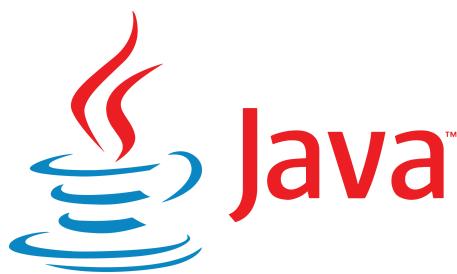


Introducere în Java



Java este un limbaj de programare utilizat pentru a dezvolta o gamă largă de aplicații, de la aplicații mobile la aplicații web și software de server.

Este un limbaj de programare orientat pe obiecte, ceea ce înseamnă că promovează organizarea software-ului în colecții modulare și reutilizabile de cod numite **obiecte**. Acest lucru ajută la simplificarea dezvoltării și la îmbunătățirea reutilizării codului.

Java este cunoscut pentru portabilitatea sa, adică aplicațiile scrise în Java pot rula pe aproape orice tip de computer, datorită lui Java Virtual Machine (JVM). Aceasta este un mediu de execuție ce interpretează codul Java.

Exemple concrete de aplicații construite folosind Java

Majoritatea aplicațiilor pentru dispozitivele Android sunt dezvoltate folosind Java:

- **WhatsApp** – Java este folosit în infrastructura de server pentru a gestiona mesageria instant și conexiunile în timp real între utilizatori;



- **Spotify** – Java este utilizat pentru backend, gestionând serviciile de streaming, logica de recomandare a muzicii, și interacțiunile utilizatorilor;



- **Instagram** – deși folosește o varietate de tehnologii pentru serverele sale, Java poate fi parte din soluțiile pentru gestionarea volumelor mari de date și interacțiunilor utilizatorilor, cum ar fi: încărcarea și descărcarea imaginilor, gestionarea profilurilor și interacțiunea cu postările.



Deși nu este principalul limbaj pentru dezvoltarea jocurilor, Java a fost folosit pentru crearea unor jocuri populare pe PC, cum ar fi **Minecraft**. Este unul dintre cele mai populare jocuri din lume, inițial creat în Java.



Multe aplicații și servicii bazate pe cloud sunt scrise în Java. Platforme precum **Google Cloud Platform** și **Amazon Web Services** folosesc Java pentru a permite dezvoltatorilor să creeze, găzduiască și să administreze aplicații în cloud.



Google Cloud Platform

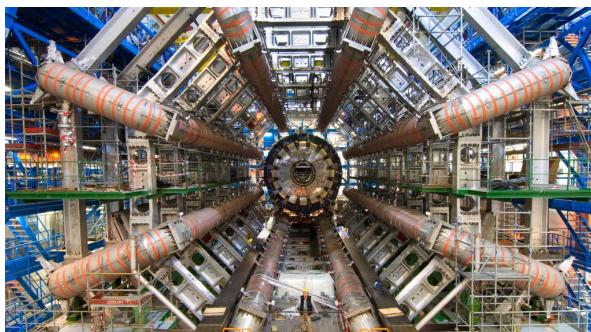


Java este utilizat și în software științific pentru simulări, calcul numeric, și analiza datelor. De exemplu, în **bioinformatică** pentru analiza secvențelor genetice și în **fizica particulelor** pentru analiza coliziunilor.

În bioinformatică, Java este folosit pentru a dezvolta software care ajută la analiza secvențelor genetice. Aceasta implică procesarea secvențelor de ADN sau ARN pentru a identifica gene, mutații și pentru a înțelege relațiile evolutive între specii. Programele pot, de exemplu, să compare secvențe genetice pentru a identifica regiuni care sunt importante pentru funcția biologică sau să detecteze variante genetice care pot influența predispoziția la boli.



În fizica particulelor, Java este utilizat pentru simularea și analiza coliziunilor de particule, cum ar fi cele care au loc în acceleratoare de particule precum **Large Hadron Collider** de la CERN. Aceste analize ajută la înțelegerea structurii fundamentale a materiei și a forțelor care guvernează universul. Software-ul poate calcula traекторii ale particulelor, analiza rezultatele coliziunilor și ajuta la testarea teoriilor fizice.



Alte lucruri despre Java...

"Write once, run anywhere" – codul Java poate rula pe orice dispozitiv care are JVM instalat, indiferent de arhitectura hardware sau sistemul de operare. Acest lucru a permis

ca Java să devină un limbaj extrem de popular pentru dezvoltarea de aplicații cross-platform.

Java rulează pe peste 3 miliarde de dispozitive în întreaga lume, incluzând telefoane mobile, PC-uri, sisteme de navigație și televizoare. Este omniprezent în tehnologia pe care o folosim zi de zi.



Java a fost numit după o insulă indoneziană bogată în cafea (cafea Java), deoarece creatorii săi consumau o cantitate mare de cafea în timp ce lucrau la dezvoltarea limbajului. Acest lucru a influențat și logo-ul oficial Java, care include o ceașcă de cafea.



Java este adesea limbajul de alegere pentru cursurile introductive de programare în universități datorită sintaxei sale clare și modelului său de gestionare automată a memoriei. Acest lucru îl face un instrument pentru învățarea conceptelor fundamentale ale programării orientate pe obiecte.

Java aplică restricții asupra a ceea ce poate face codul rulat într-un mediu de aplicație (oferă un model de securitate la nivel de **sandbox**). Aceste restricții sunt impuse prin reguli și politici de securitate care sunt definite de către dezvoltatori sau administratorii sistemului.

Sandbox este un termen folosit pentru a descrie un mediu de operare controlat și izolat în care codul este rulat. Acest mediu izolat restricționează accesul codului la resursele sistemului gazdă (cum ar fi fișierele și rețeaua), limitând astfel posibilele daune pe care codul rău intenționat le-ar putea provoca.

Exemple de Restricții de Securitate

1. Accesul la fișiere:

- **Problemă** – fără restricții, un cod rău intenționat ar putea citi sau modifica fișierele de pe dispozitiv, ducând la pierderea sau coruperea datelor sensibile.

- **Soluția Java** – Java permite stabilirea permisiunilor astfel încât anumite aplicații să fie restricționate să nu acceseze fișierele sistemului gazdă. Acest lucru previne scurgerile de informații confidențiale și alte riscuri de securitate.

2. Conexiuni de rețea:

- **Problemă** – aplicațiile pot fi folosite pentru a lansa atacuri în rețea sau pentru a comunica cu servere malicioase dacă nu sunt restricționate.
- **Soluția Java** – se pot seta restricții care împiedică aplicațiile să inițieze conexiuni de rețea sau să accepte conexiuni de la adrese IP nesigure sau necunoscute. Aceasta este o măsură importantă pentru protecția împotriva atacurilor de rețea.

Sintaxa Java

```
package _1.HelloWorld;

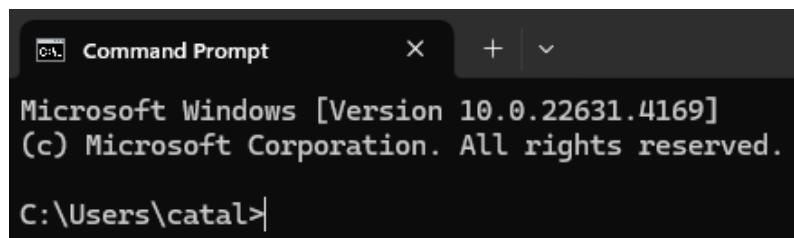
public class HelloWorld {    ▲ Catalin Catalan

    public static void main(String[] args) {    ▲ Catalin Catalan
        System.out.println("Hello world, this is a java program");
    }
}
```

Metoda **main** este punctul de intrare pentru orice aplicație Java standard. Când scrii o aplicație Java, aceasta este metoda pe care sistemul de execuție Java o apelează pentru a începe execuția programului tău.

Iată o explicație detaliată a sintaxei:

- **public** – este un modificador de acces și face ca metoda să fie accesibilă din orice altă clasă.
- **static** – cuvântul cheie **static** indică faptul că metoda poate fi apelată fără a crea o instanță a clasei în care se află.
- **void** – este tipul de returnare al metodei și indică faptul că metoda nu returnează nicio valoare.
- **main** – este numele metodei. Numele *main* este special pentru Java deoarece este metoda pe care JVM o caută ca punct de start al aplicației.
- **String[] args** – parametrii metodei, **args** reprezintă o matrice (array) de siruri de caractere (String), care este folosită pentru a primi argumente de la linia de comandă când programul este executat din command prompt.



- **System.out.println("Hello world, this is a java program")** - este o instrucțiune care afișează textul "Hello world, this is a java program" în consolă. **System.out** este un flux de ieșire standard, și **println** este o metodă care scrie pe acest flux, adăugând la sfârșit un caracter de sfârșit de linie (new line).

The screenshot shows a terminal window titled "Run" with the session name "HelloWorld". The terminal displays the command "C:\Users\catal\jdk\openjdk-21.0.1\bin\java.exe" followed by the output "Hello world, this is a java program". At the bottom, it shows "Process finished with exit code 0".

Comentariile în Java

Comentariile în Java sunt folosite pentru a adăuga note, explicații sau pentru a dezactiva părți din cod care nu sunt necesare la un moment dat, fără a fi executate de către compilator. Există trei tipuri principale de comentarii în Java:

1. Comentarii pe o singură linie

Acstea încep cu două bare oblice (//) și se întind până la sfârșitul liniei. Tot textul de după // pe acea linie este considerat comentariu și este ignorat de compilator.

```
public static void main(String[] args) { // Catalin Catalan *
    // Acesta este un comentariu pe o singură linie
    int number = 10; // Acesta este un comentariu pe o singură linie după o instrucțiune
}
```

2. Comentarii pe mai multe linii

Acste comentarii încep cu /* și se termină cu */. Orice text cuprins între aceste două simboluri este tratat ca un comentariu, indiferent de numărul de linii pe care le ocupă. Acestea sunt utile pentru comentarii mai lungi sau pentru a dezactiva temporar blocuri de cod.

```
public static void main(String[] args) { // Catalin Catalan *
    /* Acesta este un comentariu
    care se întinde pe mai multe linii
    și poate ocupa cât de mult spațiu este necesar. */
}
```

3. Comentarii de documentație

Cunoscute și sub numele de Javadoc, aceste comentarii sunt folosite pentru a genera documentație HTML pentru codul Java. Ele încep cu `/**` și se termină cu `*/`. Aceste comentarii pot include taguri care descriu funcțiile, parametrii, valoarea returnată și alte aspecte ale codului.

```
/**  
 * Aceasta este o metodă care returnează suma a două numere.  
 *  
 * @param a Primul număr întreg.  
 * @param b Al doilea număr întreg.  
 * @return Suma numerelor a și b.  
 */  
public static int add(int a, int b) { new *  
|    return a + b;  
}
```

Javadoc-urile sunt utile în proiectele mari sau când codul va fi folosit ca o bibliotecă, deoarece permit altor developeri să înțeleagă rapid și eficient funcționalitățile oferite de clasele, metodele și variabilele descrise.

Folosirea intelligentă a comentariilor poate face codul mult mai ușor de înțeles și de întreținut, ajutând atât autorul codului, cât și pe ceilalți developeri care lucrează pe același proiect sau care utilizează bibliotecile de cod.

Variabile și tipuri de dată

Variabilele sunt elemente fundamentale în orice limbaj de programare, inclusiv în Java, care permit stocarea datelor care pot fi modificate în timpul execuției unui program. O variabilă are un nume unic în contextul său, care este folosit pentru a referi valoarea pe care o stochează.

Java este un limbaj de programare puternic tipizat, ceea ce înseamnă că fiecare variabilă trebuie să fie declarată cu un **tip de date specific** înainte de a fi utilizată.

În Java, definirea unei variabile urmează reguli specifice pentru a asigura claritatea și funcționarea corectă a codului. De asemenea, există restricții privind numele pe care le poți folosi pentru variabile, pentru a evita conflictele cu sintaxa limbajului.

Reguli pentru definirea variabilelor

Tipul de Date – fiecare variabilă trebuie să aibă un tip de date specificat, care indică natura datelor pe care le poate stoca (de exemplu: int, double, String).

Numele Variabilei

- Trebuie să înceapă cu o literă (a-z sau A-Z), un underscore (_) sau un semn dolar (\$).
- După primul caracter, numele poate include litere, cifre (0-9), underscore-uri sau semne dolar.
- Numele nu trebuie să fie un cuvânt rezervat (cuvânt cheie) din Java.

Case Sensitivity – Java este sensibil la majuscule și minuscule, ceea ce înseamnă că **variable**, **Variable** și **VARIABLE** sunt considerate identificatori diferiți.

Inițializarea variabilei – o variabilă poate fi inițializată (își poate atribui o valoare) în momentul declarației. Inițializarea nu este obligatorie, dar este o practică bună, mai ales pentru variabilele locale.

Example

```
int myNum = 5;           // Integer (whole number)
float myFloatNum = 5.99f; // Floating point number
char myLetter = 'D';     // Character
boolean myBool = true;   // Boolean
String myText = "Hello"; // String
```

Cuvinte cheie în Java

Cuvintele cheie sunt termeni rezervați în Java care au un înțeles special pentru compilator și, prin urmare, **nu** pot fi folosite ca nume de variabile. Acestea sunt folosite pentru a defini structura și sintaxa programului. Iată câteva exemple de cuvinte cheie în Java: class, public, static, void, if else, while, break, continue, etc.

Constante

În Java, constantele sunt valori care nu pot fi schimbată după ce au fost inițializate. Constanțele sunt adesea utilizate pentru a oferi un nume semnificativ unor valori fixe care apar în tot codul, îmbunătățind astfel lizibilitatea și ușurința de întreținere a acestuia.

Pentru a declara o constantă în Java, se folosește cuvântul cheie **final**. O constantă trebuie să fie inițializată în momentul declarației sau în constructorul clasei dacă este o variabilă de instanță. Odată inițializată, valoarea ei nu poate fi modificată.

Prin convenție, numele constanțelor sunt scrise cu majuscule și cuvintele sunt separate prin underscore (de exemplu: MAX_HEIGHT), pentru a le diferenția ușor de variabilele obișnuite care sunt scrise în camelCase.

```
java
```

```
final int MAX_USERS = 100;  
final double PI = 3.14159;
```

Tipurile de date din Java se împart în două categorii mari: **tipuri primitive** și **tipuri de referință** (sau numite și **tipuri obiect**).

1. Tipuri primitive

Tipurile primitive reprezintă valori simple și sunt stocate direct în memorie. Java definește 8 tipuri primitive:

- **boolean** – reprezintă valori de adevăr (true sau false);
- **byte** – un tip de date întreg care ocupă 8 biți de memorie. Util pentru economisirea memoriei în array-uri mari;
- **short** – un tip de date întreg care ocupă 16 biți;
- **int** – cel mai frecvent utilizat tip de date întreg, care ocupă 32 de biți;
- **long** – un tip de date întreg extins, care ocupă 64 de biți;
- **float** – un tip de date cu punct flotant pentru valori reale, care ocupă 32 de biți;
- **double** – un tip de date cu punct flotant pentru valori reale, care ocupă 64 de biți și oferă o precizie mai mare decât float;
- **char** – reprezintă un caracter și ocupă 16 biți, fiind utilizat pentru stocarea caracterelor.

2. Tipuri de referință (tipuri obiect)

Tipurile de referință includ clase, interfețe și array-uri (tablouri/matrice). Acestea stochează referințe, adică adrese la locațiile reale în memorie unde sunt păstrate obiectele.

În Java, tipurile de date primitive, cum ar fi **int**, sunt simple și eficiente în termeni de consum de memorie și performanță. În contrast, tipurile de referință, cum sunt **String** și clasa **Integer** (echivalentul obiect pentru int), oferă funcționalități suplimentare, cum ar fi metode pentru manipularea datelor și interacțiunea cu colecțiile de obiecte.

String (tip de referință)

String este probabil cel mai utilizat tip de referință în Java și reprezintă o secvență de caractere. Este mai mult decât un simplu array de caractere (**char[]**); oferă numeroase metode pentru manipularea sirurilor de caractere, ceea ce îl face extrem de versatil.

```
public static void main(String[] args) { // Catalin Catalan *
    String name = "John Doe";
    System.out.println("Name: " + name);
    System.out.println("Length of name: " + name.length()); // Lungimea sirului
    System.out.println("Name in uppercase: " + name.toUpperCase()); // Convertirea la majuscule
}
```

Integer (tip de referință)

Integer este clasa ambalaj (wrapper class) pentru tipul de date primitiv **int**. Aceasta permite utilizarea int-urilor în colecții generice, cum ar fi **ArrayList**, care nu pot conține tipuri primitive. Clasa Integer oferă și metode utilitare, cum ar fi conversia între diferite baze numerice sau compararea valorilor.

```
public static void main(String[] args) { // Catalin Catalan *
    Integer myInteger = 10; // autoboxing din int în Integer
    System.out.println("Integer: " + myInteger);
    System.out.println("Double value: " + myInteger.doubleValue()); // Convertirea la double
    System.out.println("Binary representation: " + Integer.toBinaryString(myInteger)); // Reprezentarea binară
}
```

Avantaje ale tipurilor de referință față de tipurile primitive

- **Metode utilitare** – clasele wrapper oferă metode pentru manipularea și conversia datelor, cum ar fi `Integer.toBinaryString(int)` sau `String.toUpperCase()`.
- **Utilizare în colecții generice** – tipurile primitive nu pot fi folosite direct în colecții generice, cum ar fi `ArrayList`. Clasele wrapper permit stocarea valorilor primitive în astfel de colecții.
- **Permit valori null** – tipurile de referință pot reprezenta absența unei valori prin **null**, pe când tipurile primitive vor avea întotdeauna o valoare implicită (de exemplu, **0** pentru `int`).
- **Polimorfism** – tipurile de referință pot fi utilizate în contexte polimorfice, unde comportamentul poate varia în funcție de tipul real al obiectului.

Type casting

Type casting în Java este procesul prin care o variabilă de un tip de date este convertită într-o altă variabilă de un alt tip de date. Java suportă două tipuri de casting: **implicit** (widening conversion) și **explicit** (narrowing conversion).

1. Widening casting (implicit)

Widening casting (denumit și upcasting) este automat și se întâmplă când datele unei variabile cu un tip de date mai puțin cuprinzător sunt asignate unei variabile de un tip de

date mai cuprindător. De exemplu, convertirea de la **int** la **long**, **float** sau **double** este sigură și se face automat de către Java fără pierdere de informație.

```
public static void main(String[] args) { // Catalin Catalan *
    int myInt = 100;
    double myDouble = myInt; // Automat, fără pierdere de informatie

    System.out.println("Valoarea int: " + myInt); // Afisează 100
    System.out.println("Valoarea double: " + myDouble); // Afisează 100.0
}
```

2. Narrowing casting (explicit)

Narrowing casting necesită specificarea explicită a tipului în care dorim să convertim variabila. Aceasta se întâmplă când se asignează o variabilă de un tip de date mai cuprindător unei variabile de un tip mai puțin cuprindător, cum ar fi de la **double** la **int**. Acest tip de casting poate duce la pierdere de informație și trebuie gestionat cu precauție.

```
public static void main(String[] args) { // Catalin Catalan *
    double myDouble = 9.78;
    int myInt = (int) myDouble; // Explicit, cu posibilă pierdere de informatie

    System.out.println("Valoarea double: " + myDouble); // Afisează 9.78
    System.out.println("Valoarea int: " + myInt); // Afisează 9
}
```

Exemple de domenii și aplicații unde precizia numerică și numărul de zecimale sunt cruciale:

- Măsurarea precisă a pozițiilor geografice este esențială pentru cartografiere, navigație și gestionarea resurselor terestre. O eroare de calcul poate duce la interpretări greșite ale datelor spațiale.
- În dozarea medicamentelor, radioterapie și diagnosticare, precizia calculelor poate afecta direct sănătatea și siguranța pacienților. Erorile pot duce la tratamente ineficiente sau chiar periculoase.
- Experimentele și calculele în domeniul fizicii cuantice necesită o precizie extrem de mare pentru a testa teorii științifice și a observa fenomene subtile care apar la scară atomică și subatomică.

Operatori Java

Java suportă o varietate largă de operatori care permit manipularea și compararea datelor. Acești operatori pot fi împărțiți în mai multe categorii, fiecare având un rol specific în operarea cu variabile și valori. Iată o descriere succintă a fiecărei categorii de operatori, completată cu exemple relevante pentru fiecare:

1. Operatori aritmetici

Acești operatori sunt folosiți pentru calcule matematice de bază.

Operator	Descriere	Exemplu	Rezultat
+	Adunare	$5 + 3$	8
-	Scădere	$5 - 3$	2
*	Înmulțire	$5 * 3$	15
/	Împărțire	$5 / 3$	1 (int)
%	Modulo (rest)	$5 \% 3$	2

2. Operatori de alocare

Utilizați pentru a seta valori variabilelor.

Operator	Descriere	Exemplu	Efect
=	Alocare simplă	<code>int x = 5;</code>	x este 5
+=	Adunare și alocare	<code>x += 3;</code>	x este 8
-=	Scădere și alocare	<code>x -= 2;</code>	x este 6
*=	Înmulțire și alocare	<code>x *= 2;</code>	x este 10
/=	Împărțire și alocare	<code>x /= 2;</code>	x este 5
%=	Modulo și alocare	<code>x %= 3;</code>	x este 2

3. Operatori de comparare

Folosiți pentru a compara două valori, rezultatul fiind boolean.

Operator	Descriere	Exemplu	Rezultat
<code>==</code>	Egal cu	<code>5 == 3</code>	false
<code>!=</code>	Diferit de	<code>5 != 3</code>	true
<code>></code>	Mai mare decât	<code>5 > 3</code>	true
<code><</code>	Mai mic decât	<code>5 < 3</code>	false
<code>>=</code>	Mai mare sau egal	<code>5 >= 5</code>	true
<code><=</code>	Mai mic sau egal	<code>5 <= 5</code>	true

4. Operatori logici

Folosiți pentru operații logice.

Operator	Descriere	Exemplu	Rezultat
<code>&&</code>	Și logic	<code>true && false</code>	false
<code> </code>	Sau logic	<code>true false</code>	true
<code>!</code>	Negare logică	<code>!true</code>	false

5. Operatori unari

Operațiuni efectuate pe o singură variabilă.

Operator	Descriere	Exemplu	Rezultat
<code>++</code>	Incrementare	<code>x = 5; x++;</code>	6
<code>--</code>	Decrementare	<code>x = 5; x--;</code>	4

6. Operatori bitwise

Operații pe biți.

Operator	Descriere	Exemplu	Rezultat
&	Şi pe biţi	5 & 3	1
'	'	Sau pe biţi	'5
^	XOR pe biţi	5 ^ 3	6
~	Complement pe biţi	~5	-6
<<	Shift stânga	5 << 1	10
>>	Shift dreapta	5 >> 1	2
>>>	Shift dreapta fără semn	5 >>> 1	2

Expresii booleene

Expresiile booleene în Java sunt expresii care evaluează la o valoare de adevăr, adică **true** sau **false**. Acestea sunt cruciale în controlul fluxului programelor, cum ar fi în instrucțiuni condiționale (if, else, else if) și bucle (while, for). Utilizarea expresiilor booleene permite decizii logice bazate pe comparații, condiții logice și verificări de status.

Componențe ale expresiilor booleene

- **Operatori de comparare** – acești operatori compară două valori și returnează true sau false. Exemple includ **== (egal)**, **!= (diferit)**, **> (mai mare decât)**, **< (mai mic decât)**, **>= (mai mare sau egal cu)** și **<= (mai mic sau egal cu)**.
- **Operatori logici** – folosiți pentru a combina mai multe condiții booleene. Exemple sunt **&& (și logic)**, **|| (sau logic)** și **! (negare)**.
- **Valori booleene directe** – literalii true și false.

Exemple de expresii booleene

Îată câteva exemple care ilustrează utilizarea expresiilor booleene în diferite contexte:

1. Condiții simple

```
public static void main(String[] args) { // Catalin Catalan *
    int a = 5;
    int b = 10;
    boolean result = (a < b); // Evaluează la true, pentru că 5 este mai mic decât 10.
}
```

2. Combinarea condițiilor cu operatori logici

```
public static void main(String[] args) { // Catalin Catalan *
    int score = 75;
    boolean hasPassed = (score >= 50); // true, dacă score este 50 sau mai mare.
    boolean excellent = (score >= 90); // false, pentru că 75 nu este >= 90.
    boolean good = (score >= 70 && score < 90); // true, pentru că 75 este între 70 și 89.
}
```

3. Utilizare în instrucțiuni condiționale

```
public static void main(String[] args) { // Catalin Catalan *
    if (score > 50) {
        System.out.println("Congratulations, you passed!");
    } else {
        System.out.println("Try again next time.");
    }
}
```

4. Utilizare în bucle

```
public static void main(String[] args) { // Catalin Catalan *
    int count = 1;
    while (count <= 5) {
        System.out.println("Count is: " + count);
        count++; // Increment count by 1
    }
    // Aceasta buclă va afisa numerele de la 1 la 5.
}
```

5. Negarea unei condiții

```
public static void main(String[] args) { ▲ Catalin Catalan *
    boolean isRaining = false;
    if (!isRaining) {
        System.out.println("Let's go for a walk!");
    }
}
```

Structuri de control

În Java, structurile de control permit efectuarea deciziilor, executarea repetată a blocurilor de cod și parcurgerea structurilor de date:

1. Instrucțiuni if-else

Instrucțiunile if-else permit executarea condiționată a blocurilor de cod.

```
public static void main(String[] args) { ▲ Catalin Catalan *
    int number = 20;
    if (number > 0) {
        System.out.println("Numărul este pozitiv.");
    } else if (number < 0) {
        System.out.println("Numărul este negativ.");
    } else {
        System.out.println("Numărul este zero.");
    }
}
```

2. Bucle for

Bucurile for sunt folosite pentru a rula un bloc de cod de un număr specific de ori.

```
public static void main(String[] args) { // Catalin Catalan *
    for (int i = 1; i <= 5; i++) {
        System.out.println("Iteratia numărul: " + i);
    }
}
```

3. Bucle enhanced for

Utilizat pentru a itera prin elemente dintr-o colecție sau array fără a folosi indexul.

```
public static void main(String[] args) { // Catalin Catalan *
    int[] numbers = {1, 2, 3, 4, 5};
    for (int number : numbers) {
        System.out.println("Număr: " + number);
    }
}
```

4. Bucle while

Execută un bloc de cod atât timp cât condiția este adevărată.

```
public static void main(String[] args) { // Catalin Catalan *
    int count = 1;
    while (count <= 5) {
        System.out.println("Contor: " + count);
        count++;
    }
}
```

5. Bucle do-while

Similar cu bucla while, dar execută blocul de cod cel puțin o dată înainte de a verifica condiția.

```
public static void main(String[] args) { /* Catalin Catalan */

    int count = 1;

    do {
        System.out.println("Contor: " + count);
        count++;
    } while (count <= 5);
}
```

6. Instrucțiunea switch

Permite executarea unui bloc de cod bazat pe valoarea unei variabile.

```
public static void main(String[] args) { /* Catalin Catalan */

    int day = 4;
    switch (day) {
        case 1:
            System.out.println("Luni");
            break;
        case 2:
            System.out.println("Marți");
            break;
        case 3:
            System.out.println("Miercuri");
            break;
        case 4:
            System.out.println("Joi");
            break;
        case 5:
            System.out.println("Vineri");
            break;
        default:
            System.out.println("Weekend");
            break;
    }
}
```

În Java, instrucțiunile *break* și *continue* sunt folosite pentru a controla fluxul buclelor. Ele ajută la gestionarea execuției buclelor în funcție de condițiile specifice.

Java break

Instrucțiunea *break* este utilizată pentru a ieși dintr-o buclă imediat, indiferent de condiția buclei. Este folosită frecvent în buclele *for*, *while* și *do-while*, precum și în instrucțiunea *switch*.

Exemplu cu break:

Să presupunem că dorim să căutăm un număr într-un array și să oprim bucla imediat ce numărul este găsit.

```
public static void main(String[] args) {    // Catalin Catalan *
    int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int numberToFind = 5;
    boolean found = false;

    for (int number : numbers) {
        if (number == numberToFind) {
            found = true;
            System.out.println("Numărul a fost găsit.");
            break; // Iesim din buclă când numărul este găsit
        }
    }

    if (!found) {
        System.out.println("Numărul nu a fost găsit.");
    }
}
```

În acest exemplu, bucla for se oprește (prin break) imediat ce numărul 5 este găsit în array, fără a mai parcurge și restul tabloului.

Java Continue

Instrucțiunea continue este folosită pentru a sări peste restul codului din corpul curent al buclei și a continua cu următoarea iterare a buclei. Aceasta este utilă atunci când anumite condiții trebuie ignorate într-o iterare a buclei.

Exemplu cu continue:

Să presupunem că dorim să afișăm doar numerele impare dintr-un array, ignorând numerele pare.

```
public static void main(String[] args) { // Catalin Catalan *
    int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (int number : numbers) {
        if (number % 2 == 0) {
            continue; // Sărim peste numerele pare
        }
        System.out.println("Număr impar: " + number); // Afisăm doar numerele impare
    }
}
```

În acest exemplu, instrucțiunea continue face ca bucla for să ignore restul codului pentru numerele pare și să continue cu următoarea iterație pentru numerele impare.

Array-uri

În Java, array-urile (sau tablourile) sunt structuri de date care permit stocarea mai multor valori de același tip într-o singură variabilă. Există două tipuri principale de array-uri în Java: **unidimensionale** și **multidimensionale**.

Array-uri unidimensionale

Un array unidimensional este o listă de variabile de același tip. Pentru a declara un array unidimensional în Java, folosești următorul format:

```
java
tip[] numeArray;
```

Apoi, îl inițializezi specificând dimensiunea:

```
java
numeArray = new tip[dimensiune];
```

Sau poți declara și inițializa array-ul în același timp:

```
java
tip[] numeArray = new tip[dimensiune];
```

De exemplu, un array de întregi cu 5 elemente ar arăta astfel:

```
java
```

```
int[] numere = new int[5];
```

Exemplu:

```
public static void main(String[] args) {    Catalin Catalan *
    int[] numere = {5, 10, 15, 20, 25};
    for (int i = 0; i < numere.length; i++) {
        System.out.println(numere[i]);
    }
}
```

Array-uri multidimensionale

Array-urile multidimensionale sunt array-uri de array-uri, fiecare element fiind la rândul său un array.

Cel mai comun tip este array-ul **bidimensional**, similar cu o matrice.

Pentru a declara un array bidimensional, folosești două seturi de paranteze drepte:

```
java
```

```
tip[][] numeArray;
```

Inițializarea se face specificând dimensiunile:

```
java
```

```
numeArray = new tip[dimensiune1][dimensiune2];
```

Sau declară și inițializezi în același timp:

```
java
```

```
tip[][] numeArray = new tip[dimensiune1][dimensiune2];
```

Exemplu:

```
public static void main(String[] args) { * Catalin Catalan *
    int[][] matrice = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    for (int i = 0; i < matrice.length; i++) {
        for (int j = 0; j < matrice[i].length; j++) {
            System.out.print(matrice[i][j] + " ");
        }
        System.out.println(); // Treci la linia urmatoare dupa fiecare rand
    }
}
```

Metode

Metodele în Java sunt blocuri de cod care execută o anumită operație și sunt definite în cadrul unei clase. Ele permit modularizarea codului, îmbunătățind reutilizarea și organizarea acestuia.

Metodele pot primii parametri de intrare, pot returna valori și pot fi apelate de oriunde în cadrul clasei sau de alte clase, în funcție de nivelul lor de acces.

Structura de bază a unei metode

O metodă în Java este structurată astfel:

```
java

<modificator_acces> <tip_retur> numeMetoda(<lista_parametri>) {
    // Corpul metodei
}
```

- **Modificatorul de acces** controlează vizibilitatea metodei (e.g.: public, private).
- **Tipul de return** specifică tipul de date pe care metoda îl returnează. Dacă metoda nu returnează nimic, se folosește cuvântul cheie *void*.
- **Numele metodei** urmează regulile de denumire ale identificatorilor în Java și ar trebui să descrie acțiunea pe care o efectuează metoda.
- **Lista de parametri** (optională) specifică tipurile și numele variabilelor care primesc valorile transmise metodei la apelare.

Exemple de metode

Metodă simplă fără return și fără parametri

```
java

public void afiseazaMesaj() {
    System.out.println("Hello, world!");
}
```

Metodă care returnează o valoare

```
java

public int adunaDoi(int a, int b) {
    return a + b;
}
```

Metodă cu mai mulți parametri

```
java

public double calculeazaMedie(int x, int y, int z) {
    return (x + y + z) / 3.0;
}
```

Ce este overloading-ul metodelor?

Overloading-ul metodelor sau **suprîncărcarea metodelor**, este un concept în programarea orientată pe obiecte care permite unei clase să aibă mai multe metode cu același nume, dar cu liste diferite de parametri.

Aceasta este o formă de polimorfism și ajută la creșterea lizibilității codului prin utilizarea aceluiași nume de metodă pentru a efectua acțiuni similare, dar diferite în detaliu, bazate pe parametrii furnizați.

Caracteristicile overloading-ului

- Numele metodei rămâne același** – toate metodele supraîncărcate împărtășesc același nume.
- Listele de parametri trebuie să difere** – fie numărul parametrilor este diferit, fie tipurile acestora diferă, fie ambele. Ordinea tipurilor de parametri poate fi, de asemenea, diferită.
- Tipul de return nu contează** – overloading-ul metodelor poate avea loc chiar și dacă tipul de return este același sau diferit; acest lucru nu influențează capacitatea de a supraîncărca metodele.
- Accesibilitatea poate差别** – modificatorii de acces ai metodelor supraîncărcate pot差别.

Considerăm o clasă **Calculator** care oferă diferențe metode pentru a aduna numere, dar care acceptă diferențe tipuri și număr de parametri:

```
public class Calculator { no usages ▾ Catalin Catalan *

    // Adună două numere întregi
    public int add(int a, int b) { new *
        return a + b;
    }

    // Adună trei numere întregi
    public int add(int a, int b, int c) { new *
        return a + b + c;
    }

    // Adună două numere double
    public double add(double a, double b) { new *
        return a + b;
    }

    // Adună două numere folosind variabile variabile de tip int
    public int add(int... numbers) { new *
        int total = 0;
        for (int num : numbers) {
            total += num;
        }
        return total;
    }
}
```

Utilizare:

```

public static void main(String[] args) { // Catalin Catalan*
    Calculator calc = new Calculator();

    // Utilizează diferite versiuni ale metodei "add"
    System.out.println(calc.add(5, 10));           // Afisează 15
    System.out.println(calc.add(5, 10, 15));        // Afisează 30
    System.out.println(calc.add(2.5, 3.5));         // Afisează 6.0
    System.out.println(calc.add(1, 2, 3, 4, 5));    // Afisează 15 folosind varargs
}

```

Overloading-ul de metode este util întrucât permite clasei să implementeze metode care oferă funcționalități similare, dar care sunt potrivite pentru diferite tipuri de date sau număr diferit de argumente.

Metode recursive

Metodele recursive în Java sunt metode care se autoapeleză pentru a rezolva o problemă. Recursivitatea este o tehnică adesea folosită pentru a simplifica soluționarea problemelor care pot fi împărțite în sub-probleme mai mici, similare cu problema inițială. O condiție de oprire (sau cazul de bază) este esențială pentru a preveni apeluri recursive infinite și pentru a asigura că recursivitatea se finalizează.

Exemplu: calculul factorialului

Factorialul unui număr n, notat $n!$, este produsul tuturor numerelor pozitive mai mici sau egale cu n. De exemplu, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 1205$. Factorialul este un exemplu clasic de problemă ce poate fi rezolvată recursiv.

Îată cum poți implementa o metodă recursivă pentru calculul factorialului în Java:

```

public class FactorialCalculator { new *
    // Metoda recursivă pentru calculul factorialului
    public static int factorial(int n) { 2 usages new *
        if (n == 0) { // Cazul de bază: factorialul lui 0 este 1
            return 1;
        } else {
            return n * factorial(n - 1); // Apel recursiv
        }
    }

    public static void main(String[] args) { new *
        int num = 5;
        int result = factorial(num);
        System.out.println("Factorialul lui " + num + " este: " + result);
    }
}

```

Explicație:

- **Cazul de bază** – în recursivitate, cazul de bază este condiția care oprește recursivitatea. Aici, cazul de bază este când n este 0, caz în care metoda returnează 1, deoarece $0!=1$.
- **Apelul recursiv** – dacă n nu este 0, metoda se autoapelează cu argumentul $n-1$. Aceast proces continuă până când se ajunge la cazul de bază, adică n devine 0.