie specificate datele (atributele) pe care le conţine şi codul (operaţiile) care va opera cu respectivele date.

Datele, se vor stoca în variabile ce poartă denumirea de **variabile** de instanţă, iar operaţiile se implementează în **metode**. Variabilele şi metodele unei clase poartă denumirea comună de **membri**. Variabilele de instanţă au această denumire deoarece sunt proprii fiecărei instaţieri de clasă, adică fiecărui obiect din clasa respectivă. Datele unui obiect sunt distincte, separate de datele unui alt obiect din aceeaşi clasă.

Exemplul alăturat defineşte în clasa Punct:

- două variabile de instanţă:

```
private double x;
private double y;
```

- doi constructori:

```
Punct()
Punct(double abscisa ...)
```

- şase metode:

```
public void setX(double abscisa)
public void setY(double ordonata)
public double x()
public double y()
public double distantaOrigine()
public String toString()
```

Clasa `Grafica`, are metoda `main()`, este punctul de plecare al aplicaţiei şi defineşte două obiecte, P1 şi P2, din clasa `Punct`.

Aici se prezintă:

- modul în care se **declară obiectele** unei clase:

```
Punct p1;
```

- modul în care se **alocă spaţiu unui obiect**, prin folosirea operatorului `new`:

```
p1 = new Punct();
```

- modul în care **se referă o metodă publică** dintr-un obiect:

```
p1.setX(12);
```

```
public class Punct {
/*Atributele clasei,
permit stocarea STARII */
    private double x;
    private double y;

/*Constructorii clasei,
permit initializarea obiectelor din
clasa Punct */

  Punct() {
      setX(0);
      setY(0);
  }

  Punct(double abscisa, double ordonata) {
      setX(abscisa);
      setY(ordonata);
  }

/*Metodele clasei,
permit accesul si modificarea starii
obiectelor din clasa Punct */

  public void setX(double abscisa) {
      x = abscisa;
  }

  public void setY(double ordonata) {
      y = ordonata;
  }

  public double x() {
      return x;
  }

  public double y() {
          return y;
  }

  public double distantaOrigine() {
      return Math.sqrt(x*x+y*y);
  }

   public String toString() {
      return "<" + x + "," + y + ">";
  }
}

public class Grafica {
 public static void main(String[] args){
   Punct p1;
   Punct p2 = new Punct(-1.,7.);
   p1 = new Punct();
   System.out.println("p1 = " + p1);
   System.out.println("p2 = " + p2);
   p1.setX(12);
   p2.setY(13.345);
        System.out.println("p1   =   "   +
p1.toString());
  System.out.println("p2 = " + p2);
 }
}
```

Crearea obiectelor

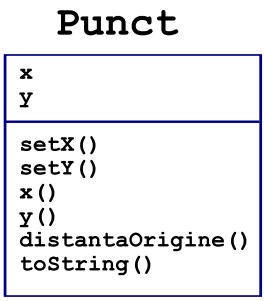
- obiectele se creează cu operatorul **new**

```
refObiect = new nume_clasa();
```

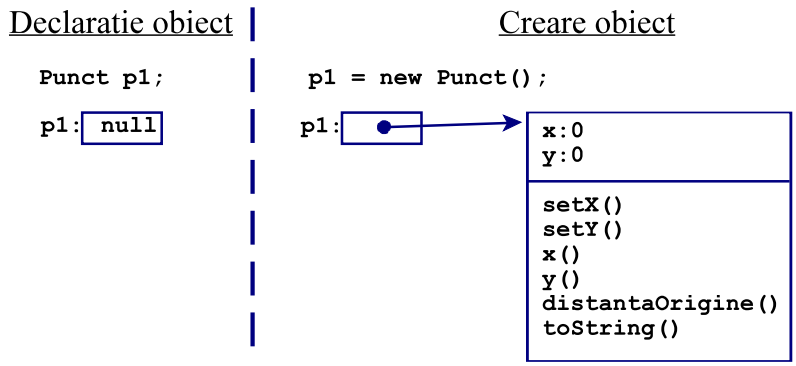
- new realizează următoarele acţiuni:
  - alocă memorie pentru nou obiect;
  - apelează o metodă specială de iniţializare numită constructor;
  - întoarce o referinţă la noul obiect.

În Java obiectele se creează cu operatorul `new`. În aplicaţia anterioară se crează două instanţe ale clasei `Punct` prin liniile de cod:

```
Punct p2 = new Punct(-1,7);
p1 = new Punct();
```



Prima linie declară şi creează variabila obiect `p2`, iar cea de a doua creează variabila `p1`, ea fiind deja declarată anterior. Operatorul new are ca efect alocarea de spaţiu în RAM pentru obiect şi are ca rezultat o referinţă către obiectul creat. Respectiva referinţă se va stoca în variabila obiect, mai sus `p1` şi `p2`, prin care vom putea accesa toţi membrii obiectului în cauză. Reprezentarea grafică a clasei `Punct` se poate da sub forma alăturată. Pe baza acestei clase vor crea două variabile obiect sau instanţe, `p1` şi `p2`. Spaţiul de date pentru instanţa `p1` va fi reprezentat grafic pentru a descrie etapele parcurse la crearea unui obiect. În momentul declarării obiectului `p1` prin linia `Punct p1;` în RAM se alocă spaţiu în RAM pentru o referinţă căte un obiect. Spaţiul de date alocat în RAM constă într-o locaţie cu numele `p1` în care este stocată o valoarea specială `null`. Această valoarea specială arată faptul că obiectului încă nu i s-a alocat spaţiu. În etapa următoare, operatorul `new` alocă spaţiu pentru obiect pe baza definiţiei de clasă, apelează constructorul clasei pentru a iniţializa variabilele de instantă `x` şi `y` cu valorile 0 şi 0, apoi întoarce o referinţă (o adresă de RAM ce se reprezintă grafic ca o săgeată cu vârful de la instanţă către spaţiul alocat obiectului) către nou obiect ce va fi stocată în variabila obiect `p1`.



Conceptul de referinţă

- prin referinţă se înţelege adresa unei locaţii de RAM
- membrii unui obiect se accesează printr-o referinţă la obiect stocată în variabila obiect:

```
p1.setX(12);
```

- la declarare, variabilele obiect sunt iniţializate automat cu **null**:

Punct p1; este **Punct p1 = null;**

- valoarea **null** poate fi comparată cu alte referinţe:

```
if (p1 == null)
    p1 = new Punct();
```

- distrugerea unui obiect se face prin setarea referinţei lui la **null**:

```
p1 = null;
```

RAM-ul este organizat sub forma unui tablou liniar de elemente numite locaţii. Fiecare locaţie este indentificată unic printr-un număr întreg numit adresă şi poate stoca o valoare. În reprezentările grafice utilizate pentru descrierea stării spaţiului de date dreptunghiul reprezintă o locaţie (sau un grup de locaţii consecutive), numele variabilei se scrie în stânga dreptunghiului fiind echivalentul adresei locaţiei. În interiorul dreptunghiului se va scrie valorea stocată în respectiva locaţie. Valorile numerice stocate în locaţii sunt dependente de tipul de date folosit la declararea variabilei. Pentru tipurile primitive locaţiile conţin reprezentarea binară a valorilor atribuite respectivelor variabile. În cazul variabilelor obiect, locaţiile stochează adresa de început în RAM a grupului de locaţii alocat de `new` pentru stocarea instanţei. O variabilă ce stochează o adresă poartă denumirea de referinţă (denumiri altenative sunt pointer - din engleză sau poantor - din franceză).

Atribuirea în situaţia referinţelor are ca efect obţinerea a două variabile ce referă acelaşi obiect. Prin atribuire ajungem să avem mai multe variabile obiecte distincte care referă însă un singur obiect.

Încapsularea (encapsulation) și vizibilitatea membrilor

- vizibilitatea membrilor este controlată prin specificatorii de acces:
  - **public**
  - **private**
  - **protected**
- încapsularea asigură:
  - protecția datelor prin accesul la acestea numai prin intermediul unei intefețe publice (**public**)
  - izolarea părților vizibile prin interfață de implementarea internă (**private**)
- avantajele încapsulării:
  - nu pot fi realizate modificări ilegale ale datelor, deoarece accesul direct la ele este restricționat
  - gruparea datelor şi a operaţiilor ce se pot realiza cu ele permit depistarea uşoară a eventualelor probleme
  - modificările interne ale clasei nu se reflectă şi în afara ei (dacă se păstrează aceeaşi interfaţă), ca urmare restul programului nu trebuie modificat

Membrii `public` vor fi vizibili pentru orice porţiune de cod din aplicaţie. Membrii `private` vor fi vizibili numai pentru membrii clasei din care face parte. Membrii `protected` sunt vizibili numai prin moştenire.

Condiţiile de implementare corectă a încapsulării:

- toate variabilele de instanţă sunt declarate cu `private`;
- numai metodele `public` ale obiectului vor putea fi utilizate pentru accesul la datele private ale acestuia;

În continuare se prezintă o astfel de implementare a clasei `Punct`:

În această implementare `x`, `y` şi `distanta` sunt toate declarate `private` , deci invizibile pentru un utilizator al clasei.

Modificarea valorilor acestor variabile de insstanţă se face numai sub controlul metodelor:

- automat la crearea instanţierea obiectelor prin acţiunea automată a constructorilor: `Punct()`, `Punct(double x, double y)`
- prin apelul metodelor: `setX(double x)`, `setY(double y)`

Vizualizarea valorilor stocate în variabilele provate se face cu metodele: `x()`, `y()` şi `distantaOrigine()`.

```
public class Punct {
//Campuri
  private double x;
  private double y;
  private double distanta;

//Constructori
  Punct() {
      setX(0);
      setY(0);
      distanta = 0;
  }

  Punct(double x, double y) {
      setX(x);
      setY(y);
      actualizareDistanta();
  }

// Metode
  public void setX(double x) {
      this.x = x;
      actualizareDistanta();
  }

  public void setY(double y) {
      this.y = y;
      actualizareDistanta();
  }

  public double x() {
      return x;
  }

  public double y() {
      return y;
  }

  public double distantaOrigine() {
      return distanta;
  }

  private void actualizareDistanta() {
       distanta = Math.sqrt(x*x+y*y);
    }

   public String toString() {
       return "<" + x + "," + y + ">";
  }
}
```

Metode

- metoda implementează o operaţie cu datele obiectului
- metodele se implementează sub forma unor proceduri sau funcţii (întorc valori cu **return**) ce realizează anumite operaţii
- metodele pot fi definite numai într-o deniniţie de clasă, astfel:

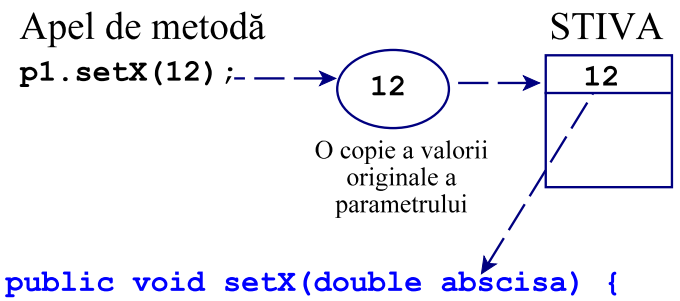
```
vizibilitate tip nume_met([lista parametri])
{
    corp_1
}
```

tip - defineşte tipul de date întors de metodă
nume_met - defineşte numele metodei
[lista parametri] - o secvenţă opţională de perechi **tip nume** separate prin virgule

Operaţiile unei clase se definesc cu ajutorul metodelor. Metodele din Java sunt echivalentul procedurilor şi funcţiilor din alte limbaje de programare, cu excepţia ca ele trebuie scrie obligatoriu în interiorul definiţie de clasă (nu există conceptul de metodă globală în Java). Vizibilitatea controlează cine anume va putea apela metoda. Pentru public acesta va fi apelabilă din toată aplicaţia, pentru private numai de alte metode din clasa în care s-a definit. Dacă metoda întoarce o valoare, adică este o funcţie, ea va avea specificat unul dintre tipurile deja discutate şi o instrucţiune de salt necondiţionat `return expresie` ca va genera întoarcerea în codul apelant cu valoarea lui `expresie`. Dacă metoda nu întoarce valoare, adică este o procedură, tipul ei va fi obligatoriu `void` şi nu va conţine instrucţiune `return`. Pentru ca operaţia implementată la nivelul unei metode să aibă loc aceasta trebuie apelată. Prin apel, va se realiza rularea instrucţiunilor cuprinse în corpul metodei. Terminarea metodei se face la rularea unei instrucţiuni `return`, din corpul ei sau, în lipsa lui `return`, la atingerea acoladei de inchidera a corpului ciclului. La terminarea rulării unei metode rularea aplicaţiei va continua cu instrucţiunea imediat următoare celei de apel a metodei. Porţiune de cod din care se face apelul metodei se numeşte cod apelant, iar porţiune de cod corespunzătoare metodei apelate se numeşte cod apelat. Metoda poate avea zero sau mai mulţi parametri. Parametrii sunt valori transferate metodei în momentul apelului acesteia. Aceste valori, dacă există, vor putea fi prelucrate în corpul metodei pentru a produce rezultatele întoarse cu `return`.

Apelul metodelor şi metodele de transfer al parametrilor

- sintaxa generală pentru apelul unei metode este: **`refObiect.nume_met(argumente)`**
- parametrii sunt variabile ce primesc valori în momentul apelului şi se numesc argumente
- valorile parametrilor ce sunt tipuri de date primitive se transferă prin valoare
- la transferul prin valoare, metodei, i se transferă o copie a valorilor parametrilor din programul apelant



Aplelul de metodă
Apelul unei metode se face se face cu ajutorul operatorului punct. Forma generală a unui apel de metodă este `refObiect.nume_met(argumente)`, în situaţia exemplului de mai sus apelul de metodă este `p1.setX(12)`.

Parametrii şi argumente
Metodele nu au nevoie de parametri. Folosirea parametrilor permite însă o generalizare a meteodei în sensul că aceasta va putea opera cu mai multe date de intrare şi/sau în mai multe situaţii uşor distincte.

Termenul de **parametru** se foloseşte pentru o variabilă definită la nivelul metodei care primeste o valoare în momentul apelării metodei. Persistenţa şi vizibilitatea parametrilor este limitată la corpul metodei. Termenul de **argument** se foloseşte pentru o valoarea care se transferă metodei atunci când aceasta este apelată. Astfel, în exemplul prezentat `abscisa` este parametru, iar `12` este argument.

Apel prin valoarea
Exită două tipuri de parametri în Java: tipuri primitive şi referinţe la obiecte. În cazul ambelor categorii de parametri apelul se face întotdeauna prin valoare. Respectiv metoda primeşte o copie a argumentelor prin stivă şi nu va putea să modifice conţinutul iniţial al argumentelor ce se transferă în parametrii din

interiorul metodei. În cazul parametrilor de tipul referinţă la obiect se poate modifica starea obiectului utilizând metodele puse la dispoziţie în acest scop, dar nu se pot modifica referinţele originale ale argumentelor de tipul referinţă la obiect în corpul metodei. Iată un exeplu:

Este imposibilă modificarea unui argument de tip primitiv prin parametrii metodei. Pentru aceasta s-a scris metoda `dubleazaX()` ce ar trebui să dubleze valorea argumentului `x` cu valoarea iniţială `5`. Observaţi în rezultate că deşi în corpul metodei dublarea se face, la terminarea metodei şi revenirea din aceasta, valoarea dublată se pierde. Iată cum lucrează Java:

1. parametrul `a` este iniţializat cu o copie a valorii stocate în argumetul `x`, adică cu o copie a lui `5`;
2. `a` este dublat, adică devine `10`, dar `x` rămâne `5`;
3. metoda se termină, iar parametrul îşi încetează persistenţa, deci variabila `a` nu mai există.

În siuaţia în care parametrul (vezi metoda `schimbaX()`) este o referinţă la un obiect, Java lucrează astfel:

1. parametrul `x` primeşte o copie a referinţei către obiectul `p1`;
2. metoda `setX()` este aplicată asupra referinţei la obiect, obiectul `PunctOK` fiind referit în acest moment de `x` şi `p1`;
3. metoda se termină şi parametrul `x` încetează să mai existe.

Metoda poate modifica starea unui parametru obiect deoarece primeşte copie o referinţei la obiectul iniţial. Atât copia cât şi argumentul referă însă acelaşi obiect.

```
public class GraficaOK {
  public static void main(String[] args) {
    double x = 5;
    PunctOK p1 = new PunctOK();
    PunctOK p2 = new PunctOK(-1.,7.);

   System.out.println("in afara lui dubleazaX x este:
" + x);
     dubleazaX(x);
   System.out.println("in afara lui dubleazaX x este:
" + x);

    System.out.println("p1 = " + p1);
    System.out.println("p2 = " + p2);
    interschimba(p1, p2);
    System.out.println("p1 = " + p1);
    System.out.println("p2 = " + p2);

    schimbaX(p1);
    System.out.println("p1 = " + p1);
    }

//modificarea valorii argumentului
//NU se reflecta in codul apelant
 public static void dubleazaX(double a) {
    a = 2. * a;
  System.out.println("in dubleazaX x este: " + a);
    }

//metoda de interschimbare NU lucreaza
//ca urmare a transferului prin valoare
 public static void interschimba(PunctOK x, PunctOK y)
{
    PunctOK aux = x;
    x = y;
    y = aux;
 }
//lucreaza corect
 public static void schimbaX(PunctOK x) {
   x.setX(100);
 }
}
```

Rezultate:

```
in afara lui dubleazaX x este: 5.0
in dubleazaX x este: 10.0
in afara lui dubleazaX x este: 5.0
p1 = <0.0,0.0>
p2 = <-1.0,7.0>
p1 = <0.0,0.0>
p2 = <-1.0,7.0>
p1 = <100.0,0.0>
```

Constructori

- definește starea inițială a obiectelor
- are același nume cu clasa
- este apelat automat după crearea obiectului, înainte ca operatorul **new** să se termine;
- nu întorc o valoarea de un anumit tip (nici măcar **void**)

```
public nume_clasa(){
...
};
```

- sintactic sunt identice cu metodele (pot avea parametri)

Constructorii clasei `PunctOK` sunt prezentați în secvența de cod alăturată. Observați că există doi contructori ce poartă numele clasei, diferența între ei fiind parametrii. În măsura în care este uitată definirea lor, Java utilizează un constructor implicit (default constructor), fără argumente și fără nici un fel de cod, ce permite, măcar crearea obiectelor din clasa respectivă și inițializarea la valori implicite a variabilelor de instanță. Constructorul implicit poate fi scris și explicit dacă inițializările implicite (toate datele numerice iau valoarea 0, `Boolean false` iar obiectele `null`) nu sunt cele dorite.

Constructorul implicit `PunctOK()` va fi apelat automat la creare unui obiect prin linia ce cod `PunctOK p1 = new PunctOK();`, iar constructorul `PunctOK(double x, double y)` va fi apelat automat la crearea unui obiect cu linia de cod `PunctOK p2 = new PunctOK(-1.,7.);`. Java identifică constructorul ce trebuie să-l apeleze pe baza numărului și tipurilor parametrilor și a argumentelor.

```
public class PunctOK {
//Campuri
 ...

//Constructori
  PunctOK() {
      setX(0);
      setY(0);
      distanta = 0;
  }

  PunctOK(double x, double y) {
      setX(x);
      setY(y);
      actualizareDistanta();
  }
 ...
}
```

Supraîncărcarea (overloading)

- permite folosirea aceleiaşi sintaxe pentru obiecte de tip diferit
- la nivel de metode de clasă, apare atunci când avem mai multe metode cu acelaşi nume, dar cu declaraţii de parametrii dinsticte
- supraîncărcarea este o formă polimorfism
- Java se foloseşte de tipul şi/sau de numărul parametrilor din apel pentru a determina care dintre metodele supraîncărcate să le apeleze

În exemplul prezentat clasa `PunctOK` are mai mulţi constructori. Deşi constructorii au acelaşi nume, au număr şi/sau tipuri de parametrii distincţi. Compilatorul determină care dintre constructori urmează să fie apelat prin compararea numărului şi a tipurilor de parametri din definiţia constructorului cu numărul şi tipurile valorilor argumentelor folosite în apelul contructorului. Dacă o astfel de corespondenţă nu se poate stabili sau se pot stabili, simultan, mai mult de o singură corespondenţă biunivocă va genera o eroare legată de rezolvarea supraîncărcătrii (overloading resolution).

Supraîncărcarea în Java este posibilă atât la nivel de constructori cât şi la nivel de metode. Descrierea completă a unei metode se realizează pe baza numelui metodei împreună cu tipurile parametrilor acesteia. Aceste informaţii poarte denumire de semnătura metodei (method signature). Tipul întors de metodă nu face parte din semnătura metodei, atfel nu putem avea două metode cu acelaţi nume şi parametri, dar cu tip întors diferit.

Referința **this**

- **this** este argument implicit al oricărei metode
- **this** este o referință la obiectul curent
- **this** poate fi folosit în corpul oricărei metode ale obiectului curent

```
public void setX(double x) {
    this.x = x;
    actualizareDistanta();
}
```

Obiectul curent poate fi referit în Java prin folosirea cuvântului cheie `this`. Toate metodele unei instanţe primesc ca argument implicit pe `this`. Obiectul curent este obiectul al cărei metode a fost apleată.

Exemplul alăturat prezintă modul de utilizare în cod a lui `this`. Dintre situaţiile în care se doreşte folosirea explicită a lui `this` amintesc:

- numele unei variabile de instanţă este identic cu cel al unui parametru de metodă (vezi exemplul alăturat), în această situaţie accesul la variabila instanţă se face prin `this.nume_var_instanta`;
- este necesară transferarea unei referinţe la obiectul curent ca argument, unei alte metode;
- un constructor al unei clase apelează un alt constructor al aceleiaşi clase (permite evitarea reluării unor secvenţe de iniţializare).

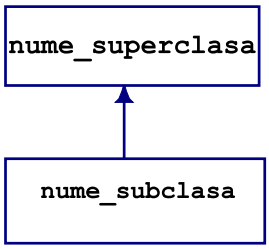
```
public class Punct {
//Campuri
  private double x;
  private double y;
  private double distanta;

...

// Metode
  public void setX(double x) {
      this.x = x;
      actualizareDistanta();
  }

  public void setY(double y) {
      this.y = y;
      actualizareDistanta();
  }
...
}
```

Moştenirea

- moştenirea permite crearea unor ierarhii de clasificare
- **extends** se foloseşte în Java pentru a descrie moştenire între clase

```
class nume_subclasa extends nume_superclasa {
    adăugarea de noi variabile şi metode
}
```

- reprezentarea grafică a moştenirii se face printr-o săgeată (care se citeşte "moşteneşte de la")
- clasa care există se numeşte superclasă, clasă de bază sau clasă părinte, noua clasă se numeşte sublclasă, clasă derivată sau clasă copil



Moştenirea permite crearea de noi clase pe baza unora existente. Noua clasă se numeşte subclasă şi se defnieşte prin extinderea unei clase deja exisntente numită superclasă. La definirea subclasei se vor specifica numai diferenţele faţă de superclasă. Cele mai generale variabile şi metode sunt plasate în superclasă iar cele mai specializate în subclasă. În situaţia în care unele metode ale superclasei nu sunt corerspunzătoare subclasei acestea pot şi suprascrise. Aplicaţia ce urmează îşi propune să creeze o bibliotecă grafică. Comceptul de bază aici este cel de cooronate pe care se va baza acela de punct. Clasa de bază va fi deci `Coordonate` iar cea derivată din aceasta este `Punct`. Noua clasă are adăugate variabila de instanţă `nume` , metoda `setNume()` şi suprascrisă metoda `toString()`. Deşi există două metode cu acelaşi nume `toString()` în superclasă şi însubclasă, Java va ştii să apeleze metoda dorită corect. Constructorii noi clase sunt `Punct()`, `Punct(double x, double y)` şi `public Punct(double x, double y, String nume)`. Fiecare subclasă poate accesa constructorul superclasei iar codul comun poate fi scris în superclasă şi apelat în subclasă prin folosirea lui `super`. Noua clasă însă nu moşteneşte constructorii clasei de bază ea beneficiind în mod direct numai de constructorul implicit. Deşi subclasa include toţi membrii superclasei, aceasta nu va putea membrii care au fost declaraţi cu `private`.

```
public class Coordonate {
    private double x;
    private double y;

    Coordonate(){
        setX(0);
        setY(0);
    }

    Coordonate(double x, double y){
        setX(x);
        setY(y);
    }

    public void setX(double x) {
        this.x = x;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public void setY(double y) {
        this.y = y;
    }

    public String toString() {
       return "(" + x + "," + y + ")";
    }
}


public class Punct extends Coordonate
{
    private String nume;

    public Punct() {
        super();
        setNume("P");
    }

    public Punct(double x, double y) {
        super(x,y);
        setNume("P");
    }

     public Punct(double x, double y,
String nume) {
        this(x,y);
        setNume(nume);
    }

    public Punct(Coordonate c) {
        this(c.getX(),c.getY());
        setNume(nume);
    }

    public void setNume(String nume){
        this.nume = nume;
    }

    public String toString() {
        return nume +"(" + getX() + "," +
getY() + ")";
    }
}

public class Mostenire {
  public static void main(String[] args) {
   Coordonate c1 = new Coordonate();
   Coordonate c2 = new Coordonate(1,2);

   Punct p1 = new Punct();
   Punct p2 = new Punct(1,1);
   Punct p3 = new Punct(1,2,"P3");
   Punct p4 = new Punct(c1);

   System.out.println(c1);
   System.out.println(c2);
   System.out.println(p1);
   System.out.println(p2);
   System.out.println(p3);
   System.out.println(p4);

   c1 = p1;
   System.out.println(c1);
   c1 = c2;
   System.out.println(c1);

  }
}
```

Rezultate:

```
(0.0,0.0)
(1.0,2.0)
P(0.0,0.0)
P(1.0,1.0)
P3(1.0,2.0)
P(0.0,0.0)
P(0.0,0.0)
(1.0,2.0)
```

Referința **super**

- **super** este o referință la clasa de bază
- se utilizează la apelul constructorilor din clasa de bază
- trebuie să fie fie prima linie în constructorul clasei derivate
- se poate apela constructorul oricărei clase de bază
- se poate referi oricare membru al superclasei

Apelul unui constructor al superclasei:

```
super(lista parametri)
```

Referirea unui membrul al superclasei:

```
super.membru
```

Aplicaţia anterioară prezintă modul de utilizare a lui `super`. Asemenea lui `this`, ce avea două semnificaţii:

- o referinţă de tip argument implicit la metodele obiectului;
- o referinţă la obiectul în cauză prin care se pot apela alţi construcori ai aceleiaşi clase;

şi cuvântul cheie `super` are două semnificaţii asigurând:

- apelul unei metode din superclasă;
- apelul constructorilor superclasei.

De exemplu, în codul următor, metoda `toString()` a clasei derivate `Punct` poate fi rescrisă utilizând pe super pentru apelul `toString()` din clasa de bază `Coordonate` astfel:

```
public String toString() {
   return nume + super.toString();
   //return nume +"(" + getX() + "," + getY() + ")";
}
```

Polimorfism

- polimorfismul defineşte abilitatea de a avea mai multe forme
- în OOP apare atunci când o variabilă, în timpul rulării, este ataşată unor obiecte de tipuri distincte
- moştenirea generează subclase între care avem o dependenţă de tipul “is-a”
- “is-a” reprezintă un principiu de substituţie - conform lui se poate folosi o subclasă în locul unei superclase
- variabilele Java sunt polimorfice

Din exemplul prezentat vom analiza liniile de cod:

```
Coordonate c1 = new Coordonate();
Coordonate c2 = new Coordonate(1,2);
Punct p1 = new Punct();

c1 = p1; //atribuire polimorfica
System.out.println(c1);
c1 = c2;
System.out.println(c1);
```

```
P(0.0,0.0) //c este un p
(1.0,2.0)  // c este un c
```

Observaţi că `c1` şi `c2` sunt obiecte declarate din clasa `Coordonate`, iar `p1` este din clasa `Punct`. În aceste condiţii variabila `c1` ajunge să refere două tipuri distincte de obiecte. La utilizarea lui `System.out.println(c1)` este aplelată automat metoda `toString()` a obiectului în cauză (linia este echivalentă cu `System.out.println(c1.toString())`). Atunci când `c1` referă obiectul `p1` se va apela metoda `toString()` a clasei `Punct`, iar atunci când `c1` referă pe `c2` se va apela `toString()` a clasei `Coordonate`. JVM va şti tipul actual al obiectului la care `c1` se referă, motiv pentru care va apela corect metoda dorită. Faptul ca variabila obiect `c1` poate referi, în timpul rulării aplicaţiei, tipuri multiple poartă denumirea de **polimorfism**. Selectarea automată a metodei de apelat, în timpul rulării aplicaţiei, poartă denumirea de **legare dinamică** (dynamic binding). Procedura de aplicare a legării dinamice este, principial, următoarea:

1. Compilatorul caută pentru obiectul în cauză tipul şi numele metodei. Pot exista mai multe metode cu acelaşi nume într-o clasă şi în superclase. Compilatorul va determina toate metodele candidate la apel din clasă şi superclase;

2. Compilatorul determină tipul argumentelor din apelul de metodă. Dacă între metodele cu acelaşi nume există una care are parametrii ce se potrivesc cel mai bine cu argumentele apelului atunci metoda în cauză va fi selectată pentru apel. În acest moment compilatorul ştie numele metodei şi tipul parametrilor metodei ce urmează să fie apelată.
3. Dacă metoda este private, static, final sau constructor compilatorul vă şti precis care metodă urmează să fie apelată. Acestă situaţie poartă denumirea de **legare statică** (static binding). Altfel, determinarea metodei de apelat depinde de tipul actual al argumentelor şi legarea dinamică trebuie să fie utilizată.
4. În timpul rulării aplicaţie JVM trebuie să apeleze metoda pe baza tipului actual de obiect la care variabila obiect se referă. Dacă tipul actual este subclasa `Punct` atunci se aplelează respectiva metodă, dacă nu este găsită se va trece la superclasă etc. Deoarece această căutarea este mare consumatoare de timp şi ar trebui derulată pentru fiecare apel de metodă, JVM, determină în avans pentru fiecare clasă o **tabelă a metodelor** (method table) care conţie lista tuturor semnăturilor de metode împreună cu argumentele cu care acestea sunt apelate. La apelul efectiv al unei metode JVM realizează o căutare numai în această tabelă.

Linia de cod `c1 = p1;` poartă denumirea de **atribuire polimorfică**. Într-o astfel de atribuire tipul sursei (`p1`) este diferit de tipul destinaţiei (`c1`). Atribuirea în sensul opus este însă imposibilă, adică nu se va putea scrie `p1 = c1;` deoarece nu orice coordonată este şi un punct (nu se poate aplica principiul substituţiei). Totuşi, aşă cum în cazul tipurilor simple putem face conversii cu forţare de tip, adică putem face dintr-un `double` un `int` prin codul:

```
double x = 12.345;
int i = (int) x;
```

şi în cazul obiectelor este posibilă conversia unei referinţe la obiect de la o clasă la o alta. O astfel de conversie este însă corectă numai contextul unei ierarhii şi se scrie astfel:

```
p1 = (Punct) c1;
```

Compilatorul verifică dacă valoarea stocată într-o variabilă de un anumit tip nu produce probleme (este de alt tip sau în afara domeniului permis). În cazul obiectelor, atunci când se atribuie referinţei de subclasă o superclasă, trecerea este oarecum echivalentă cu cea de la un domeniu mai îngust la unui mai larg, deoarece superclasa este mai generală decât subclasa. Atribuirea inversă este însă problematică deoare se încercă o trecere de la puţin la mai mult. Din acest motiv compilatorul trebuie anunţat explicit că urmează o conversie cu ajutorul operatorului de forţare de tip.

6

Excepții, tablouri în Java

Excepţii

- Surse de erori: date de intrare, probleme cu dispozitive fizice, cod Java neglijent;
- Excepţia este o condiţie de eroare ce modifică rularea normală a programului: se face o împărţire cu zero, se introduc date de intrare greşit etc. JAVA are un mecanism de tratare a acestor situaţii cu instrucţiunile **`try - catch`**;
- În Java eroarea se defineşte ca o situaţie anormală de funcţionare din care nu se mai poate reveni: JVM are o problemă internă de funcţionare sau rămâne fără RAM.

Există situaţii în care apar probleme în timpul execuţiei instrucţiunilor unui program ca urmare a unor erori. Câteva dintre aceste **erori tipice** sunt:
Introducerea eronată a datelor: de exemplu, se aşteptă introducerea de la tastatură a unui număr întreg - 173 - iar utilizatorul introduce un şir de caractere - 17w3 - deoarece atinge două taste simultan la introducerea întregului (pe "w" şi pe "3"); un alt exemplu ar fi scrierea incorectă a unui URL atunci când aplicaţia cere introducerea acestuia de la tastatură (în loc de http://www.utcluj.ro se scrie http;//www.utcluj.ro, adică pe locul lui ":" se scrie ";") situaţia în care pachetul de reţea din Java v-a semnala eroarea fără milă;
Erori datorate unor dispozitive hardware: de exemplu, în timpul tipăririi imprimata rămâne fără hârtie, server-ul de web ale cărui pagini le vizualizăm cade sau în timpul scrierii pe disc acesta rămâne fără spaţiu;
Erori de programare: de exemplu o secvenţă de instrucţiuni nu îşi face treaba corect, se face o împăţire cu zero, se încearcă referirea unui element de tablou inexistent etc.

În limbaje de nivel înalt tradiţionale erorile sunt tratate prin utilizarea unor coduri de eroare ce trebuie testate înainte de a continua execuţia programului. La nivel de pseudocod o astfel de aplicaţie are forma:

Tratarea erorilor în limbaje tradiţionale (C, Pascal)

```
cod_de_eroare1=0
grup de instrucţiuni1
   dacă (cod_de_eroare1 ≠ 0) atunci {
     instrucţiuni pentru tratarea erorii 1
   }
cod_de_eroare2=0
grup de instrucţiuni2
   dacă (cod_de_eroare2 ≠ 0 atunci {
     instrucţiuni pentru tratarea erorii 2
  }
```

Tratarea erorilor în Java

```
try {
   grup de instrucţiuni1
   grup de instrucţiuni2
}
catch(cod de eroare1) {
   instrucţiuni pentru tratarea erorii 1
}
catch(cod de eroare2) {
   instrucţiuni pentru tratarea erorii 2
}
```

Acest mod de programare a problemei erorilor intercalează instrucţiunile de program cu cele de tratare a erorilor. Problemele ce apar din această cauză sunt:

- programul nu este lizibil (se citeşte greu), se modifică greu, iar depanarea lui este dificilă;
- viteza aplicaţiei scade deoarece după fiecare grup de instrucţiuni care ar putea genera erori intervine testul prin care se decide între continuarea rulării aplicaţiei sau tratare a erorii.

În Java termenul de excepţie (exception) arată că, datorită unei probleme, o instrucţiune nu s-a putut executa în condiţii normale de către JVM. JVM detectează automat aceste situaţii oprind şi terminând imediat execuţia programului. Numele de excpţie vine de la faptul că deşi a apărut o problemă situaţia nu este frecventă (adică regula, este execuţia normală, iar eroare este situaţia excepţională). Limbajul Java separă detaliile de tratare a erorilor (`instrucţiuni pentru tratarea erorii`) de secvenţa de instrucţiuni care le-ar putea genera (`grup de instrucţiuni1`) codul devenid în acest lizibil. `try` (încearcă) însemnă că la nivel de cod se încercă ceva ce ar putea genera o excepţie, iar `catch` (prinde) însemnă că excepţia respectivă este tratată printr-un grup de instrucţiuni specifice. Fie următoare aplicaţie Java:

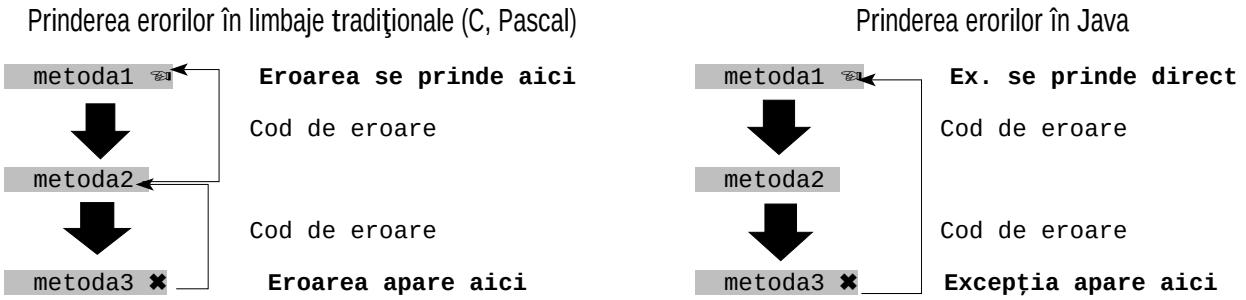
```
public class ExMat {
    public static void main(String[] args) {
        int a, b, c;
        for (int i = 0 ; i < 7 ; ++i ) {
            a = (int) ( Math.random() * 10 );
            b = (int) ( Math.random() * 4 );
            System.out.print( i + ") " + a + "/" + b  );
            c = a / b; // linia 8 - aici apare exceptia
            System.out.println( "=" + c );
        }
    }
}
```

Rezultatele afişate de aceasta la o rulare ar putea fi:

```
0) 7/2=3
1) 6/1=6
2) 9/3=3
3) 7/1=7
4) 5/2=2
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at ExMat.main(ExMat.java:8)
5) 8/0Process exited with exit code 1.
```

Corpul ciclului `for` se reia de cel mult 7 ori (`i = 0, 1, 2, ..., 6`), totuşi pentru i = 5 (la cea de a 6-a reluare) aplicaţia crapă la execuţia liniei 8 (`c = a / b`) cu mesajul de eroare "`java.lang.ArithmeticException: / by zero`".

În limbajele tradiţionale, întoarcerea unui cod de eroare nu implică şi obligativitatea testării acestuia. Fie schema următoare:



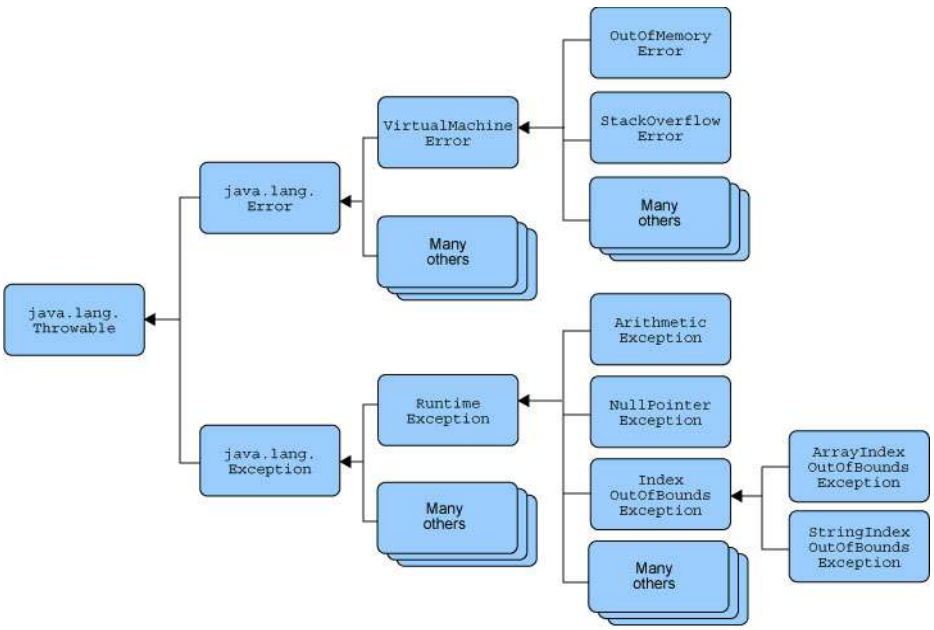
Aici avem `metoda1` care apelează `metoda2` şi care apelează `metoda3`. Într-un limbaj tradiţional, dacă eroarea apare în `metoda3` aceasta v-a fi luată în considerare numai după ce se revine din `metoda2` în `metoda1`. Aici apar două probleme:

- prima constă în faptul că eroarea nu este tratată atunci când apare ci atunci când există posibilitatea ca aceasta fie tratată deci, eroare şi codul de tratare a acesteia, sunt desincronizate;
- a doua problemă constă în volutariatul tratării acesteia, adică tratarea ei este lasată la latitudinea programatorului, aceasta nefiind obligatorie.

În Java, prin mecanismul de interceptare a excepţiilor, aceste probleme nu mai apar astfel:

- atunci când apare o excepţie, JVM transferă controlul direct proţiunii de cod care tratează excepţia;
- dacă o metodă a generat o excepţie (thrown an exception), aceasta nu se poate ignora, ea trebuie prinsă şi tratată undeva, altfel compilatorul semnalează eroarea (excepţie fac erorile din timpul execuţie aplicaţiei).

Clasificarea excepțiilor în Java



- Orice excepţie este un obiect creat folosind o subclasă a clasei de bază **`java.lang.Throwable`**;
- subclasa **`Error`** include erorile din care nu se mai poate reveni - programul se termină;
- subclasa **`Exception`** include erorile din care se poate reveni - corectarea situaţie execpţionale se face în rutina de tratare a erorii (exception handler)

În Java, excepţia este un obiect care se instanţiază pe baza unei subclase a clasei `java.lang.Throwable`. Ierarhia de clasă are două subclase mari: `Error` şi `Exception`. Subclasa `Error` corespunde unor situaţii de erori deosebit de grave, mediul rămâne fără memorie sau are o eroare internă. Programatorul nu poate face mare lucru în aceste situaţii, cel mult poate forţa terminarea aplicaţiei într-o manieră elegantă, în locul celei abrupte. Clasa `Exception` este cea pe care programatorul o utilizează uzual în situaţia în care programul nu se poate rula în condiţii normale. Aceasta are la rândul ei mai multe subclase dintre ele, `RuntimeException`, este utilizată des. O `RuntimeException` apare din cauza unei erori de programare, orice altă excepţie apare ca urmare a unor cauze externe care opresc programul din execuţie (de exemplu, datele de intrere nu sunt introduse corect). Excepţiile din subclasa `RuntimeException` sunt tratate diferit de celelalte de către compilator. Orice alte erori se prind şi se tratează după cum urmează.

Tratarea erorilor în Java

- secvenţa de instrucţiuni ce pot genera erori se cuprind într-un bloc **try**;
- pentru fiecare excepţie se foloseşte un bloc **catch** pentru tratarea ei;
- dacă este cazul, orice prelucrări finale se fac în blocul **finally**

```
try {
//cod care poate genera erori
}
catch(tipexceptie1 ob) {
//rutină pentru tratarea excepţiei 1
}
catch(tipexceptie2 ob) {
//rutină pentru tratarea excepţiei 2
}

. . .

finally {
//cod de executat înainte de terminarea
// blocului try
}
```

Instrucţiunile unei metode care urmează să fie monitorizate pentru excepţii trebuie cuprinse în blocul format dintre acoladele instrucţiunii `try`. Instrucţiunea `try` conţine un grup de instrucţiuni, numite şi bloc, ce urmează să fie executate. Dacă o problemă apare între instrucţiuni atunci se generează o excepţie (generarea are ca efect întreruperea execuţiei codului şi saltul la codul de tratare a excepţiei). Prinderea excepţiei generate se face folosind clauza `catch`, iar tratarea ei prin codul cuprins între acoladele respectivului `catch`. Prin `tipexceptie` se înţelege tipul excepţie care a generat eroarea. O instrucţiune `try` poate avea orice număr de clauze `catch`. Programatorul are posibilitatea ca o excepţie de un anumit tip să o prindă şi să o trateze prin rearucarea ei ca şi o nouă excepţie. O altă posibilitate ar fi ca excepţia să nu fie tratată la nivelul metodei în care aceasta a apărut. În această situaţie excepţia este pasată metodei apelante (metoda din care s-a apelat metoda în care a apărut excepţia). În general, dacă o excepţie nu este tratată în metoda apelată aceasta este pasată metodei apelante. Această procedură de propagare a excepţiei se repetă până când se ajunge la metoda `main`. Dacă nici aici excepţia nu este tratată atunci aplicaţia se termină brusc şi anormal. Excepţiile sunt obiecte generate automat de JVM în situaţii de eroare.

catch

Clauza `catch` se declară cu parametru, acesta specificând tipul de execpţie care va fi tratat. Tipul parametrului trebuie să fie obligatoriu `Throwable` sau o subclasă a lui. Când apare o excepţie toate clauzele `catch` asociate acestuia sunt parcurse, dacă se găseşte un parametru de `catch` de acelşi tip cu cel al obiectului excepţie generat în `try` se vor executa instrucţiunile corespunzătoare acelui bloc

`catch`.

throw

Utilizatorul poate genera manual excepţii prin folosirea instrucţiunii `throw` (ea se foloseşte pentru regenerarea unei excepţii în exemplu următor). Forma generală a instrucţiunii este **`throw`** `obiectThrowable`. Existe două metode pentru crearea unui obiect `Throwable`, prin folosirea unui parametru în `catch` sau prin folosirea operatorului `new` (de exemplu, `throw new InputMismatchException("*** Prea ... 1 numar real. ***");`). La atingerea acestei instrucţiuni codul se opreşte din execuţie, orice instrucţiuni următoare ne mai fiind executate, execuţia fiind transferată celui mai apropiat `try`. Aici se verifică dacă există un `catch` ce are aceaşi tip cu cel al excepţiei, dacă nu se trece la umătorul `try` etc.

throws

Dacă o metodă ar putea genera o excepţie pe care nu o tratează trebuie să specifice acest comportament către metoda apelantă. Pentru aceasta în declaraţia metodei se se foloseşte clauza `throws` urmată de o listă a tipurilor posibile de excepţii. Procedura este obligatorie pentru toate tipurile de excepţii mai puţin `Error` şi `RuntimeException`. Forma generală a declaraţie de metodă ce include clauza `throws` este:

```
tip nume_metodă(listă_de_parametri) throws listă_de_excepţii
{
     //corp metodă
}
```

finally

La generarea unei excepţii execuţia secvenţială a programului se întrerupe continuându-se cu un grup de salturi. În funcţie de anumite condiţii, este posibil să revenim prematur dintr-o metodă, această situaţie poate genera probleme uneori. Pentru păstratea continuităţii execuţie codului a fost creată instrucţiunea `finally`. Ea defineşte un bloc de instrucţiuni care se execută întotdeauna, indiferent dacă excepţia a fost sau nu prinsă, după un `try - catch.` Instrucţiunea este utilizată, tipic, pentru a face curăţenie după `try` (închiderea unor fişiere şi eliberarea unor resurse alocate înainte de revenirea din metodele generatoare de excepţii).

Este obligatoriu ca o instrucţiune `try` să aibă asociată o instrucţiune `catch` sau una `finally`.

Aplicaţia 1 - try, catch, throw, throws

Aplicaţia ce urmează este formată din două fişiere `Functie.java` şi `TestExceptii.java`. Ea trebuie să evalueze valoarea unei funcţii F(x) definită în metoda `F` din clasa `Functie`.

$$F(x) = \begin{cases} \frac{x}{x+2}, & x<1 \\ \sqrt{4.5-x}, & 1 \le x \le 5 \\ \frac{\sin(x)}{x}, & x>5 \end{cases}$$

Această metodă generează excepţie de tipul `ArithmeticException`. Excepţiile sunt generate deoarece pentru x = -2 expresia x/(x+2) nu se poate calcula, iar pentru x $\in$ (4.5, 5] radicalul are argumentul negativ. Metoda `citeste_real()` se foloseşte pentru a citi o valoare numerică reală (sau întreagă) de la tastatură. Reluarea citirii se face de cel mult 3 ori, după care aplicaţia se opreşte deoarece se generează o eroare ca nu este prinsă în aplicaţie. În această situaţie aplicaţie afişează starea stivei pentru a putea determina precis linia şi codul în care a apărut eroarea.

Clasa Functie.java:

```
import java.util.*;
public class Functie {
    public double F(double x) throws ArithmeticException{
        if (x < 1.) {
            if (x == -2.)
                throw new ArithmeticException("Impartire cu 0");
            return x/(x+2);
        }
        else if (x <= 5.) {
            if (x > 4.5)
                throw new ArithmeticException("Argument radical negativ.");
            return Math.sqrt(4.5-x);
        }
        else
            return Math.sin(x)/x;
    }

    public double citeste_real() throws InputMismatchException
    {
        Scanner in =  null;
        int contorErr = 0; //contor erori, pt. 3 aplicatia se opreste.
        double x;

        while(true) {
            in = new Scanner(System.in);
            try {
                x = in.nextDouble();
                break;  //daca nu sunt erori se iese din ciclu
            }
            catch (InputMismatchException e) {
                ++contorErr;
                 System.out.printf("Eroare(%d) la citire, trebuie sa introduci o
valoarea reala.\n", contorErr);
                if ( contorErr == 3 )
                   throw new InputMismatchException("*** Prea prost ca sa bage 1
numar real. ***");
            }
        }
        return x;
    }
}
```

Clasa TestExceptii.java:

```
public class TestExceptii {

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

```
        Functie F1 = new Functie();
        double x1, x2;

        System.out.print("x1=");
        x1 = F1.citeste_real();
        System.out.print("x2=");
        x2 = F1.citeste_real();

        for(double x=x1;x<=x2;++x) {
            try {
                System.out.printf("%6.3f %10.7f\n",x, F1.F(x));
            }
            catch (ArithmeticException e) {
                System.out.printf("%6.3f %s\n",x, e);
            }
        }
    }
}
```

Rezultate posibile:

```
x1=-7
x2=7
-7.000  1.4000000
-6.000  1.5000000
-5.000  1.6666667
-4.000  2.0000000
-3.000  3.0000000
-2.000 java.lang.ArithmeticException: Impartire cu 0
-1.000 -1.0000000
 0.000  0.0000000
 1.000  1.8708287
 2.000  1.5811388
 3.000  1.2247449
 4.000  0.7071068
 5.000 java.lang.ArithmeticException: Argument radical negativ.
 6.000 -0.0465692
 7.000  0.0938552
Process exited with exit code 0.
```

sau:

```
x1=werw
Eroare(1) la citire, trebuie sa introduci o valoarea reala.
werw
Eroare(2) la citire, trebuie sa introduci o valoarea reala.
wrwe
Eroare(3) la citire, trebuie sa introduci o valoarea reala.
Exception in thread "main" java.util.InputMismatchException: *** Prea prost ca sa
bage 1 numar real. ***
      at Functie.citeste_real(Functie.java:36)
      at TestExceptii.main(TestExceptii.java:8)
Process exited with exit code 1.
```

Se poate observa cum aplicaţia, deşi întâlneşte situaţii excepţionale, nu se termină brusc şi abrupt, ci ca urmare a tratării acestora, ajunge să se termine normal.

Propagare excepției codului apelant

- se foloseşte **throws** în declarația metodei;
- nu avem un bloc **catch** pentru excepție;
- este un **catch** în metoda apelantă care prinde excepția.

```
public double F(double x) throws
ArithmeticException {
 . . .
     throw new ArithmeticException("/ cu 0");
 . . .

    }

public static void main(String[] args) {
 . . .
for(double x=x1;x<=x2;++x) {
 try {
  System.out.printf("%6.3f %10.7f\n",x, F1.F(x));
 }
 catch (ArithmeticException e) {
  System.out.printf("%6.3f %s\n", x, e);
 }
}
```

În aplicaţia 1 se prezintă modul în care o excepţie apărută la nivelul metodei F este propagată în metoda `main`, de unde se realizează apelul lui F prin linia `F1.F(x)`. Metoda `main` poartă denumirea de cod apelant, iar metoda F de cod apelat. Deoarece metoda apelată nu prinde excepţia ci o lasă să se propage declaraţia acesteia conţie pe **`throws ArithmeticException`**. În general, dacă am dori ca şi metoda apelantă să realizeze propagarea excepţie mai departe şi declaraţia ei ar trebui să conţină pe **`throws ArithmeticException`**.

Conceptual, propagarea erorii în codul apelant se face atunci când la nivel local, în codul apelat, nu putem da o interpretare clară a problemei apărute.

Generarea manuală a unei excepții

- excepția se generează manual cu **throw new obExcepție();**
- în declarația metodei se folosește **throws**.

```
public double F(double x) throws
ArithmeticException {
 . . .
     throw new ArithmeticException("/ cu 0");
 . . .

    }
```

Generarea manuală a unei excepţii predefinite în ierahia de excepţii Java sau a unei definite de utilizator se face cu instrucţiune `throw`. Orice excepţie generată în acest fel trebuie pusă şi în clauza `throws` a metodei. Dacă se vor genera mai multe excepţii dintr-o metodă, toate numele respectivelor excepţii trebuie scrise în listă, separate prin virgulă.

Există însă şi excepţii care nu trebuie puse în listă şi anume `Error`, `RuntimeException`. Motivul pentru care aceste nu trebuie scrise explicit constă în numărul mare de situaţii (JVM rămâne fără memorie, împărţire cu zero, indice de tablou ilegal etc.) în care ele pot să apară în diferite porţiuni de cod. Aceasta ar implica scrierea lor în fiecare declaraţie de metodă pentru fiecare porţiune de cod. Conform celor spuse în loc de `public double F(double x) throws ArithmeticException` se poate scrie mai scurt doar `public double F(double x)` deoarece **`ArithmeticException`** este o subclasă a lui `RuntimeException`.

Generarea manuală a unei excepţii permite ca la prinderea unei excepţii să se genereze în locul acesteia o alta. Un cod tipic este:

```
catch (exceptie1 e) {
     throw new exceptie2( ... );
}
```

Aici s-a prins excepţia `excepție1`, însă în codul de tratare a excepţiei se generează o alta, şi anume `excepție2`.

Crearea unei excepții noi

! o excepţie nouă se crează prin extinderea clasei **Exception**.

```
public class ExceptiiAritmetice extends
Exception{
    private String mesaj;

    public ExceptiiAritmetice(String m) {
        super(m);
        mesaj = m;
    }

    public String toString() {
        return "\nExceptie: " + mesaj;
    }
```

Programatorul poate să-şi creeze propriile excepţii prin extinderea clasei `Exception`. La crearea unei subclase a lui `Exception` programatorul are deja o nouă clasă funcţională în care nu trebuie să implementeze nimic. `Exception` nu defineşte metode noi, însă le moşteneşte de la `Throwable`. Nu este cazul extinderii clasei `RuntimeException` deoarece ea este creată pentru a lucra cu erori comune ce nu trebuie tratate explicit mai mult decât sunt deja. Extinderea excepţiilor are rost în situaţia unor particularităţi specifice ale aplicaţiei.

Aplicația 2 - extindere Exception

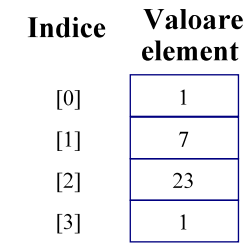
Aplicaţia care urmează perintă modul în care se extinde clasă `Exception` la clasa `ExceptiiAritmetice`.

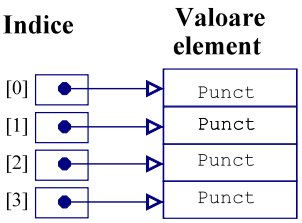
```
public class ExceptiiAritmetice extends
Exception{
  private String mesaj;

  public ExceptiiAritmetice(String m) {
      super(m);
      mesaj = m;
  }

  public String toString() {
      return "\nExceptie: " + mesaj;
  }
}
```

```
public class TestEx2 {
   public static int cat(int a, int b)
throws ExceptiiAritmetice {
            if (b == 0) throw new
ExceptiiAritmetice("Imparire cu zero!");
      return a / b;
  }

  public static void main(String[] args)
{
      int a, b, c;
      a=1;
      b=0;
      System.out.print(a + "/" + b  );
      try {
          c = (int)cat(a,b);
         System.out.println( "=" + c  );
        }
      catch (ExceptiiAritmetice e) {
          System.out.println(e);
      }
  }
}
```

Tablouri (arrays)

- tabloul este o colecţie de elemente de acelaşi tip;
- un element poate stoca o singură valoare;
- elementele se identifică unic printr-un număr întreg numit indice;
- tipul elementelor poate fi simplu sau obiect;
- numărul de elemente de tablou se fixează în momentul creării acestuia.





Tabloul este o colecţie de date, numite şi elemente ale tabloului, identice ca tip - tipul se mai numeşte şi tip de bază - ce pot fi identificate unic prin indice. Numărul de dimensiuni ale tabloului depinde de limbaj, dar în general, nu este limitat. Se zice că un tablou este multidimensuional dacă foloseşt mai mulţi indici pentru accesarea unui element de tablou.

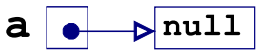
Mai sus se prezintă modul de reprezentate a două tablouri, unul de întregi (tipul de bază este int) şi unul de obiecte (tipul de bază este Coordonate).

O variabilă de tip primitiv - un scalar - poate fi privită ca un tablou cu dimensiunea zero. Un tablou cu o singură dimensiune este cunoscut sub numele de "vector", iar unul cu două dimensiuni sub denumirea de "matrice". În majoritatea limbajelor imperative, o referinţă la un element de tablou se scrie sub forma a[i,j] unde, a este numele tabloului, iar i şi j sunt indexuri. Elementele tabloului sunt de obicei stocate în locaţii de memorie consecutive. Limbajele diferă de obicei prin metoda de stocare a datelor pe rânduri continue sau pe coloane continue.

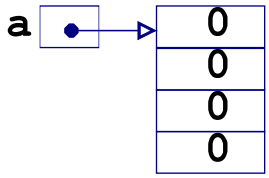
Tablourile sunt folosite pentru stocarea unui grup de valori care priimesc un singur nume şi care se vor accesa într-o ordine imprevizibilă (de exemplu, o structură de dată numită listă se pretează mult mai bine pentru stocarea unor valori care se accesează secvenţial).

Etapele lucrului cu tablouri

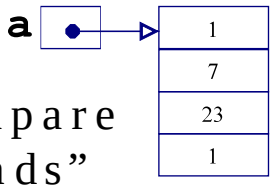
- declararea tabloului:
  `int []a;`



- crearea unui obiect tablou:
  `a = new int[4];`



- iniţializarea tabloului:
  `int[] a = {1, 7, 23, 1};`



- indicele ∈ [0, nr_elem-1] > altfel apare excepţia “array index out of bounds”

- accesul la un element:
  `a[indice]`

Pentru a putea utiliza un tablou cu tipul de bază dintre cele primitive trebuie să:

- declarăm o variabilă de tipul tablou având `nume_tablou`, ce v-a stoca o referinţă către tablou prin: `tip [] nume_tablou;` sau `tip nume_tablou [];` în acest moment `nume_tablou` se iniţializează cu valoarea specială `null`; această valoare se modifică la adresa tabloului după utilizarea operatorului `new`;
- creăm tabloul folosind operatorul `new` specificând lungimea acestuia (numărul maxim de elemente pe care îl poate stoca); lungimea trebuie să fie o valoare întreagă de tipul `int` (constantă sau expresie care se calculează în momentul rulării aplicaţiei)
- iniţializăm tabloul cu valori diferite de cele cu care acesta se iniţializaeză implicit de către Java.

Valorile de iniţializare implicită pentru tablouri:

`char`	`boolean`	`byte`, `short`, `int`, `long`	`float`, `double`
’\u0000’ - Unicode 0000 (Java foloseşte codificarea Unicode)	false	0	0.0

Deşi un tabloul este un obiect, nu există o clasă de tipul tablou. Java creează automat o clasă pentru fiecare tablou de tipuri primitive sau de clase. Pentru a afla numărul de elemente dintr-un tablou folosim notaţia `nume_tablou .length .`

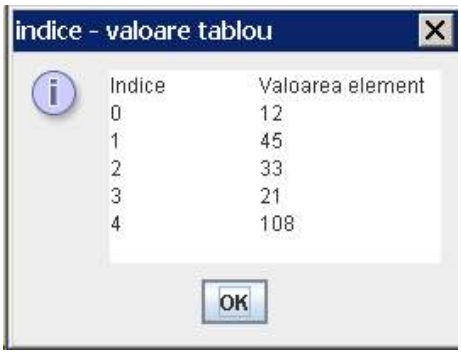
Aplicația 1 - tablou de tipul int, citire şi afişare

Aplicaţia următoare exemplifică modul de citire şi de afişare a elemtelor unui taboul. Tabloul are numele a şi numărul maxim de elemente fixat la 5. Valorile sunt citite de la tastatură iar afişarea se face prin două metode.

```
import java.util.Scanner;
import javax.swing.*;

public class TablouPrimitive {

    public static void main(String[] args) {
        final int NRELEM = 5; //lungimea tabloului
        int [] a; //declararea tabloului
        Scanner intrare;
        String iesire = "Indice\tValoarea element\n";

        intrare = new Scanner(System.in);
        a = new int[NRELEM];  //crearea tabloului;

        //citire elemente tablou
        for(int i = 0; i < a.length;++i) {
            System.out.print("a[" + i +"] = ");
            a[i]= intrare.nextInt();

            //pt. cea de a 2-a metoda de afisare
            iesire+= i +"\t"+ a[i] + "\n";
        }

        //metoda 1: afisare simpla de elemente tablou
        for(int i = 0; i < a.length;++i)
            System.out.println(a[i]);

        //metoda 2: afisare cu fereastra de dialog
        JTextArea fereastraiesire = new JTextArea();
        fereastraiesire.setText(iesire);
           JOptionPane.showMessageDialog(null,fereastraiesire,"indice - valoare
tablou", JOptionPane.INFORMATION_MESSAGE);
      }
}
```

Rezultate:

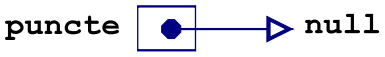
```
a[0] = 12
a[1] = 45
a[2] = 33
a[3] = 21
a[4] = 108
12
45
33
21
108
```



Etapele lucrului cu tablouri de obiecte

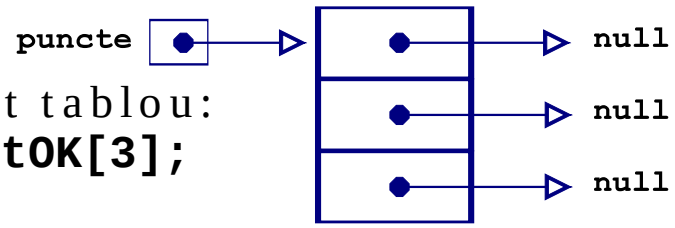
! declararea tabloului:

```
PunctOK puncte[];
//sau PunctOK[] puncte;
```



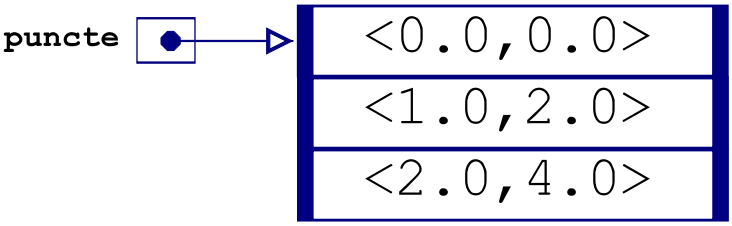
! crearea unui obiect tablou:

```
puncte = new PunctOK[3];
```



! iniţializarea obiectelor din tablou:

```
puncte[i] = new PunctOK(i,2*i);
```



În situaţia lucrului cu tablouri de obiecte, etapele sunt indentice cu cele descrise la tablourile de primitive, cu excecţia că aici iniţializarea elementelor de tablou este obligatorie şi nu opţională.

Aplicaţia 2 - tablou de obiecte, iniţializare şi afişare

Aplicaţia are la bază clasa `PuncteOK`. Se creează un tablou de obiecte `PuncteOK` cu numele `puncte`. Iniţializarea obiectelor din tablou se face în ciclul `for` cu linia `puncte[i] = new PunctOK(i,2*i);`. Rezultatele afişate se obţin prin iniţializarea fiecărui obiect `PuncteOK` cu x = i şi y =2*i.

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

        //declarare tablou
        PunctOK puncte[];
```

```
        //creare tablou de obiecte
        puncte = new PunctOK[3];

        for(int i = 0; i < puncte.length;++i) {
            //initializarea obiectelor din tablou
            puncte[i] = new PunctOK(i,2*i);
        }

        //afisarea elementelor de tablou
        for(int i = 0; i < puncte.length;++i)
            System.out.println(puncte[i]);
    }
}
```

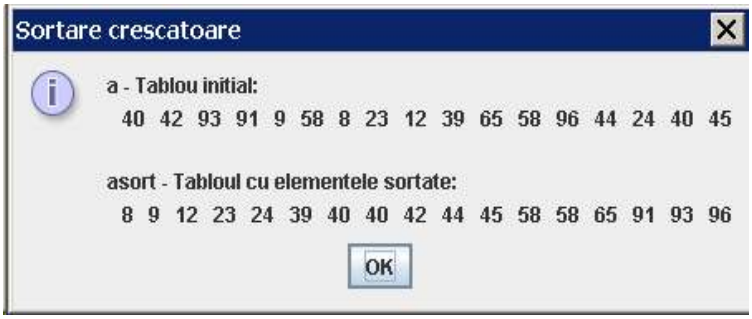
Rezultate:

```
<0.0,0.0>
<1.0,2.0>
<2.0,4.0>
```

Aplicația 3 - sortarea crescătoare a elementelor unui tablou

```
// Sortarea crescatoare a unui tablou de intregi
import javax.swing.*;

public class sortTab {
    int a [];    //tabloul
    int asort[]; //clona
    int n;       //numarul de elemete de tablou

     sortTab(int n) {
        this.n = n;

        //tabloul initial
        a = new int[n];
        //tabloul sortat
        asort = new int [n];

        //generam aleator n numere de la 0 la 100
        //de tipul int si le stocam in a
        for(int i=0; i<n ; ++i)
            a[i] = (int)(100*Math.random());

         //copiem tabloul a in asort
         try {
            System.arraycopy(a,0,asort,0,n);
         }
         catch(ArrayStoreException e) {
             System.out.println(e);
         }
     }

     //sortarea
     public void bubbleSort( )
     {
        boolean esteinterschimbare = true;

        while (esteinterschimbare) {
           esteinterschimbare = false;
```

```
            for ( int i = 0; i < asort.length-1; ++i ) {
               if ( asort[i] > asort[i+1] ) {
                     swap(asort,i,i+1);
                     esteinterschimbare = true;
               }
            }
         }
      }

    //interschimbarea a 2 elemente de tablou
    private void swap( int tablou[], int i, int j )
    {
        int aux;

        aux = tablou[i];
        tablou[i] = tablou[j];
        tablou[j] = aux;
    }

    //afisare tablou initial
    public String afisare(String txt)
    {
        String rez;
        rez = txt;
        for (int i=0; i<a.length; ++i )
           rez += "   " + a[ i ];

        return rez;
    }

    //afisarea tablou sortat
    public String afisaresort(String txt)
    {
        String rez;
        rez = txt;
        for (int i=0; i<asort.length; ++i )
           rez += "   " + asort[ i ];

        return rez;
    }


   public static void main(String
args[])
   {
      sortTab st;
      String rez;

      st = new sortTab(17);

         rez  =  st.afisare("a  -
Tablou initial:\n");
      st.bubbleSort();  // sortare tablou
      rez += st.afisaresort("\n\nasort - Tabloul cu elementele sortate:\n");


      JTextArea ferRez = new JTextArea();
      ferRez .setText( rez );
                        JOptionPane.showMessageDialog(null,    rez,    "Sortare
crescatoare",JOptionPane.INFORMATION_MESSAGE);
   }

}
```



Excepții întâlnite la tablouri

! utilizarea unei indice în afara domeniului permis generează excepția ArrayIndexOutOfBoundsException:

```
int [] a = new int[7];
System.out.println(a[13]);
```

(în exemplul anterior indicele poate fi în domeniu 0 - 6 = **7**-1)

! accesarea unor membri unui element obiect care încă nu a fost inițializat generează excepția NullPointerException:

```
puncte = new PunctOK[3];
System.out.println(puncte[0].x());
```

(în exemplul anterior $0 \in [0, 2]$, dar nu s-a folosit **new punct[0]** pentru crearea unui obiect **PunctOK**)

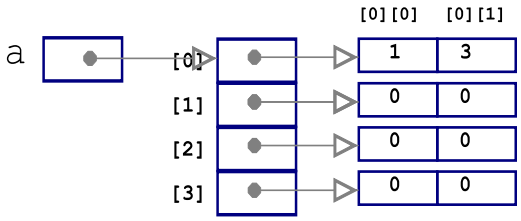
Tablouri multidimensionale

- un tablou multidimensional este un tablou de tablouri;
- pentru accesarea elementelor de folosesc mai mulţi indici;
- declararea şi crearea unui tablou cu 2 dimensiuni (matrice):

```
tip [][] numetablou = new tip[nl][nc];

int [][] a = new int[4][2];
a[0][0] = 1;
a[0][1]=3;
```

- iniţializare:

```
int[][] a =
    {
        {1,3},
        {0,0},
        {0,0},
        {0,0}
    };
```



- declarare şi iniţializare fără specuficarea numărului de coloane:

```
tip [][] numetablou = new tip[nl][];

numetablou[il] = new tip[nc];
```

Aplicația 1 - crearea unui tablou cu număr “variabil” de coloane

În aplicaţia care urmează lungimea liniilor din matrice este diferită, adică variază de la linie la linie.

```
public class TabMulti {
    public static void main(String[] args) {
        int tab2d [][] = new int [4][];
        tab2d[0] = new int[5];
        tab2d[1] = new int[2];
        tab2d[2] = new int[4];
        tab2d[3] = new int[7];

        int k = 0;

        for(int i=0; i < tab2d.length; ++i)
            for(int j=0; j < tab2d[i].length; ++j)
                tab2d[i][j] = k++;


        for(int i=0; i < tab2d.length; ++i) {
            for(int j=0; j < tab2d[i].length; ++j)
                System.out.print(tab2d[i][j]+" ");
            System.out.println();
        }
    }
}
```

Rezultate:

```
0 1 2 3 4
5 6
7 8 9 10
11 12 13 14 15 16 17
```