

Răzvan BOCU

Cezar CONSTĂNDOIU

# **MEDII ȘI INSTRUMENTE DE PROGRAMARE. PROGRAMARE JAVA**

**NOTE DE CURS**



Editura  
Universității  
Transilvania  
din Brașov

2026

## EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAȘOV

Adresa: Str. Iuliu Maniu nr. 41A

500091 Brașov

Tel.: 0268 476 050

Fax: 0268 476 051

E-mail: editura@unitbv.ro

**Editură acreditată de CNCSIS, cod 81.**

**ISBN 978-606-19-1879-9 (ebook)**

Copyright © Autorii, 2026

Lucrarea a fost avizată de Consiliul Departamentului de  
Matematică și Informatică, Facultatea de Matematică și Informatică a  
Universității Transilvania din Brașov.

**Copertă:** Imaginea de pe copertă înfățișează un vultur de Java  
(Elang Jawa / *Nisaetus bartelsi*) și este realizată de Jarotornito,  
preluată de pe Wikimedia Commons și licențiată sub termenii CC BY 4.0.



### Despre acest material

Aceste note de curs însoțesc cursul de *Medii și Instrumente de Programare* (componenta Java), an II, Informatică. Structura urmează, cu adaptări, bibliografia:

- ▶ Cay S. Horstmann, *Core Java, Volume I: Fundamentals*, ediția a 14-a, Addison-Wesley, 2025 [1].
- ▶ Cay S. Horstmann, *Core Java, Volume II: Advanced Features*, ediția a 14-a, Addison-Wesley, 2025 [2].
- ▶ Benjamin J. Evans, Jason Clark, David Flanagan, *Java in a Nutshell*, ediția a 8-a, O'Reilly, 2023 [3].

### Colofon

Acest document a fost redactat cu  $\text{\LaTeX}$  folosind clasa `kaobook`.

Imaginea de pe copertă înfățișează un vultur de Java (*Elang Jawa / Nisaetus bartelsi*) și este realizată de [Jarotornito](#), preluată de pe [Wikimedia Commons](#) și licențiată sub termenii [CC BY 4.0](#).

# Cuprins

|                                                            |           |
|------------------------------------------------------------|-----------|
| Cuprins                                                    | 3         |
| Prefață                                                    | 15        |
| <b>Fundamentele limbajului Java</b>                        | <b>17</b> |
| <b>1 Platforma Java. Mediul de dezvoltare</b>              | <b>19</b> |
| 1.1 Java ca platformă de programare                        | 19        |
| 1.1.1 Cele 11 caracteristici ale limbajului                | 19        |
| 1.1.2 Mașina virtuală Java, bytecode și compilarea JIT     | 20        |
| 1.1.3 Evoluția Java: de la applet-uri la platforma modernă | 20        |
| 1.1.4 Concepții greșite comune despre Java                 | 21        |
| 1.2 Instalarea și configurarea JDK-ului                    | 21        |
| 1.2.1 Descărcarea și instalarea JDK-ului                   | 22        |
| 1.2.2 Structura unei instalări JDK                         | 22        |
| 1.3 Utilizarea instrumentelor din linia de comandă         | 22        |
| 1.4 Un mediu de dezvoltare integrat (IDE)                  | 23        |
| 1.5 JShell — interpretorul interactiv al Java              | 24        |
| <b>2 Tipuri primitive. Operatori. Controlul fluxului</b>   | <b>27</b> |
| 2.1 Primul program Java, revizitat                         | 27        |
| 2.1.1 Comentarii                                           | 27        |
| 2.1.2 Convenții de stil                                    | 28        |
| 2.2 Tipuri de date primitive                               | 28        |
| 2.2.1 Literalii numerici                                   | 28        |
| 2.2.2 Unicode și tipul char                                | 29        |
| 2.3 Variabile, constante și tipuri enumerate               | 29        |
| 2.3.1 Declararea variabilelor                              | 29        |
| 2.3.2 Tipuri enumerate                                     | 29        |
| 2.4 Operatori                                              | 30        |
| 2.4.1 Operatori aritmetici și de atribuire                 | 30        |
| 2.4.2 Operatori relaționali și logici                      | 30        |
| 2.4.3 Conversii și cast-uri                                | 30        |
| 2.4.4 Expresii switch                                      | 31        |
| 2.5 Structuri de control                                   | 31        |
| 2.5.1 Instrucțiuni condiționale                            | 31        |
| 2.5.2 Bucle                                                | 31        |
| 2.5.3 switch clasic                                        | 32        |
| 2.5.4 break și continue                                    | 32        |
| 2.6 Array-uri                                              | 33        |
| 2.6.1 Declarare, alocare, acces                            | 33        |
| 2.6.2 Bucla for-each                                       | 33        |
| 2.6.3 Copierea array-urilor                                | 33        |
| 2.6.4 Sortare și alte operații utilitare                   | 34        |
| 2.6.5 Array-uri multidimensionale și neregulate            | 34        |
| <b>3 Șiruri de caractere. Input/Output. Numere mari</b>    | <b>37</b> |
| 3.1 Clasa String                                           | 37        |
| 3.1.1 Concatenare și conversii                             | 37        |
| 3.1.2 Indexare și metode uzuale                            | 37        |
| 3.1.3 Imutabilitate                                        | 38        |
| 3.1.4 Egalitate: == versus equals                          | 38        |

|          |                                                                         |           |
|----------|-------------------------------------------------------------------------|-----------|
| 3.2      | Text blocks și <code>StringBuilder</code> . . . . .                     | 39        |
| 3.2.1    | Text blocks . . . . .                                                   | 39        |
| 3.2.2    | <code>StringBuilder</code> . . . . .                                    | 39        |
| 3.3      | Input/Output de bază . . . . .                                          | 39        |
| 3.3.1    | Citirea datelor de la tastatură . . . . .                               | 39        |
| 3.3.2    | Formatarea ieșirii . . . . .                                            | 40        |
| 3.4      | Numere mari . . . . .                                                   | 40        |
| 3.4.1    | <code>BigInteger</code> . . . . .                                       | 41        |
| 3.4.2    | <code>BigDecimal</code> . . . . .                                       | 41        |
| <b>4</b> | <b>Obiecte și clase (I): încapsulare</b> . . . . .                      | <b>43</b> |
| 4.1      | Introducere în programarea orientată pe obiecte . . . . .               | 43        |
| 4.1.1    | Clase și obiecte . . . . .                                              | 43        |
| 4.1.2    | Identificarea claselor; relații între clase . . . . .                   | 43        |
| 4.2      | Utilizarea claselor predefinite . . . . .                               | 44        |
| 4.2.1    | Variabile obiect versus obiecte . . . . .                               | 44        |
| 4.2.2    | Metode mutator și accesori . . . . .                                    | 45        |
| 4.3      | Definirea claselor proprii . . . . .                                    | 45        |
| 4.3.1    | Anatomia unei clase . . . . .                                           | 45        |
| 4.3.2    | Constructorii; <code>var</code> ; referințe <code>null</code> . . . . . | 46        |
| 4.3.3    | Parametri implicați și expliți . . . . .                                | 47        |
| 4.3.4    | Suprascrierea constructorilor (overloading) . . . . .                   | 48        |
| 4.3.5    | Blocuri de inițializare . . . . .                                       | 48        |
| 4.4      | Încapsulare . . . . .                                                   | 49        |
| 4.4.1    | Modificatori de acces . . . . .                                         | 49        |
| 4.4.2    | Metode private . . . . .                                                | 50        |
| 4.4.3    | Câmpuri <code>final</code> . . . . .                                    | 50        |
| 4.4.4    | Câmpuri statice și constante . . . . .                                  | 51        |
| 4.4.5    | Metode statice; metode <code>factory</code> . . . . .                   | 51        |
| 4.4.6    | Metoda <code>main</code> . . . . .                                      | 52        |
| 4.5      | Records . . . . .                                                       | 53        |
| 4.5.1    | Conceptul de record . . . . .                                           | 53        |
| 4.5.2    | Constructorii canonici, compacti și personalizați . . . . .             | 53        |
| <b>5</b> | <b>Obiecte și clase (II): pachete, JAR, Javadoc</b> . . . . .           | <b>57</b> |
| 5.1      | Pachete . . . . .                                                       | 57        |
| 5.1.1    | Denumirea și organizarea pachetelor . . . . .                           | 57        |
| 5.1.2    | Importuri de clase; <code>import static</code> . . . . .                | 58        |
| 5.1.3    | Class path; compilare cu pachete . . . . .                              | 58        |
| 5.1.4    | Acces la nivel de pachet . . . . .                                      | 59        |
| 5.2      | Fișiere JAR . . . . .                                                   | 59        |
| 5.2.1    | Crearea unui JAR . . . . .                                              | 59        |
| 5.2.2    | Manifestul și JAR-uri executabile . . . . .                             | 60        |
| 5.2.3    | JAR-uri multi-release . . . . .                                         | 60        |
| 5.3      | Comentarii de documentație (Javadoc) . . . . .                          | 61        |
| 5.3.1    | Sintaxa comentariilor Javadoc . . . . .                                 | 61        |
| 5.3.2    | Etichete (tag-uri) comune . . . . .                                     | 61        |
| 5.3.3    | Generarea documentației . . . . .                                       | 62        |
| 5.4      | Recomandări de design pentru clase . . . . .                            | 63        |
| 5.4.1    | Principii generale . . . . .                                            | 63        |
| 5.4.2    | Greșeli frecvente de evitat . . . . .                                   | 63        |
| <b>6</b> | <b>Moștenire</b> . . . . .                                              | <b>65</b> |
| 6.1      | Superclase și subclase . . . . .                                        | 65        |
| 6.1.1    | Definirea unei subclase . . . . .                                       | 65        |

|          |                                                                                         |           |
|----------|-----------------------------------------------------------------------------------------|-----------|
| 6.1.2    | Apelul <code>super(...)</code> și constructorii în subclase                             | 66        |
| 6.1.3    | Suprascrierea metodelor                                                                 | 66        |
| 6.2      | Polimorfism                                                                             | 67        |
| 6.2.1    | Legarea dinamică ( <i>dynamic dispatch</i> )                                            | 67        |
| 6.2.2    | Cast-uri între clase înrudite; <code>instanceof</code> și <code>pattern matching</code> | 67        |
| 6.2.3    | Modificatorul <code>protected</code> ; clase și metode <code>final</code>               | 68        |
| 6.3      | Clasa <code>Object</code> – superclasa cosmică                                          | 69        |
| 6.3.1    | <code>equals</code>                                                                     | 69        |
| 6.3.2    | <code>hashCode</code>                                                                   | 70        |
| 6.3.3    | <code>toString</code>                                                                   | 70        |
| 6.4      | <code>ArrayList</code> generic; autoboxing                                              | 71        |
| 6.4.1    | <code>ArrayList</code> de bază                                                          | 71        |
| 6.4.2    | Autoboxing și clase wrapper                                                             | 71        |
| 6.5      | Metode cu număr variabil de argumente ( <i>varargs</i> )                                | 72        |
| 6.6      | Clase abstracte, enum avansat, clase <code>sealed</code>                                | 73        |
| 6.6.1    | Clase abstracte                                                                         | 73        |
| 6.6.2    | Clase enumerate, revizitate                                                             | 74        |
| 6.6.3    | Clase <code>sealed</code>                                                               | 75        |
| <b>7</b> | <b>Interfețe. Lambda. Clase imbricate</b>                                               | <b>77</b> |
| 7.1      | Interfețe                                                                               | 77        |
| 7.1.1    | Conceptul de interfață                                                                  | 77        |
| 7.1.2    | Metode <code>default</code> , <code>static</code> și <code>private</code> în interfețe  | 78        |
| 7.1.3    | Rezolvarea conflictelor între interfețe                                                 | 79        |
| 7.1.4    | Interfețe ca <i>callback</i> -uri; <code>Comparator</code>                              | 79        |
| 7.1.5    | Clonarea obiectelor                                                                     | 80        |
| 7.2      | Expresii Lambda                                                                         | 80        |
| 7.2.1    | Sintaxă                                                                                 | 80        |
| 7.2.2    | Interfețe funcționale                                                                   | 81        |
| 7.2.3    | Interfețe funcționale standard ( <code>java.util.function</code> )                      | 82        |
| 7.3      | Referințe la metode și constructori; <code>lambda</code> și <code>this</code>           | 82        |
| 7.3.1    | Referințe la metode                                                                     | 82        |
| 7.3.2    | Referințe la constructori                                                               | 83        |
| 7.3.3    | Domeniul de vizibilitate; <code>lambda</code> și <code>this</code>                      | 83        |
| 7.4      | Clase imbricate                                                                         | 84        |
| 7.4.1    | Clase imbricate statice                                                                 | 84        |
| 7.4.2    | Clase interne (non-stactice)                                                            | 85        |
| 7.4.3    | Clase locale și anonime                                                                 | 86        |
| 7.4.4    | Records imbricate                                                                       | 86        |
| <b>8</b> | <b>Pattern Matching. Reflecție</b>                                                      | <b>89</b> |
| 8.1      | Pattern matching cu <code>instanceof</code> ; deconstruirea recordurilor                | 89        |
| 8.1.1    | Recapitulare: <code>instanceof</code> cu <code>pattern matching</code>                  | 89        |
| 8.1.2    | Deconstruirea recordurilor ( <i>record patterns</i> )                                   | 89        |
| 8.1.3    | Pattern-uri imbricate                                                                   | 90        |
| 8.2      | Pattern matching cu <code>switch</code>                                                 | 90        |
| 8.2.1    | <code>Switch</code> pe tipuri                                                           | 90        |
| 8.2.2    | Gărzi ( <i>guards</i> )                                                                 | 91        |
| 8.2.3    | Exhaustivitate și dominanță                                                             | 91        |
| 8.2.4    | Gestionarea <code>null</code> în <code>switch</code>                                    | 92        |
| 8.3      | Reflecție: clasa <code>Class</code> și inspecția la runtime                             | 92        |
| 8.3.1    | Clasa <code>Class</code>                                                                | 92        |
| 8.3.2    | Analiza câmpurilor și metodelor la runtime                                              | 93        |

|           |                                                              |            |
|-----------|--------------------------------------------------------------|------------|
| 8.4       | Invocare dinamică; Service Loaders și Proxy . . . . .        | 94         |
| 8.4.1     | Invocarea dinamică a metodelor . . . . .                     | 94         |
| 8.4.2     | Invocarea dinamică a constructorilor . . . . .               | 94         |
| 8.4.3     | Service Loaders și Proxy (context) . . . . .                 | 95         |
| <b>9</b>  | <b>Excepții. Aserții. Logging</b> . . . . .                  | <b>97</b>  |
| 9.1       | Clasificarea excepțiilor . . . . .                           | 97         |
| 9.1.1     | Ierarhia claselor de excepții . . . . .                      | 97         |
| 9.1.2     | Excepții verificate versus neverificate . . . . .            | 97         |
| 9.1.3     | Declararea și aruncarea excepțiilor . . . . .                | 98         |
| 9.1.4     | Clase de excepții proprii . . . . .                          | 98         |
| 9.2       | Capturarea excepțiilor . . . . .                             | 99         |
| 9.2.1     | try/catch/finally . . . . .                                  | 99         |
| 9.2.2     | Capturare multiplă (multi-catch) . . . . .                   | 100        |
| 9.2.3     | Re-aruncare și <i>chaining</i> de excepții . . . . .         | 100        |
| 9.3       | try-with-resources; stack trace; bune practici . . . . .     | 101        |
| 9.3.1     | try-with-resources . . . . .                                 | 101        |
| 9.3.2     | Analiza stack trace-ului . . . . .                           | 102        |
| 9.3.3     | Bune practici . . . . .                                      | 102        |
| 9.4       | Aserții și Logging . . . . .                                 | 102        |
| 9.4.1     | Aserții . . . . .                                            | 102        |
| 9.4.2     | API-ul de logging . . . . .                                  | 103        |
| 9.4.3     | Configurare: handlers, filtre, formatters . . . . .          | 104        |
| 9.4.4     | O rețetă practică de logging . . . . .                       | 104        |
| 9.4.5     | Tehnici de debugging . . . . .                               | 105        |
|           | <b>Programare generică, colecții și concurență</b> . . . . . | <b>107</b> |
| <b>10</b> | <b>Programare generică</b> . . . . .                         | <b>109</b> |
| 10.1      | Clase și metode generice . . . . .                           | 109        |
| 10.1.1    | Motivație: de ce generice? . . . . .                         | 109        |
| 10.1.2    | O clasă generică proprie . . . . .                           | 109        |
| 10.1.3    | O metodă generică . . . . .                                  | 110        |
| 10.2      | <i>Bounds</i> (limite) pentru parametri de tip . . . . .     | 111        |
| 10.2.1    | <i>Bounds</i> simple . . . . .                               | 111        |
| 10.2.2    | <i>Bounds</i> multiple . . . . .                             | 111        |
| 10.3      | Type erasure . . . . .                                       | 112        |
| 10.3.1    | Cum funcționează erasure . . . . .                           | 112        |
| 10.3.2    | Consecințe practice ale erasure . . . . .                    | 112        |
| 10.3.3    | Reguli de moștenire pentru tipuri generice . . . . .         | 113        |
| 10.4      | Wildcard-uri . . . . .                                       | 113        |
| 10.4.1    | Wildcard nemărginit (?) . . . . .                            | 113        |
| 10.4.2    | Wildcard mărginit superior (? <i>extends</i> ) . . . . .     | 114        |
| 10.4.3    | Wildcard mărginit inferior (? <i>super</i> ) . . . . .       | 114        |
| 10.4.4    | Capturarea wildcard . . . . .                                | 115        |
| 10.5      | Restricții și limitări; generice și reflecție . . . . .      | 115        |
| 10.5.1    | Alte restricții . . . . .                                    | 115        |
| 10.5.2    | Generice și reflecție . . . . .                              | 116        |
| <b>11</b> | <b>Colecții (I): liste, mulțimi, cozi</b> . . . . .          | <b>119</b> |
| 11.1      | Framework-ul de colecții . . . . .                           | 119        |
| 11.1.1    | Ierarhia interfețelor . . . . .                              | 119        |
| 11.1.2    | Iteratori . . . . .                                          | 119        |

|           |                                                                              |            |
|-----------|------------------------------------------------------------------------------|------------|
| 11.2      | Liste                                                                        | 120        |
| 11.2.1    | ArrayList, revizitat                                                         | 120        |
| 11.2.2    | LinkedList                                                                   | 121        |
| 11.3      | Mulțimi                                                                      | 121        |
| 11.3.1    | HashSet                                                                      | 121        |
| 11.3.2    | TreeSet                                                                      | 122        |
| 11.4      | Cozi                                                                         | 123        |
| 11.4.1    | Queue și Deque                                                               | 123        |
| 11.4.2    | PriorityQueue                                                                | 123        |
| 11.5      | Algoritmi pe colecții                                                        | 124        |
| 11.5.1    | Sortare și <i>shuffling</i>                                                  | 124        |
| 11.5.2    | Căutare binară                                                               | 125        |
| 11.5.3    | Operații bulk                                                                | 125        |
| 11.5.4    | Conversie colecții și array-uri                                              | 125        |
| <b>12</b> | <b>Colecții (II): Maps. Vizualizări. Colecții imutabile</b>                  | <b>129</b> |
| 12.1      | Map: operații de bază și actualizare                                         | 129        |
| 12.1.1    | Operații de bază                                                             | 129        |
| 12.1.2    | Metode de actualizare convenabile                                            | 129        |
| 12.1.3    | Vizualizări ale unui Map                                                     | 130        |
| 12.2      | Variante specializate de Map/Set                                             | 131        |
| 12.2.1    | LinkedHashMap și LinkedHashSet                                               | 131        |
| 12.2.2    | TreeMap                                                                      | 131        |
| 12.2.3    | EnumMap și EnumSet                                                           | 132        |
| 12.2.4    | WeakHashMap și IdentityHashMap (context)                                     | 132        |
| 12.3      | Copii și vizualizări imutabile                                               | 133        |
| 12.3.1    | Colecții mici, imutabile prin construcție                                    | 133        |
| 12.3.2    | Copii și vizualizări imutabile ale colecțiilor existente                     | 133        |
| 12.3.3    | Subranges                                                                    | 134        |
| 12.3.4    | Vizualizări inversate și sincronizate                                        | 134        |
| 12.4      | Colecții legacy (context istoric)                                            | 135        |
| <b>13</b> | <b>Concurență (I): thread-uri. Executors</b>                                 | <b>137</b> |
| 13.1      | Pornirea thread-urilor; stările unui thread                                  | 137        |
| 13.1.1    | Crearea și pornirea unui thread                                              | 137        |
| 13.1.2    | Stările unui thread                                                          | 137        |
| 13.1.3    | Întreruperea thread-urilor; daemon threads; priorități                       | 138        |
| 13.2      | Virtual threads                                                              | 139        |
| 13.2.1    | Motivație: platform threads versus virtual threads                           | 139        |
| 13.3      | Coordonarea task-urilor: Callable/Future; ExecutorService                    | 139        |
| 13.3.1    | Callable și Future                                                           | 139        |
| 13.3.2    | ExecutorService                                                              | 140        |
| 13.3.3    | Invocarea unui grup de task-uri                                              | 141        |
| 13.4      | Thread-local variables; scoped values; fork-join                             | 141        |
| 13.4.1    | ThreadLocal                                                                  | 141        |
| 13.4.2    | Scoped values                                                                | 142        |
| 13.4.3    | Framework-ul fork-join (context)                                             | 142        |
| <b>14</b> | <b>Concurență (II): sincronizare. Colecții concurente. CompletableFuture</b> | <b>145</b> |
| 14.1      | Condiții de cursă; synchronized; deadlock-uri                                | 145        |
| 14.1.1    | Condiții de cursă                                                            | 145        |
| 14.1.2    | synchronized; conceptul de monitor                                           | 145        |
| 14.1.3    | Deadlock-uri                                                                 | 146        |
| 14.2      | Lock/Condition; volatile; tipuri atomice                                     | 147        |
| 14.2.1    | Lock explicit                                                                | 147        |

|                                                       |                                                                 |            |
|-------------------------------------------------------|-----------------------------------------------------------------|------------|
| 14.2.2                                                | Câmpuri volatile                                                | 147        |
| 14.2.3                                                | Tipuri atomice                                                  | 148        |
| 14.2.4                                                | Inițializare on-demand; publicare sigură                        | 148        |
| 14.3                                                  | Colecții thread-safe                                            | 149        |
| 14.3.1                                                | BlockingQueue                                                   | 149        |
| 14.3.2                                                | ConcurrentHashMap                                               | 149        |
| 14.3.3                                                | Colecții copy-on-write                                          | 150        |
| 14.4                                                  | CompletableFuture; procese                                      | 150        |
| 14.4.1                                                | CompletableFuture de bază                                       | 150        |
| 14.4.2                                                | Compunerea de future-uri                                        | 151        |
| 14.4.3                                                | Procese externe (ProcessBuilder)                                | 152        |
| <b>Streams, intrare/ieșire, rețea și baze de date</b> |                                                                 | <b>155</b> |
| <b>15</b>                                             | <b>Streams (I): transformări și reduceri</b>                    | <b>157</b> |
| 15.1                                                  | De la iterare la stream-uri                                     | 157        |
| 15.1.1                                                | Motivație: stil imperativ versus declarativ                     | 157        |
| 15.1.2                                                | Crearea stream-urilor                                           | 157        |
| 15.2                                                  | filter, map, flatMap                                            | 158        |
| 15.2.1                                                | filter                                                          | 158        |
| 15.2.2                                                | map                                                             | 158        |
| 15.2.3                                                | flatMap                                                         | 158        |
| 15.2.4                                                | Alte transformări                                               | 159        |
| 15.3                                                  | Reduceri simple                                                 | 159        |
| 15.3.1                                                | forEach, count, min/max                                         | 159        |
| 15.3.2                                                | reduce                                                          | 160        |
| 15.4                                                  | Tipul Optional                                                  | 160        |
| 15.4.1                                                | Obținerea și consumarea valorilor                               | 160        |
| 15.4.2                                                | Pipelining cu Optional                                          | 161        |
| 15.4.3                                                | Anti-pattern-uri                                                | 161        |
| 15.4.4                                                | flatMap cu Optional                                             | 162        |
| <b>16</b>                                             | <b>Streams (II): Collectors. Gatherers. Stream-uri paralele</b> | <b>165</b> |
| 16.1                                                  | Collectors: colectare în colecții și Map                        | 165        |
| 16.1.1                                                | collect(), toList/toSet                                         | 165        |
| 16.1.2                                                | Colectare într-un Map                                           | 165        |
| 16.2                                                  | Grupare, partiționare, downstream collectors                    | 166        |
| 16.2.1                                                | groupingBy                                                      | 166        |
| 16.2.2                                                | partitioningBy                                                  | 166        |
| 16.2.3                                                | Downstream collectors                                           | 167        |
| 16.2.4                                                | Collectors proprii (context)                                    | 168        |
| 16.3                                                  | Gatherers                                                       | 168        |
| 16.3.1                                                | Motivație și gatherers predefiniți                              | 168        |
| 16.3.2                                                | Implementare proprie (context)                                  | 168        |
| 16.4                                                  | Stream-uri pe tipuri primitive; stream-uri paralele             | 169        |
| 16.4.1                                                | IntStream, DoubleStream, LongStream                             | 169        |
| 16.4.2                                                | Stream-uri paralele                                             | 169        |
| <b>17</b>                                             | <b>Input/Output (I): stream-uri și fișiere</b>                  | <b>173</b> |
| 17.1                                                  | Stream-uri de octeți                                            | 173        |
| 17.1.1                                                | InputStream și OutputStream                                     | 173        |
| 17.1.2                                                | Combinarea filtrelor                                            | 174        |

|           |                                                                                     |            |
|-----------|-------------------------------------------------------------------------------------|------------|
| 17.2      | Input/output de text; codificări de caractere . . . . .                             | 174        |
| 17.2.1    | Reader și Writer . . . . .                                                          | 174        |
| 17.2.2    | Codificări de caractere . . . . .                                                   | 175        |
| 17.3      | Date binare; fișiere cu acces aleator; arhive ZIP . . . . .                         | 175        |
| 17.3.1    | DataInput/DataOutput . . . . .                                                      | 175        |
| 17.3.2    | Fișiere cu acces aleator . . . . .                                                  | 176        |
| 17.3.3    | Arhive ZIP . . . . .                                                                | 177        |
| 17.4      | Lucrul cu fișiere (NIO.2) . . . . .                                                 | 177        |
| 17.4.1    | Path și operații de bază . . . . .                                                  | 177        |
| 17.4.2    | Citire/scriere directă de text . . . . .                                            | 177        |
| 17.4.3    | Creare, copiere, mutare, ștergere . . . . .                                         | 178        |
| 17.4.4    | Informații despre fișiere; vizitarea directoarelor . . . . .                        | 178        |
| <b>18</b> | <b>Input/Output (II): serializare. Expresii regulate. Fișiere mapate în memorie</b> | <b>181</b> |
| 18.1      | Fișiere mapate în memorie; blocarea fișierelor . . . . .                            | 181        |
| 18.1.1    | Fișiere mapate în memorie . . . . .                                                 | 181        |
| 18.1.2    | Blocarea fișierelor . . . . .                                                       | 181        |
| 18.2      | Serializare . . . . .                                                               | 182        |
| 18.2.1    | Salvare și încărcare de obiecte serializabile . . . . .                             | 182        |
| 18.2.2    | Câmpuri transient . . . . .                                                         | 183        |
| 18.2.3    | readObject/writeObject personalizate . . . . .                                      | 183        |
| 18.2.4    | Versionare și securitate . . . . .                                                  | 184        |
| 18.3      | Expresii regulate: sintaxă de bază . . . . .                                        | 184        |
| 18.3.1    | Sintaxă și testarea potrivirilor . . . . .                                          | 184        |
| 18.3.2    | Grupuri de captură . . . . .                                                        | 185        |
| 18.4      | Expresii regulate: aplicații . . . . .                                              | 186        |
| 18.4.1    | split . . . . .                                                                     | 186        |
| 18.4.2    | replace . . . . .                                                                   | 186        |
| 18.4.3    | Flags . . . . .                                                                     | 186        |
| <b>19</b> | <b>XML</b>                                                                          | <b>189</b> |
| 19.1      | Structura unui document XML; parsare DOM . . . . .                                  | 189        |
| 19.1.1    | Structura unui document XML . . . . .                                               | 189        |
| 19.1.2    | Parsare DOM . . . . .                                                               | 189        |
| 19.2      | Validare (DTD, XML Schema) . . . . .                                                | 190        |
| 19.2.1    | DTD . . . . .                                                                       | 190        |
| 19.2.2    | XML Schema . . . . .                                                                | 191        |
| 19.3      | XPath; namespace-uri; parsare streaming . . . . .                                   | 192        |
| 19.3.1    | XPath . . . . .                                                                     | 192        |
| 19.3.2    | Namespace-uri . . . . .                                                             | 192        |
| 19.3.3    | Parsare streaming: SAX și StAX . . . . .                                            | 193        |
| 19.4      | Generarea documentelor XML; transformări XSL . . . . .                              | 194        |
| 19.4.1    | Generarea documentelor XML . . . . .                                                | 194        |
| 19.4.2    | Transformări XSL (context) . . . . .                                                | 194        |
| <b>20</b> | <b>Conectivitate în rețea</b>                                                       | <b>197</b> |
| 20.1      | Conectarea la un server . . . . .                                                   | 197        |
| 20.1.1    | Socket: conectare de bază . . . . .                                                 | 197        |
| 20.1.2    | Adrese de Internet; timeouts . . . . .                                              | 197        |
| 20.2      | Implementarea serverelor . . . . .                                                  | 198        |
| 20.2.1    | ServerSocket: server de bază . . . . .                                              | 198        |
| 20.2.2    | Deservirea mai multor clienți . . . . .                                             | 198        |
| 20.2.3    | Comunicare securizată (SSL/TLS) . . . . .                                           | 199        |
| 20.3      | Obținerea de date web; HTTP Client modern . . . . .                                 | 200        |
| 20.3.1    | URL, URI și URLConnection (context istoric) . . . . .                               | 200        |

|           |                                                                 |            |
|-----------|-----------------------------------------------------------------|------------|
| 20.3.2    | HttpClient . . . . .                                            | 200        |
| 20.3.3    | Procesare asincronă . . . . .                                   | 201        |
| 20.4      | Simple HTTP Server; trimiterea de e-mail . . . . .              | 201        |
| 20.4.1    | Simple HTTP Server (context) . . . . .                          | 201        |
| 20.4.2    | Trimiterea de e-mail (context) . . . . .                        | 202        |
| <b>21</b> | <b>Programarea cu baze de date</b>                              | <b>205</b> |
| 21.1      | Designul JDBC; SQL — recapitulare . . . . .                     | 205        |
| 21.1.1    | Designul JDBC . . . . .                                         | 205        |
| 21.1.2    | SQL — recapitulare . . . . .                                    | 205        |
| 21.2      | Configurare și conectare . . . . .                              | 206        |
| 21.2.1    | URL-uri de bază de date; JAR-uri de driver . . . . .            | 206        |
| 21.2.2    | Conectare . . . . .                                             | 206        |
| 21.3      | Statement-uri; PreparedStatement; excepții SQL . . . . .        | 207        |
| 21.3.1    | Statement: execuție de bază . . . . .                           | 207        |
| 21.3.2    | PreparedStatement . . . . .                                     | 207        |
| 21.3.3    | Excepții SQL . . . . .                                          | 208        |
| 21.4      | Chei autogenerate; metadata; result set-uri avansate . . . . .  | 209        |
| 21.4.1    | Chei autogenerate . . . . .                                     | 209        |
| 21.4.2    | Metadata . . . . .                                              | 209        |
| 21.4.3    | Result set-uri scrollabile și actualizabile; row sets . . . . . | 210        |
| 21.5      | Tranzacții; batch updates; gestionarea conexiunilor . . . . .   | 210        |
| 21.5.1    | Tranzacții . . . . .                                            | 210        |
| 21.5.2    | Savepoints . . . . .                                            | 211        |
| 21.5.3    | Batch updates . . . . .                                         | 212        |
| 21.5.4    | Gestionarea conexiunilor . . . . .                              | 212        |
|           | <b>Interfețe grafice și tehnologii complementare</b>            | <b>215</b> |
| <b>22</b> | <b>Interfețe grafice (I): AWT. Swing de bază</b>                | <b>217</b> |
| 22.1      | Istoricul toolkit-urilor UI; JFrame . . . . .                   | 217        |
| 22.1.1    | Istoricul toolkit-urilor UI în Java . . . . .                   | 217        |
| 22.1.2    | JFrame; proprietăți de fereastră . . . . .                      | 217        |
| 22.2      | Desenare 2D; culoare, fonturi, imagini . . . . .                | 218        |
| 22.2.1    | Desenare de bază . . . . .                                      | 218        |
| 22.2.2    | Culoare și fonturi . . . . .                                    | 219        |
| 22.2.3    | Imagini . . . . .                                               | 219        |
| 22.3      | Gestionarea evenimentelor . . . . .                             | 220        |
| 22.3.1    | Listeners; clase adapter . . . . .                              | 220        |
| 22.3.2    | Action . . . . .                                                | 220        |
| 22.3.3    | Evenimente de mouse . . . . .                                   | 221        |
| 22.4      | Swing, Model-View-Controller, layout managers . . . . .         | 221        |
| 22.4.1    | Swing și Model-View-Controller . . . . .                        | 221        |
| 22.4.2    | Layout managers . . . . .                                       | 222        |
| <b>23</b> | <b>Interfețe grafice (II): componente Swing. Dialoguri</b>      | <b>225</b> |
| 23.1      | Input de text . . . . .                                         | 225        |
| 23.1.1    | TextField, JPasswordField . . . . .                             | 225        |
| 23.1.2    | TextArea; scroll panes . . . . .                                | 225        |
| 23.2      | Componente de selecție . . . . .                                | 226        |
| 23.2.1    | Checkbox-uri și radio buttons . . . . .                         | 226        |
| 23.2.2    | JComboBox . . . . .                                             | 226        |
| 23.2.3    | Slidere . . . . .                                               | 227        |

|           |                                                                         |            |
|-----------|-------------------------------------------------------------------------|------------|
| 23.3      | Meniuri, toolbar-uri, tooltips . . . . .                                | 227        |
| 23.3.1    | Meniuri . . . . .                                                       | 227        |
| 23.3.2    | Toolbar-uri . . . . .                                                   | 228        |
| 23.3.3    | Tooltips . . . . .                                                      | 228        |
| 23.4      | Dialoguri . . . . .                                                     | 228        |
| 23.4.1    | JOptionPane . . . . .                                                   | 228        |
| 23.4.2    | Dialoguri personalizate (JDialog) . . . . .                             | 229        |
| 23.4.3    | Dialoguri de fișiere (JFileChooser) . . . . .                           | 230        |
| 23.5      | JTable și JTree (introducere) . . . . .                                 | 230        |
| 23.5.1    | JTable . . . . .                                                        | 230        |
| 23.5.2    | JTree (context) . . . . .                                               | 231        |
| <b>24</b> | <b>Adnotări. Java Platform Module System</b>                            | <b>233</b> |
| 24.1      | Adnotări: concept și adnotări predefinite . . . . .                     | 233        |
| 24.1.1    | Conceptul de adnotare . . . . .                                         | 233        |
| 24.1.2    | Adnotări predefinite . . . . .                                          | 233        |
| 24.2      | Definirea adnotărilor proprii; procesare; adnotări la runtime . . . . . | 234        |
| 24.2.1    | Definirea unei adnotări proprii . . . . .                               | 234        |
| 24.2.2    | Procesoare de adnotări (context) . . . . .                              | 234        |
| 24.2.3    | Adnotări la runtime, prin reflecție . . . . .                           | 235        |
| 24.3      | Java Platform Module System: module, module-info.java . . . . .         | 236        |
| 24.3.1    | Conceptul de modul . . . . .                                            | 236        |
| 24.3.2    | module-info.java; requires/exports . . . . .                            | 236        |
| 24.4      | Module automate vs. explicite; migrarea codului legacy . . . . .        | 237        |
| 24.4.1    | Module automate . . . . .                                               | 237        |
| 24.4.2    | Migrarea codului legacy . . . . .                                       | 237        |
| <b>25</b> | <b>API-ul Date and Time</b>                                             | <b>239</b> |
| 25.1      | Linia timpului; LocalDate; ajustori de date . . . . .                   | 239        |
| 25.1.1    | Instant: linia timpului absolută . . . . .                              | 239        |
| 25.1.2    | LocalDate, recapitulat; ajustori de date . . . . .                      | 239        |
| 25.2      | LocalTime; ZonedDateTime . . . . .                                      | 240        |
| 25.2.1    | LocalTime . . . . .                                                     | 240        |
| 25.2.2    | ZonedDateTime; fusuri orare . . . . .                                   | 240        |
| 25.3      | Formatare și parsare . . . . .                                          | 241        |
| 25.3.1    | DateTimeFormatter: formatare . . . . .                                  | 241        |
| 25.3.2    | Parsare . . . . .                                                       | 241        |
| 25.4      | Interoperare cu API-ul legacy (Date, Calendar) . . . . .                | 242        |
| <b>26</b> | <b>Internaționalizare</b>                                               | <b>245</b> |
| 26.1      | Locale . . . . .                                                        | 245        |
| 26.1.1    | Specificarea unui Locale . . . . .                                      | 245        |
| 26.1.2    | Locale implicit; denumiri afișate . . . . .                             | 245        |
| 26.2      | Formate numerice și monetare . . . . .                                  | 246        |
| 26.2.1    | NumberFormat, DecimalFormat . . . . .                                   | 246        |
| 26.2.2    | Formatarea monedelor . . . . .                                          | 246        |
| 26.3      | Formatare dată/timp internaționalizată; collation . . . . .             | 247        |
| 26.3.1    | Formatare dată/timp internaționalizată . . . . .                        | 247        |
| 26.3.2    | Collation . . . . .                                                     | 247        |
| 26.4      | Formatare de mesaje; frontiere de text . . . . .                        | 248        |
| 26.4.1    | MessageFormat; <i>choice formats</i> . . . . .                          | 248        |
| 26.4.2    | Frontiere de text (context) . . . . .                                   | 249        |
| 26.5      | Resource bundles: localizarea aplicațiilor . . . . .                    | 249        |
|           | <b>Bibliografie</b>                                                     | <b>253</b> |



# Listă de figuri

|      |                                                                                                                                                                                    |     |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 1.1  | Fluxul de compilare și execuție a unui program Java: fișierul sursă este compilat de javac în bytecode, care este apoi interpretat și/sau compilat JIT de mașina virtuală. . . . . | 21  |
| 4.1  | Notăția UML simplificată pentru o clasă: numele clasei, câmpurile (cu vizibilitate, aici toate private, marcate „-”) și metodele (aici publice, marcate „+”). . . . .              | 44  |
| 9.1  | Ierarhia (simplificată) a claselor de excepții din Java. . . . .                                                                                                                   | 97  |
| 11.1 | Ierarhia (simplificată) a interfețelor din framework-ul de colecții. . . . .                                                                                                       | 119 |
| 13.1 | Ciclul de viață (simplificat) al unui thread Java. . . . .                                                                                                                         | 138 |

# Listă de tabele

|      |                                                                                                                                                |     |
|------|------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 1.1  | Versiunile Java cu suport pe termen lung (LTS). . . . .                                                                                        | 20  |
| 2.1  | Tipurile primitive din Java și dimensiunile lor, fixate prin specificație. . . . .                                                             | 28  |
| 3.1  | Specificatorii de format cei mai des folosiți cu printf/String.format. . . . .                                                                 | 40  |
| 4.1  | Nivelurile de vizibilitate din Java, de la cel mai restrictiv la cel mai permisiv. . . . .                                                     | 49  |
| 5.1  | Etichetele Javadoc cel mai frecvent folosite. . . . .                                                                                          | 62  |
| 6.1  | Câteva clase wrapper corespunzătoare tipurilor primitive (lista completă are opt perechi, câte una pentru fiecare tip din Tabela 2.1). . . . . | 72  |
| 7.1  | Interfețele funcționale generice cele mai frecvent folosite din java.util.function.                                                            | 82  |
| 7.2  | Cele trei forme de referințe la metode. . . . .                                                                                                | 83  |
| 9.1  | Nivelurile de severitate ale API-ului de logging, de la cel mai important la cel mai detaliat. . . . .                                         | 104 |
| 11.1 | Complexitatea operațiilor uzuale ale ArrayList. . . . .                                                                                        | 121 |
| 13.1 | Cele mai frecvente metode factory pentru crearea unui ExecutorService. . . . .                                                                 | 140 |
| 14.1 | Metodele de compunere cele mai frecvente ale CompletableFuture. . . . .                                                                        | 151 |
| 16.1 | Downstream collectors frecvent combinate cu groupingBy. . . . .                                                                                | 167 |
| 18.1 | Simbolurile de bază ale sintaxei expresiilor regulate în Java. . . . .                                                                         | 185 |
| 24.1 | Adnotările predefinite, din java.lang, deja folosite pe parcursul cursului. . . . .                                                            | 233 |



# Prefață

Materialul pe care îl aveți în față reprezintă suportul teoretic și aplicativ pentru componenta de Java a cursului de *Medii și Instrumente de Programare*, adresat studenților din anul II al programului de studiu Informatică.

De ce Java? La peste trei decenii de la lansarea sa, Java rămâne unul dintre pilonii industriei software globale, susținând de la aplicații mobile până la arhitecturi *enterprise* complexe. Totuși, limbajul pe care îl veți învăța din aceste pagini nu este acel Java greoi și pur imperativ din anii 2000. Am conceput acest curs pentru a reflecta stadiul modern și agil al platformei. Veți descoperi un limbaj expresiv, care îmbrățișează programarea funcțională și declarativă (prin *Streams* și expresii lambda), care simplifică modelarea datelor (prin *records* și *pattern matching*) și care revoluționează execuția concurentă (prin noile *virtual threads*).

Pentru a facilita o asimilare progresivă a conceptelor, materialul este organizat în 26 de capitole, structurate în patru părți tematice majore:

- ▶ **Fundamentele limbajului:** Trecerea de la gândirea procedurală la cea orientată pe obiecte, stăpânirea principiilor de încapsulare, moștenire și polimorfism, precum și gestionarea corectă a erorilor.
- ▶ **Programare generică, colecții și concurență:** Explorarea structurilor de date fundamentale și înțelegerea provocărilor aduse de execuția simultană a codului, folosind cele mai noi instrumente de sincronizare.
- ▶ **Streams, intrare/ieșire, rețea și baze de date:** Prelucrarea datelor în stil declarativ, interacțiunea cu sistemul de fișiere, comunicarea prin rețea și accesul la baze de date relaționale (JDBC).
- ▶ **Interfețe grafice și tehnologii complementare:** Proiectarea aplicațiilor cu interfață grafică (Swing) și aprofundarea unor concepte arhitecturale precum modulele (JPMS) și API-ul modern de *Date and Time*.

Abordarea noastră este una pragmatică, dar riguroasă. Am inclus nu doar detalii de sintaxă, ci și *bune practici*, capcane frecvente (anti-pattern-uri) de evitat și explicații legate de modul în care mecanismele funcționează „în culise” — de la *Garbage Collector* până la *Type Erasure*.

Totuși, programarea nu se învață doar citind. Vă încurajăm să scrieți cod, să experimentați în IDE sau în JShell, să testați exercițiile de la finalul fiecărui capitol și să faceți propriile greșeli. Ele sunt, deseori, cel mai bun profesor.

Vă dorim un semestru plin de curiozitate și linii de cod care compilează din prima!

*Autorii*

Brașov, 2026



# Fundamentele limbajului Java



# Platforma Java. Mediul de dezvoltare

# 1

## Introducere

Java nu este doar un limbaj de programare, ci o *platformă*: un limbaj, o mașină virtuală și o bibliotecă standard vastă, gândite să funcționeze împreună. În acest capitol vom parcurge istoricul și filozofia din spatele acestei platforme, modul în care codul Java ajunge să fie executat, și instrumentele cu care vom lucra pe tot parcursul cursului: JDK-ul, linia de comandă, un mediu de dezvoltare integrat (IDE) și JShell, interpretorul interactiv al Java.

## 1.1 Java ca platformă de programare

Prima versiune a Java a fost lansată în 1995 de Sun Microsystems și a generat un entuziasm neobișnuit pentru o tehnologie de programare, ajungând subiect de discuție inclusiv în presa generalistă [1]. Promisiunea din spatele limbajului era aceea de a deveni *liantul universal* care leagă utilizatorii de informație, indiferent de unde provine aceasta — servere web, baze de date, alte surse. La peste trei decenii distanță, o mare parte din această promisiune s-a materializat: Java rămâne unul dintre cele mai folosite limbaje de programare din lume, cu aplicații care merg de la sisteme enterprise la aplicații Android.<sup>1</sup>

[1]: Horstmann (2026), *Core Java, Volume I: Fundamentals*

1: Deși Java rulează și pe Android, mașina virtuală folosită acolo (ART) diferă de JVM-ul „clasic” pe care îl vom studia în acest curs.

### Definiție — Platformă vs. limbaj

Un *limbaj de programare* definește sintaxa și semantica prin care exprimăm algoritmi. O *platformă* de programare adaugă la limbaj un mediu de execuție (mașina virtuală), un set de instrumente (compilator, depanator, instrumente de build) și o bibliotecă standard. Java este, prin proiectare, o platformă completă.

### 1.1.1 Cele 11 caracteristici ale limbajului

Autorii Java au publicat, odată cu prima versiune a limbajului, un *white paper* care rezuma obiectivele de proiectare prin 11 caracteristici. Deși unele dintre ele sună astăzi ca marketing, majoritatea au rezistat testului timpului și explică de ce Java a fost adoptat la scară largă:

- **Simple** — sintaxă apropiată de C++, dar fără pointeri expliți, moștenire multiplă de clase, sau supraîncărcare de operatori.
- **Orientat pe obiecte** — totul (cu excepția tipurilor primitive) este obiect.
- **Distribuit** — suport nativ pentru rețea, integrat în bibliotecă încă de la început.
- **Robust** — verificări stricte la compilare și la execuție; fără aritmetică de pointeri care ar putea corupe memoria.
- **Sigur** — gândit inițial pentru execuția de cod nesigur (applet-uri) într-un mediu izolat (*sandbox*).
- **Neutru față de arhitectură** — codul compilat (bytecode) rulează neschimbat pe orice platformă cu o mașină virtuală Java.

- **Portabil** — dimensiunile tipurilor primitive sunt fixate prin specificație, spre deosebire de C/C++.
- **Interpretat** — bytecode-ul este interpretat de mașina virtuală, cu opțiunea compilării *just-in-time* pentru performanță.
- **Performant** — compilarea JIT aduce codul Java aproape de performanța codului nativ, în ciuda naturii interpretate.
- **Multithreaded** — suport nativ pentru execuție concurentă, parte din limbaj, nu doar din bibliotecă.
- **Dinamic** — încărcarea claselor se poate face la momentul execuției, permițând extensibilitate.

#### Notă

Nu este necesar să memorați această listă cuvânt cu cuvânt. Este utilă mai ales pentru a înțelege *de ce* anumite decizii de proiectare din limbaj (lipsa pointerilor, gestionarea automată a memoriei, modelul de mașină virtuală) sunt exact acelea și nu altele.

### 1.1.2 Mașina virtuală Java, bytecode și compilarea JIT

Codul sursă Java (fișiere `.java`) nu este compilat direct în cod mașină, ci într-un format intermediar numit bytecode, stocat în fișiere `.class`. Acest bytecode este executat de JVM (JVM), care poate fi implementată diferit pe fiecare sistem de operare, dar respectă aceeași specificație — de aici vine portabilitatea „write once, run anywhere”.

La execuție, JVM nu se limitează la interpretarea instrucțiune cu instrucțiune a bytecode-ului. Un *compiler just-in-time* (JIT) identifică porțiunile de cod executate frecvent (*hot spots*) și le compilează în cod mașină nativ, în timp ce programul rulează.<sup>2</sup> Această combinație — interpretare inițială plus compilare selectivă — este motivul pentru care Java poate atinge performanțe apropiate de codul nativ, păstrând totodată portabilitatea bytecode-ului.

2: Numele implementării de referință a JVM-ului, *HotSpot*, vine chiar de la această tehnică de optimizare.

### 1.1.3 Evoluția Java: de la applet-uri la platforma modernă

Java a debutat ca tehnologie pentru *applet-uri* — mici programe rulate în browser. Această utilizare a dispărut aproape complet, browserele moderne nemaisusținând applet-uri Java din motive de securitate. Platforma a supraviețuit însă și a prosperat în alte zone: aplicații server-side, sisteme enterprise, procesare de date la scară mare, aplicații Android și, mai recent, chiar aplicații desktop moderne.

Începând cu Java 9 (2017), Oracle a trecut la un **ciclu de lansare de șase luni**, cu o versiune LTS (*Long-Term Support*) desemnată o dată la aproximativ doi-trei ani. Această schimbare este importantă pentru un curs de Java în 2026:

**Tabela 1.1:** Versiunile Java cu suport pe termen lung (LTS).

| Versiune | Statut                                             |
|----------|----------------------------------------------------|
| Java 8   | LTS (istoric, foarte răspândită încă în producție) |
| Java 11  | LTS                                                |
| Java 17  | LTS                                                |
| Java 21  | LTS                                                |
| Java 25  | LTS (versiunea de referință a acestui curs)        |

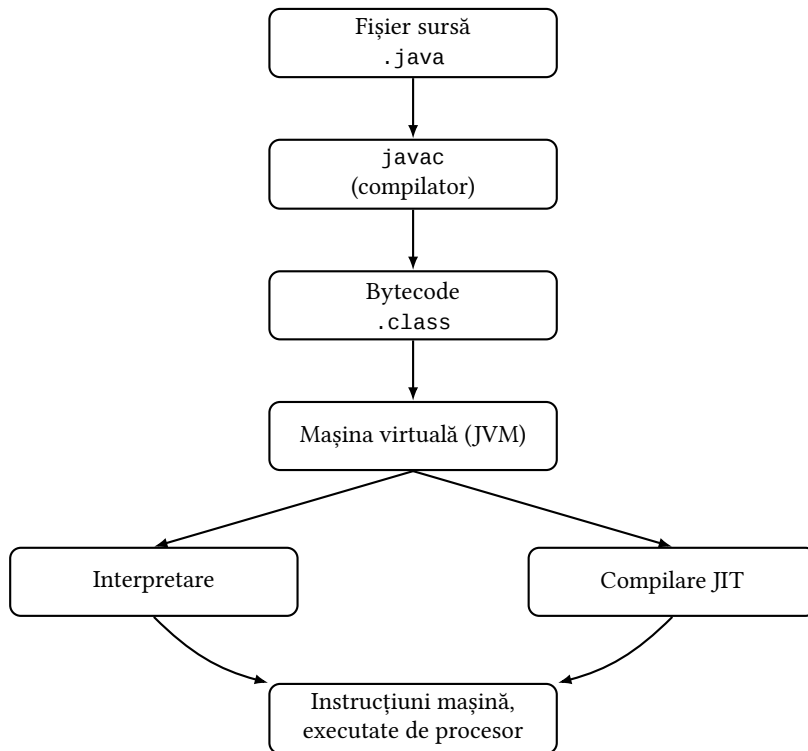


**Figura 1.1:** Fluxul de compilare și execuție a unui program Java: fișierul sursă este compilat de javac în bytecode, care este apoi interpretat și/sau compilat JIT de mașina virtuală.

Materialul acestui curs este aliniat la **Java 25**, versiunea folosită de ediția a 14-a a manualului de referință. Vom marca explicit, pe parcurs, funcționalitățile relativ recente (de exemplu *records*, *pattern matching*, *virtual threads*), pentru ca la finalul cursului să înțelegeți nu doar Java ca limbaj static, ci și direcția în care evoluează.

### 1.1.4 Concepții greșite comune despre Java

Înainte de a instala orice, merită demontate câteva confuzii răspândite, mai ales printre studenți care vin din alte limbaje:

- ▶ **„Java este interpretat, deci lent.”** Fals în practică: compilarea JIT aduce Java aproape de viteza codului nativ pentru majoritatea aplicațiilor.
- ▶ **„Java și JavaScript sunt înrudite.”** Fals: dincolo de asemănarea numelui (motivată istoric de marketing), cele două limbaje au sintaxe similare superficial, dar semantici, modele de execuție și ecosisteme complet diferite.
- ▶ **„Java = applet-uri.”** Applet-urile sunt o tehnologie părăsită de mai bine de un deceniu; Java modern se folosește în cu totul alte contexte (backend, Android, sisteme distribuite).
- ▶ **„Java nu are gestionare automată a memoriei ca și C++.”** Fals: (*garbage collection*) este parte integrantă a JVM-ului încă de la prima versiune.

## 1.2 Instalarea și configurarea JDK-ului

Pentru a scrie și rula cod Java avem nevoie de JDK (JDK) — *Java Development Kit*. Acesta include:

- **Compilatorul** (`javac`), care traduce fișiere `.java` în bytecode (`.class`);
- **Mașina virtuală** (`java`), care execută bytecode-ul;
- **Biblioteca standard**, mii de clase organizate în pachete (`java.lang`, `java.util`, `java.io` etc.);
- **Instrumente auxiliare**: `javadoc` (generare documentație), `jshell` (REPL interactiv), `jar` (arhivare), printre altele.

#### Notă

Nu confundați JDK-ul cu JRE (JRE, *Java Runtime Environment*), care conținea doar mașina virtuală și biblioteca standard, fără instrumentele de dezvoltare. Începând cu Java 11, Oracle nu mai distribuie JRE separat — se distribuie doar JDK-ul complet, chiar și pentru mașini destinate exclusiv execuției.

### 1.2.1 Descărcarea și instalarea JDK-ului

Recomandăm folosirea unei distribuții OpenJDK gratuite și open-source (de exemplu Eclipse Temurin sau distribuția Oracle OpenJDK), în locul distribuției comerciale Oracle, ale cărei condiții de licențiere pentru uz în producție sunt mai restrictive.<sup>3</sup>

După instalare, verificăm configurarea corectă din linia de comandă:

```
1 $ java -version
2 openjdk version "25" 2026-...
3 $ javac -version
4 javac 25
```

Dacă aceste comenzi nu sunt recunoscute, variabila de mediu `PATH` nu include directorul bin al instalării JDK — o problemă frecventă la prima configurare, mai ales pe Windows.

### 1.2.2 Structura unei instalări JDK

O instalare JDK tipică conține, printre altele:

- `bin/` — executabilele (`java`, `javac`, `jshell`, `jar`, `javadoc` ...);
- `lib/` — biblioteci interne folosite de instrumentele JDK;
- `jmods/` — modulele platformei (vezi Capitolul 24, despre Java Platform Module System).

Spre deosebire de versiunile mai vechi de Java, biblioteca standard nu mai este distribuită ca un singur fișier monolitic (`rt.jar`), ci este organizată modular — o schimbare introdusă în Java 9 despre care vom discuta pe larg mai târziu în curs.

## 1.3 Utilizarea instrumentelor din linia de comandă

Înainte de a folosi orice IDE, este esențial să înțelegem ce se întâmplă „sub capotă” atunci când compilăm și rulăm un program Java. Vom scrie primul nostru program:

3: Pentru scopurile acestui curs, orice distribuție OpenJDK 25 (LTS) este potrivită. Verificați compatibilitatea cu sistemul de operare pe care lucrați.

**Listing 1.1:** Verificarea instalării JDK-ului

```

1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Salut, lume!");
4     }
5 }

```

Listing 1.2: Primul program Java — HelloWorld.java

#### Notă

Începând cu Java 21 (ca funcționalitate *preview*, stabilizată complet mai târziu), este posibilă și o formă simplificată, fără clasă explicită și fără modificatori, gândită pentru scripturi scurte și pentru introducerea în programare:

```

1 void main() {
2     IO.println("Salut, lume!");
3 }

```

Vom reveni asupra acestei sintaxe simplificate în Capitolul 4, dar pe parcursul cursului vom folosi în general forma completă, cu clasă și metodă `main` explicite, pentru compatibilitate cu majoritatea codului Java existent.

Compilarea și execuția din linia de comandă:

```

1 $ javac HelloWorld.java
2 $ java HelloWorld
3 Salut, lume!

```

Listing 1.3: Compilare și execuție manuală

Comanda `javac` produce fișierul `HelloWorld.class`, conținând bytecode-ul; comanda `java` pornește JVM-ul, încarcă această clasă și invocă metoda `main`.

#### Notă

Pentru fișiere sursă unice, precum cel de mai sus, Java permite din versiunile recente rularea directă, fără pasul explicit de compilare:

```

1 $ java HelloWorld.java
2 Salut, lume!

```

Această comandă compilează fișierul în memorie și îl execută imediat — utilă pentru scripturi și prototipare rapidă, dar echivalentă, conceptual, cu pașii separați de mai sus.

## 1.4 Un mediu de dezvoltare integrat (IDE)

Deși linia de comandă este suficientă pentru programe mici, un IDE (IDE, *Integrated Development Environment*) devine indispensabil pe măsură ce proiectele cresc: completare automată de cod, depanare vizuală, refactorizare, integrare cu instrumente de build (Maven, Gradle) și control de versiuni.

Cele mai folosite IDE-uri pentru Java sunt:

- **IntelliJ IDEA** (Community Edition, gratuită, sau Ultimate Edition, gratuită cu cont de student) — recomandarea noastră pentru acest curs, datorită ergonomiei și suportului excelent pentru funcționalitățile Java moderne;
- **Eclipse** — open-source, foarte răspândit istoric în mediul enterprise;

**Notă**

Indiferent de IDE-ul ales, la seminare/laboratoare veți avea nevoie să configurați corect *JDK-ul de proiect (Project SDK)*, de obicei setat implicit la versiunea instalată în sistem. O configurare greșită la acest pas este cea mai frecventă sursă de erori de tipul „merge la mine, dar nu la coleg”.

## 1.5 JShell — interpretorul interactiv al Java

Introdus în Java 9, `jshell` este un REPL (*Read-Eval-Print Loop*) pentru Java: permite scrierea și evaluarea de expresii și instrucțiuni Java, linie cu linie, fără a fi nevoie de o clasă completă cu metodă `main`. Este util pentru experimentare rapidă, verificarea comportamentului unei metode din bibliotecă, sau pur și simplu pentru a explora limbajul în timpul cursului.

Listing 1.4: O sesiune scurtă `jshell`

```

1 | $ jshell
2 | Welcome to JShell -- Version 25
3 | For an introduction type: /help intro
4 |
5 | jshell> int x = 5;
6 | x ==> 5
7 |
8 | jshell> x * 2
9 | $2 ==> 10
10 |
11 | jshell> System.out.println("Salut din JShell!");
12 | Salut din JShell!
13 |
14 | jshell> /exit
15 | Goodbye

```

Observați două lucruri specifice JShell-ului, care nu ar fi valide într-un fișier `.java` obișnuit:

- Instrucțiunile nu necesită să fie încapsulate într-o clasă sau o metodă;
- Expresiile fără atribuire (precum `x * 2`) sunt evaluate automat, iar rezultatul este stocat într-o variabilă temporară generată (\$2) și afișat.

Comenzile care încep cu `/` (precum `/exit`, `/help`, `/list`, `/vars`) sunt comenzi ale JShell-ului însuși, nu cod Java — controlează sesiunea, nu programul evaluat.

**Notă**

JShell este un excelent instrument didactic, dar nu înlocuiește un proiect real: nu gestionează dependențe multiple, nu are un ciclu de build propriu-zis și starea sesiunii se pierde la închidere (cu excepția cazului în care salvați explicit istoricul). Recomandăm folosirea lui punctual, pentru verificări rapide, și nu ca mediu principal de dezvoltare.

**Rezumat**

- Java este o *platformă*: limbaj + mașină virtuală + bibliotecă standard, nu doar un limbaj de programare.

- ▶ Codul sursă este compilat în *bytecode*, executat de JVM prin interpretare și compilare JIT selectivă — acesta este mecanismul din spatele portabilității „write once, run anywhere”.
- ▶ De la Java 9, lansările au ciclu de șase luni, cu versiuni LTS periodice; acest curs folosește Java 25 (LTS).
- ▶ JDK-ul conține compilatorul, mașina virtuală, biblioteca standard și instrumente auxiliare (javadoc, jshell, jar).
- ▶ Vom lucra atât din linia de comandă (javac, java), cât și dintr-un IDE (recomandat: IntelliJ IDEA), plus jshell pentru experimentare rapidă.

### Exerciții

1. Instalați o distribuție OpenJDK 25 pe sistemul vostru și verificați instalarea cu `java -version` și `javac -version`.
2. Scrieți, compilați și rulați programul `HelloWorld.java` din linia de comandă, fără a folosi un IDE.
3. Deschideți o sesiune `jshell` și evaluați câteva expresii aritmetice simple, apoi verificați cu `/vars` ce variabile temporare au fost create.
4. Documentați-vă (pe scurt) diferența dintre modelul de lansare al Java înainte și după versiunea 9. De ce credeți că schimbarea la un ciclu de șase luni a fost necesară pentru ecosistem?



# Tipuri primitive. Operatori. Controlul fluxului

# 2

## Introducere

Am văzut în capitolul anterior cum se compilează și rulează un program Java. În acest capitol intrăm efectiv în sintaxa limbajului: cum declarăm date, cum le combinăm prin operatori, cum controlăm fluxul de execuție al unui program și cum lucrăm cu colecții indexate de valori — array-urile. Acesta este, alături de Capitolul 4, fundamentul peste care se construiește tot restul cursului.

## 2.1 Primul program Java, revizitat

Reluăm programul din capitolul anterior pentru a-i analiza, de această dată, fiecare componentă sintactică:

```
1 public class HelloWorld {  
2     public static void main(String[] args) {  
3         System.out.println("Salut, lume!");  
4     }  
5 }
```

Listing 2.1: Structura minimală a unui program Java

- Numele fișierului sursă **trebuie** să coincidă cu numele clasei publice (HelloWorld.java pentru clasa HelloWorld) — o regulă impusă de compilator, nu doar o convenție.
- `public static void main(String[] args)` este *punctul de intrare* pe care JVM-ul îl caută la execuție; semnătura exactă este fixată de specificația limbajului.
- Corpul metodei este delimitat de acolade; fiecare instrucțiune se încheie cu punct și virgulă.

### 2.1.1 Comentarii

Java oferă trei forme de comentarii:

```
1 // Comentariu pe o singură linie  
2  
3 /* Comentariu  
4 pe mai multe linii */  
5  
6 /**  
7  * Comentariu de documentatie (Javadoc) - vezi Capitolul 5  
8  */
```

## Notă

Comentariile de documentație (`/** ... */`) nu sunt doar comentarii obișnuite — instrumentul javadoc le extrage și le folosește pentru a genera documentație HTML. Vom reveni asupra lor pe larg în Capitolul 5, dar e util să recunoașteți de pe acum sintaxa.

## 2.1.2 Convenții de stil

Convențiile de mai jos nu sunt impuse de compilator, ci de comunitate — respectarea lor este însă așteptată de la orice cod Java profesionist:

- ▶ Numele claselor încep cu literă mare, stil *PascalCase* (HelloWorld, BankAccount);
- ▶ Numele variabilelor și metodelor încep cu literă mică, stil *camelCase* (totalAmount, computeInterest);
- ▶ Numele constantelor se scriu integral cu majuscule, cu underscore ca separator (MAX\_VALUE);
- ▶ Indentare consecventă (de obicei 4 spații, nu tab-uri) pentru corpurile de bloc.

## 2.2 Tipuri de date primitive

Java este un limbaj cu **tipizare statică** (*statically typed*): fiecare variabilă are un tip declarat explicit, verificat la compilare. Spre deosebire de tipurile obiect (pe care le vom studia în Capitolul 4), de tip primitiv sunt cele opt tipuri fundamentale, **nu** sunt obiecte și au dimensiune fixă, independentă de platformă — o garanție de portabilitate specifică Java, absentă din C/C++.

**Tabela 2.1:** Tipurile primitive din Java și dimensiunile lor, fixate prin specificație.

| Tip     | Dimensiune | Domeniu                         |
|---------|------------|---------------------------------|
| byte    | 1 octet    | −128 ...127                     |
| short   | 2 octeți   | −32 768 ...32 767               |
| int     | 4 octeți   | ≈ −2.1 până la 2.1 miliarde     |
| long    | 8 octeți   | ≈ ±9.2 × 10 <sup>18</sup>       |
| float   | 4 octeți   | virgulă mobilă, precizie simplă |
| double  | 8 octeți   | virgulă mobilă, precizie dublă  |
| char    | 2 octeți   | un caracter Unicode (UTF-16)    |
| boolean | —          | true sau false                  |

### Notă

Dimensiunea tipului boolean nu este specificată explicit — reprezentarea internă e lăsată la latitudinea JVM-ului. Din perspectiva programatorului, o valoare boolean poate fi doar true sau false; spre deosebire de C, **nu** există conversie implicită între boolean și tipuri întregi.

### 2.2.1 Literalii numerici

**Listing 2.2:** Literalii numerici în diverse baze și cu separatori

```

1 | int hex = 0xFF;           // hexazecimal -> 255
2 | int oct = 010;           // octal -> 8
3 | int bin = 0b1010;        // binar -> 10
4 | long big = 3_000_000_000L; // underscore ca separator vizual;
   |     sufix L pentru long
5 | double pi = 3.14159;
6 | float f = 3.14f;         // sufix f, altfel literalul e
   |     interpretat ca double

```

Sufixul `L` (sau `l`, descurajat din cauza confuziei vizuale cu cifra 1) marchează un literal `long`; sufixul `f`/`F` marchează un literal `float`. Fără aceste sufixe, un literal întreg este implicit `int`, iar unul cu virgulă este implicit `double`.

### 2.2.2 Unicode și tipul `char`

Java a fost printre primele limbaje larg răspândite care au adoptat Unicode ca set de caractere nativ, nu ca o extensie ulterioară. Un `char` reprezintă o unitate de cod UTF-16 pe 16 biți:

```
1 char litera = 'A';
2 char unicode = '\u00E9'; // 'e' cu accent ascuțit, scris ca
   escape Unicode
```

#### Notă

Caracterele din afara planului Unicode de bază (de exemplu unele emoji sau simboluri istorice) necesită **două** valori `char` (o pereche surrogat), fiindcă nu încap într-un singur bloc de 16 biți. Această complicație este motivul pentru care manipularea corectă de text la nivel de „caracter vizibil” este, în general, mai subtilă decât pare la prima vedere — vom reveni asupra acestui subiect când discutăm clasa `String` pe larg.

## 2.3 Variabile, constante și tipuri enumerate

### 2.3.1 Declararea variabilelor

```
1 int count = 0;
2 double price; // declarata, dar neinitializata
3 price = 19.99; // atribuire ulterioara
4 final int MAX = 100; // constanta - nu poate fi reatribuita
```

Modificatorul `final` marchează o variabilă ca fiind o **constantă**: odată inițializată, valoarea ei nu mai poate fi schimbată. Compilatorul semnalează eroare la orice încercare de reatribuire.

### 2.3.2 Tipuri enumerate

Pentru un set fix, cunoscut la compilare, de valori posibile, Java oferă tipul `enum` — o alternativă net superioară folosirii unor constante întregi „magice”:

```
1 enum Marime { MIC, MEDIU, MARE, FOARTE_MARE }
2
3 Marime m = Marime.MEDIU;
4
5 switch (m) {
6     case MIC -> System.out.println("Marime mica");
7     case MEDIU -> System.out.println("Marime medie");
8     case MARE, FOARTE_MARE -> System.out.println("Marime mare");
9 }
```

Listing 2.3: Un tip enumerat simplu

Vom relua tipurile enum pe larg în Capitolul 6 (unde vor apărea ca un caz special de clasă), dar e util să le recunoașteți de acum ca alternativă tipizată la constantele întregi.

## 2.4 Operatori

### 2.4.1 Operatori aritmetici și de atribuire

```
1 | int a = 7, b = 2;
2 | int suma = a + b;           // 9
3 | int diferenta = a - b;      // 5
4 | int produs = a * b;         // 14
5 | int cat = a / b;            // 3 (impartire intreaga!)
6 | int rest = a % b;           // 1
7 | double catExact = a / (double) b; // 3.5, dupa conversie
8 |                               // explicita
```

#### Notă

Împărțirea a două valori `int` produce un rezultat `int` — partea fracționară este trunchiată, nu rotunjită. Aceasta este una dintre cele mai frecvente surse de erori pentru începători: `7 / 2` produce 3, nu 3.5. Pentru a obține rezultatul cu virgulă mobilă, cel puțin unul dintre operanzi trebuie convertit explicit la `double` sau `float` **înainte** de împărțire.

Operatorii de atribuire compusă (`+=`, `-=`, `*=`, `/=`, `%=`) combină o operație aritmetică cu atribuirea:

```
1 | int x = 10;
2 | x += 5; // echivalent cu x = x + 5; -> x devine 15
3 | x *= 2; // x devine 30
```

### 2.4.2 Operatori relaționali și logici

```
1 | boolean egal = (a == b);
2 | boolean diferit = (a != b);
3 | boolean siLogic = (a > 0) && (b > 0); // evaluare lenesa
4 |                                     // (short-circuit)
5 | boolean sauLogic = (a > 0) || (b > 0);
6 | boolean negatie = !(a > 0);
```

#### Notă

Operatorii `&&` și `||` folosesc *evaluare leneșă* (*short-circuit*): în `a && b`, dacă `a` este fals, `b` nici măcar nu mai este evaluat. Această proprietate este esențială pentru cod defensiv, de exemplu `obj != null && obj.metoda()` — ordinea contează, altfel al doilea operand ar arunca o excepție dacă `obj` este `null`.

### 2.4.3 Conversii și cast-uri

Java distinge între **conversii implicite** (*widening*, fără pierdere de informație) și **conversii explicite** (*cast*, care pot pierde informație):

```

1 | int i = 100;
2 | long l = i; // conversie implicita: int -> long,
3 | // fara pierderi
4 | double d = l; // conversie implicita: long -> double
5 |
6 | double x = 3.99;
7 | int truncat = (int) x; // cast explicit -> 3
8 | // (trunchiere, nu rotunjire!)
9 |
10 | long mare = 3_000_000_000L;
11 | int trunchiat = (int) mare; // cast explicit -> rezultat
12 | // "surprinzator" (overflow)

```

### Notă

Un *cast* de la un tip cu domeniu mai mare la unul cu domeniu mai mic (de exemplu long → int) **nu** generează nicio eroare sau avertisment la execuție — pur și simplu trunchiază biții cei mai semnificativi, ceea ce poate produce rezultate greșite silențios. Este responsabilitatea programatorului să se asigure că valoarea încapă în tipul destinație.

## 2.4.4 Expresii switch

Pe lângă instrucțiunea clasică switch (§2.5.3), Java oferă, ca **expresie**, o formă modernă care returnează direct o valoare:

```

1 | int zi = 3;
2 | String numeZi = switch (zi) {
3 |     case 1 -> "Luni";
4 |     case 2 -> "Marti";
5 |     case 3 -> "Miercuri";
6 |     default -> "Zi necunoscuta";
7 | };

```

Listing 2.4: switch ca expresie

Spre deosebire de forma clasică, ramurile cu -> nu au *fallthrough* (nu se „scurg” automat în ramura următoare) și pot fi folosite direct într-o atribuire sau ca argument.

## 2.5 Structuri de control

### 2.5.1 Instrucțiuni condiționale

```

1 | if (varsta >= 18) {
2 |     System.out.println("Major");
3 | } else if (varsta >= 14) {
4 |     System.out.println("Minor cu discernamant");
5 | } else {
6 |     System.out.println("Minor");
7 | }

```

### 2.5.2 Bucle

```

1 | // for clasic, cu contor explicit
2 | for (int i = 0; i < 5; i++) {
3 |     System.out.println(i);
4 | }

```

Listing 2.5: Cele trei forme principale de bucle

```

5
6 // while - verifica conditia inainte de fiecare iteratie
7 int n = 5;
8 while (n > 0) {
9     System.out.println(n);
10    n--;
11 }
12
13 // do-while - executa cel putin o data, verifica conditia dupa
14 int m = 0;
15 do {
16     System.out.println(m);
17     m++;
18 } while (m < 3);

```

#### Notă

Alegerea între `while` și `do-while` depinde de o singură întrebare: *trebuie corpul buclei executat cel puțin o dată, indiferent de condiție?* Dacă da, `do-while` este forma corectă; altfel, `while` este preferat, fiind mai ușor de citit.

### 2.5.3 switch clasic

Listing 2.6: switch clasic, cu `fallthrough`

```

1 int luna = 2;
2 switch (luna) {
3     case 12:
4     case 1:
5     case 2:
6         System.out.println("Iarna");
7         break;
8     case 3:
9     case 4:
10    case 5:
11        System.out.println("Primavara");
12        break;
13    default:
14        System.out.println("Alt anotimp");
15 }

```

#### Notă

Omiterea instrucțiunii `break` este o sursă clasică de erori: execuția trece automat (*fallthrough*) în ramura următoare, indiferent de valoarea comparată. Acest comportament este uneori folosit intenționat (ca în exemplul de mai sus, unde 12, 1 și 2 conduc la același rezultat), dar de cele mai multe ori e o greșeală de neatenție. Expresiile `switch` din §2.4.4 elimină complet acest risc.

### 2.5.4 break și continue

```

1 for (int i = 0; i < 10; i++) {
2     if (i == 5) break; // iese complet din bucla
3     if (i % 2 == 0) continue; // sare direct la urmatoarea
4                                     // iteratie
5     System.out.println(i);
6 }

```

Pentru bucle imbricate, o **etichetă** (*label*) permite un break/continue care vizează explicit bucla exterioară:

```

1 cautare:
2 for (int i = 0; i < 5; i++) {
3     for (int j = 0; j < 5; j++) {
4         if (i * j > 6) {
5             break cautare; // iese din AMBELE bucle,
6                             // nu doar din cea interioara
7         }
8     }
9 }

```

Listing 2.7: break etichetat, pentru bucle imbricate

## 2.6 Array-uri

Un array este o structură de date de dimensiune fixă, care stochează elemente de același tip, accesibile prin index numeric (începând de la 0).

### 2.6.1 Declarare, alocare, acces

```

1 int[] numere = new int[5];
2     // array de 5 intregi, initializati implicit cu 0
3 numere[0] = 10;
4 numere[1] = 20;
5
6 int[] literali = { 1, 2, 3, 4, 5 };
7     // initializare directa cu literali
8
9 System.out.println(numere.length);
10    // 5 - proprietate, NU metoda (fara paranteze!)

```

#### Notă

Accesul la un index în afara domeniului valid (0 până la lungime - 1) nu produce un comportament nedefinit ca în C, ci aruncă o excepție de tip `ArrayIndexOutOfBoundsException` la execuție. Vom discuta pe larg mecanismul excepțiilor în Capitolul 9.

### 2.6.2 Bucla for-each

Pentru parcurgerea simplă, fără nevoia indexului explicit, forma for-each este preferată:

```

1 int[] numere = { 10, 20, 30 };
2 for (int n : numere) {
3     System.out.println(n);
4 }

```

### 2.6.3 Copierea array-urilor

Atribuirea directă (`a2 = a1`) copiază doar **referința**, nu conținutul — ambele variabile ajung să indice către același array din memorie. Pentru o copie reală a conținutului:

```

1 int[] original = { 1, 2, 3 };
2
3 int[] copie = Arrays.copyOf(original, original.length);
4 copie[0] = 99; // nu afecteaza 'original'
5
6 int[] copiePartiala = Arrays.copyOfRange(original, 1, 3);
7     // { 2, 3 }

```

### 2.6.4 Sortare și alte operații utilitare

Clasa utilitară `java.util.Arrays` oferă operații comune, deja optimizate, pentru array-uri:

```

1 int[] valori = { 5, 3, 8, 1, 9 };
2 Arrays.sort(valori); // sortare in loc, crescator
3 int pozitie = Arrays.binarySearch(valori, 8);
4     // cautare binara (necesita array sortat!)
5 System.out.println(Arrays.toString(valori));
6     // afisare lizibila: [1, 3, 5, 8, 9]

```

#### Notă

`System.out.println(valori)` (fără `Arrays.toString`) afișează ceva de forma `[I@1b6d3586` — reprezentarea implicită moștenită din `Object`, nu conținutul array-ului. Aceasta e o capcană foarte frecventă pentru începători; vom explica exact de ce se întâmplă acest lucru când discutăm metoda `toString` în Capitolul 6.

### 2.6.5 Array-uri multidimensionale și neregulate

Java nu are array-uri multidimensionale „native” (ca în *Fortran*); un array bidimensional este, de fapt, un **array de array-uri**:

Listing 2.8: Array bidimensional drept-unghiular

```

1 int[][] matrice = new int[3][4]; // 3 linii, 4 coloane
2 matrice[1][2] = 7;
3
4 for (int[] linie : matrice) {
5     for (int valoare : linie) {
6         System.out.print(valoare + " ");
7     }
8     System.out.println();
9 }

```

Deoarece fiecare „linie” este, de fapt, un array independent, liniile pot avea **lungimi diferite** — un array *neregulat* (*ragged array*):

Listing 2.9: Array neregulat (ragged)

```

1 int[][] triunghi = new int[5][];
2 for (int i = 0; i < triunghi.length; i++) {
3     triunghi[i] = new int[i + 1]; // linia i are i+1 elemente
4 }

```

#### Rezumat

- Java are opt tipuri primitive, cu dimensiuni fixe garantate prin specificație (spre deosebire de C/C++); `char` reprezintă o unitate de cod UTF-16.
- Împărțirea între valori `int` este întreagă (trunchiere, nu rotunjire); conversia la `double/float` trebuie făcută explicit înainte

de operație.

- ▶ Operatorii `&&/||` folosesc evaluare leneșă; cast-urile de îngustare (*narrowing*) pot trunchia biți fără avertisment.
- ▶ Expresiile `switch` moderne (`->`) elimină riscul de *fallthrough* al formei clasice.
- ▶ Array-urile au dimensiune fixă și lungime accesibilă prin proprietatea `.length` (fără paranteze); atribuirea directă copiază doar referința, nu conținutul.
- ▶ Un array bidimensional este, conceptual, un array de array-uri — de unde rezultă posibilitatea array-urilor neregulate (*ragged*).

### Exerciții

1. Scrieți un program care citește un număr întreg `n` și afișează dacă este par sau impar, folosind operatorul `%`.
2. Rescrieți instrucțiunea `switch` clasică din §2.5.3 ca expresie `switch` modernă, care returnează direct un `String`.
3. Scrieți o metodă care primește un array de numere întregi și returnează un nou array conținând doar elementele pare, fără să modifice array-ul original.
4. Creați un array neregulat cu 5 linii, unde linia `i` conține exact `i + 1` valori egale cu `i` (de exemplu, linia 2 conține `[2, 2, 2]`). Afișați rezultatul folosind bucle `for-each` imbricate.
5. Explicați, cu un exemplu concret, de ce `(int) 3_000_000_000L` produce un rezultat neașteptat. Ce reprezintă, la nivel de biți, valoarea obținută?



# Șiruri de caractere. Input/Output. Numere mari

# 3

## Introducere

Cu tipurile primitive, operatorii și structurile de control din Capitolul 2 putem deja exprima algoritmi simpli. Ne lipsesc însă două ingrediente esențiale pentru programe utile: capacitatea de a manipula **text** și de a **comunica** cu utilizatorul (citire/afișare).

## 3.1 Clasa `String`

Spre deosebire de tipurile primitive discutate în capitolul anterior, un șir de caractere (`String`) este un obiect — prima întâlnire din acest curs cu o clasă din biblioteca standard folosită intensiv [3].

### 3.1.1 Concatenare și conversii

```
1 String prenume = "Ana";
2 String nume = "Popescu";
3 String numeComplet = prenume + " " + nume; // "Ana Popescu"
4
5 int varsta = 20;
6 String mesaj = "Varsta: " + varsta;
7 // "Varsta: 20" conversie automata int -> String
```

Listing 3.1: Concatenarea șirurilor cu operatorul +

## Notă

Operatorul + aplicat unui `String` și oricărui alt tip declanșează automat conversia acelui operand la `String` (prin apelul implicit al metodei `toString()`, discutată pe larg în Capitolul 6). Această comoditate are însă un cost de performanță în bucle cu multe concatenări — vezi §3.2.2.

### 3.1.2 Indexare și metode uzuale

```
1 String s = "Programare in Java";
2
3 int lungime = s.length();           // 19
4 char c = s.charAt(0);               // 'P'
5 String sub = s.substring(0, 11);    // "Programare "
6 int poz = s.indexOf("Java");        // 14
7 boolean contine = s.contains("in"); // true
8 String majuscule = s.toUpperCase(); // "PROGRAMARE IN JAVA"
9 String[] cuvinte = s.split(" ");    // "Programare",
                                     // "in", "Java" String
10
    curatat = " spatii ".strip(); // "spatii" (elimina//
    spatiile marginale)
```

Listing 3.2: Metode frecvent folosite ale clasei `String`

## Notă

Indexarea în `String` este bazată pe unități de cod UTF-16 (`char`), nu pe „caractere vizibile” — o distincție relevantă mai ales pentru text cu

caractere din afara planului Unicode de bază (vezi §2.2.2 din capitolul anterior). Pentru textul folosit în majoritatea aplicațiilor de curs, această subtilitate nu ridică probleme practice.

### 3.1.3 Imutabilitate

#### Definiție — Imutabilitate

Un obiect este *imutabil* dacă starea sa internă nu poate fi modificată după construcție. Toate metodele clasei `String` care „par” să modifice șirul (`toUpperCase`, `substring`, `replace` etc.) returnează de fapt un **obiect nou**, lăsând șirul original neschimbat.

Listing 3.3: `String` este imutabil

```
1 String original = "java";
2 String majuscule = original.toUpperCase();
3
4 System.out.println(original); // "java" -- neschimbat!
5 System.out.println(majuscule); // "JAVA" -- un obiect nou
```

#### Notă

O greșeală frecventă la începători este să scrie `s.toUpperCase()`; fără a reține rezultatul — instrucțiunea nu are niciun efect vizibil, fiindcă `s` rămâne neschimbat. Rezultatul trebuie fie reatribuit (`s = s.toUpperCase()`), fie folosit direct.

Datorită imutabilității, literalii `String` identici pot fi **partajați** în memorie, într-o zonă specială numită *string pool* — o optimizare posibilă tocmai pentru că un șir imutabil nu poate fi alterat pe ascuns de una dintre referințele care îl partajează.

### 3.1.4 Egalitate: `==` versus `equals`

Aceasta este, probabil, cea mai importantă distincție sintactică din acest capitol:

Listing 3.4: `==` compară referințe, `equals` compară conținut

```
1 String a = "salut";
2 String b = "salut";
3 String c = new String("salut");
4
5 System.out.println(a == b); // true - ambele din
   string pool, aceeași referință
6 System.out.println(a == c); // false - c e un obiect
   NOU, altă referință
7 System.out.println(a.equals(c)); // true - conținutul e
   identic
```

#### Notă

Regula practică, valabilă nu doar pentru `String`, ci pentru **orice** tip obiect: `==` compară dacă două referințe indică *același obiect* în memorie; `equals` compară dacă două obiecte au *conținut echivalent*, conform definiției din acea clasă. Pentru tipuri primitive, `==` este singura opțiune și compară direct valorile. Vom relua acest subiect riguros, inclusiv implementarea metodei `equals` pentru clase proprii, în Capitolul 6.

## 3.2 Text blocks și StringBuilder

### 3.2.1 Text blocks

Pentru text pe mai multe linii (JSON, HTML, SQL încorporat etc.), Java oferă *text blocks* — introduse ca funcționalitate stabilă începând cu Java 17:

```
1 String json = """
2 {
3     "nume": "Ana",
4     "varsta": 20
5 }
6 """;
```

**Listing 3.5:** Un text block, pentru text multi-linie

Indentarea comună tuturor liniilor (determinată automat de indentarea celei mai puțin indentate linii, inclusiv delimitatorul final) este eliminată la compilare — textul rezultat nu conține indentarea sursei Java, doar cea intenționată logic.

### 3.2.2 StringBuilder

#### Notă

Deoarece String este imutabil, fiecare concatenare (`s = s + "x";`) creează un obiect nou, copiind tot conținutul anterior. Într-o buclă cu multe iterații, acest lucru are cost pătratic în timp. Pentru construcția incrementală de text, folosiți `StringBuilder` — un obiect **mutabil**, gândit exact pentru acest scop.

```
1 StringBuilder sb = new StringBuilder();
2 for (int i = 1; i <= 5; i++) {
3     sb.append(i).append(", ");
4 }
5 sb.setLength(sb.length() - 2);
6 // elimina ultima virgula si spatiu
7 String rezultat = sb.toString(); // "1, 2, 3, 4, 5"
```

**Listing 3.6:** Construirea eficientă a unui șir cu `StringBuilder`

Alte metode utile: `insert(pozitie, text)`, `delete(start, end)`, `reverse()` — toate modifică **în loc** conținutul, spre deosebire de metodele corespunzătoare din `String`.

## 3.3 Input/Output de bază

### 3.3.1 Citirea datelor de la tastatură

Clasa `Scanner`, din pachetul `java.util`, este cea mai simplă cale de a citi date introduse de utilizator de la consolă:

```
1 import java.util.Scanner;
2
3 Scanner sc = new Scanner(System.in);
4
5 System.out.print("Introduceți numele: ");
6 String nume = sc.nextLine();
7
```

**Listing 3.7:** Citirea de date cu `Scanner`

```

8 System.out.print("Introduceți varsta: ");
9 int varsta = sc.nextInt();
10
11 System.out.println("Salut, " + nume + "! Ai " + varsta + "
    ani.");

```

#### Notă

Amestecarea apelurilor `nextInt()/nextDouble()` cu `nextLine()` este o capcană clasică: metodele `nextX()` nu consumă și caracterul de linie nouă rămas în flux, care este apoi „văzut” de următorul `nextLine()` ca o linie goală. Soluția obișnuită este un apel suplimentar, gol, la `sc.nextLine()` imediat după un `nextInt()/nextDouble()`, pentru a „consuma” linia nouă rămasă.

Dacă utilizatorul introduce un text care nu poate fi convertit la tipul cerut (de exemplu litere acolo unde se așteaptă un număr), `Scanner` aruncă o excepție de tip `InputMismatchException` — vom învăța să gestionăm astfel de situații corect în Capitolul 9.

### 3.3.2 Formatarea ieșirii

Listing 3.8: `printf` și `String.format` pentru ieșire formatată

```

1 double pret = 19.5;
2 int cantitate = 3;
3
4 System.out.printf("Total: %.2f lei pentru %d bucati%n", pret
    * cantitate, cantitate);
5 // Total: 58.50 lei pentru 3 bucati
6
7 String linie = String.format("%-10s | %5d", "Mere", 42);
8 // "Mere | 42"

```

Tabela 3.1: Specificatorii de format cei mai des folosiți cu `printf/String.format`.

| Specificator       | Semnificație                                                    |
|--------------------|-----------------------------------------------------------------|
| <code>%d</code>    | întreg (zecimal)                                                |
| <code>%f</code>    | virgulă mobilă (implicit 6 zecimale)                            |
| <code>%.2f</code>  | virgulă mobilă, 2 zecimale                                      |
| <code>%s</code>    | șir de caractere (sau orice obiect, via <code>toString</code> ) |
| <code>%-10s</code> | alinieare la stânga, lățime minimă 10                           |
| <code>%n</code>    | separator de linie, portabil între sisteme de operare           |

#### Notă

Folosiți `%n`, nu `\n`, pentru saltul de linie în șirurile de format — `%n` produce automat secvența corectă de sfârșit de linie pentru sistemul de operare curent (`\r\n` pe Windows, `\n` pe Linux/macOS), în timp ce `\n` este mereu identic, indiferent de platformă.

## 3.4 Numere mari

Tipurile primitive numerice au domenii fixe (Tabelul 2.1, Capitolul 2). Pentru calcule care depășesc aceste limite — criptografie, calcule financiare exacte, combinatorică — Java oferă două clase din pachetul `java.math`.

### 3.4.1 BigInteger

BigInteger reprezintă numere întregi de precizie **arbitrară**, limitate doar de memoria disponibilă:

```
1 import java.math.BigInteger;
2
3 BigInteger a = new BigInteger("123456789012345678901234567");
4 BigInteger b = BigInteger.valueOf(2);
5
6 BigInteger suma = a.add(b);
7 BigInteger produs = a.multiply(b);
8 BigInteger factorial20 = calculeazaFactorial(20);
9     // vezi mai jos
10
11 System.out.println(produs);
```

Listing 3.9: Aritmetică cu precizie arbitrară folosind BigInteger

#### Notă

BigInteger (la fel ca String) este **imutabil** — fiecare operație (add, multiply etc.) returnează un obiect nou. Nu există operatori aritmetici supraîncărcați pentru BigInteger (Java nu permite supraîncărcarea de operatori pentru clase definite de utilizator sau bibliotecă) — toate operațiile se fac prin apeluri de metodă explicite.

### 3.4.2 BigDecimal

BigDecimal reprezintă numere zecimale cu precizie **exactă**, esențial pentru calcule financiare, unde erorile de rotunjire ale tipurilor float/double sunt inacceptabile:

```
1 import java.math.BigDecimal;
2
3 BigDecimal pret = new BigDecimal("19.99");
4 BigDecimal cantitate = new BigDecimal("3");
5 BigDecimal total = pret.multiply(cantitate);
6
7 System.out.println(total);
8     // 59.97 - exact, fara erori de reprezentare
```

Listing 3.10: Aritmetică zecimală exactă cu BigDecimal

#### Notă

Constructorul BigDecimal trebuie apelat **cu un String** (new BigDecimal("19.99")), nu cu un literal double (new BigDecimal(19.99)). A doua formă moștenește imprecizia reprezentării binare a lui 19.99 ca double **înainte** de a fi convertită la BigDecimal, anulând practic avantajul de precizie pe care îl căutați. Aceasta este una dintre cele mai frecvente greșeli întâlnite în cod Java real, inclusiv în proiecte de producție.

#### Rezumat

- ▶ String este un obiect **imutabil**: orice metodă care „modifică” șirul returnează de fapt un obiect nou.
- ▶ == compară **referințe** (aceeași locație în memorie), equals compară **conținut** — o distincție valabilă pentru orice tip obiect, nu doar String.
- ▶ Pentru text pe mai multe linii, folosiți *text blocks* ("\" . . . \"");

pentru concatenare intensivă în bucle, folosiți `StringBuilder` (mutabil), nu operatorul `+` repetat.

- ▶ `Scanner` citește date de la consolă; atenție la amestecarea `nextInt()/nextDouble()` cu `nextLine()`.
- ▶ `printf/String.format` oferă control precis asupra formatării ieșirii; folosiți `%n`, nu `\n`, pentru portabilitate.
- ▶ `BigInteger` și `BigDecimal` oferă precizie arbitrară/exactă pentru situații unde tipurile primitive numerice nu sunt suficiente; `BigDecimal` trebuie construit din `String`, nu din `double`.

### Exerciții

1. Scrieți un program care citește un nume și o vârstă de la tastatură (folosind `Scanner`) și afișează un mesaj formatat corespunzător, folosind `printf`.
2. Scrieți o metodă care primește un `String` și returnează `true` dacă acesta este un *palindrom* (se citește la fel din ambele direcții), ignorând diferențele majuscule/minuscule.
3. Explicați, cu un exemplu de cod, diferența dintre `a == b` și `a.equals(b)` pentru două obiecte `String` construite explicit cu `new String(...)`.
4. Scrieți o metodă care calculează  $n!$  (factorial) pentru  $n = 25$ , folosind `BigInteger`. Explicați de ce tipul `long` nu ar fi suficient pentru acest calcul.
5. Folosind `BigDecimal`, scrieți un program care calculează suma a trei prețuri introduse ca șiruri de caractere (de exemplu "19.99", "5.50", "0.99") și afișează rezultatul exact. Repetați apoi calculul cu `double` și comentați diferența (dacă există) față de rezultatul cu `BigDecimal`.

# Obiecte și clase (I): încapsulare

## Introducere

Până acum am scris programe folosind doar tipuri primitive, șiruri de caractere și array-uri — suficient pentru algoritmi simpli, dar insuficient pentru a modela probleme complexe, cu multe date și comportamente înrudite. În acest capitol facem trecerea la programarea orientată pe obiecte (OOP): cum identificăm clasele dintr-o problemă dată, cum le definim, și cum protejăm datele lor interne printr-un principiu numit *încapsulare*. Vom continua acest fir în Capitolul 5 (organizarea codului în pachete) și în Capitolul 6 (relații de moștenire între clase).

## 4.1 Introducere în programarea orientată pe obiecte

### 4.1.1 Clase și obiecte

Programarea procedurală (cea practică până acum în acest curs) organizează un program ca o secvență de instrucțiuni care operează asupra unor date separate. Programarea orientată pe obiecte propune o organizare diferită: datele și comportamentele care le operează sunt grupate împreună, într-o unitate numită *clasă*.

#### Definiție — Clasă și obiect

O *clasă* este un șablon (o „rețetă”) care descrie structura și comportamentul unei categorii de entități: ce date reține fiecare instanță și ce operații poate efectua asupra lor. Un *obiect* este o instanță concretă a unei clase, cu propriile valori pentru datele descrise de acea clasă.

Relația dintre clasă și obiect este analoagă celei dintre un plan arhitectural și clădirile construite după el: planul (*clasa*) există o singură dată, dar pot exista oricâte clădiri (*obiecte*) construite conform lui, fiecare cu propria adresă, culoare a fațadei etc.

```
1 ContBancar cont1 = new ContBancar("Ana Popescu", 500.0);  
2 ContBancar cont2 = new ContBancar("Ion Ionescu", 1200.0);
```

Aici, ContBancar este clasa (șablonul), iar cont1 și cont2 sunt două obiecte distincte, create după același șablon, dar cu date proprii (titular, sold). Vom defini efectiv această clasă în Secțiunea 4.3.

**Listing 4.1:** O clasă simplă și două obiecte create după ea

### 4.1.2 Identificarea claselor; relații între clase

O întrebare naturală, la începutul oricărui proiect orientat pe obiecte, este: *ce clase ar trebui să existe?* O euristică simplă, dar surprinzător de eficientă, este să identificăm **substantivele** din descrierea problemei — acestea sunt de obicei candidați buni pentru clase, în timp ce **verbele** devin de obicei metode ale acestor clase.



```

5 | System.out.println(azi == mereReferinta); // true - aceeași
6 |                                           // referința

```

#### Notă

Această distincție este identică celei discutate pentru `String` în Subsecțiunea 3.1.4: de fapt, `String` este doar un caz particular de tip obiect, nu o excepție de la regulă. Comportamentul se aplică la fel oricărei clase, inclusiv celor pe care le vom defini noi înșine în Secțiunea 4.3.

### 4.2.2 Metode mutator și accesori

Folosim clasa `LocalDate` (din pachetul `java.time`, pe care îl vom studia detaliat în Capitolul 25) ca exemplu concret pentru două categorii de metode pe care le vom întâlni în aproape orice clasă:

```

1 | import java.time.LocalDate;
2 |
3 | LocalDate ziuaNasterii = LocalDate.of(2005, 3, 17);
4 |
5 | int an = ziuaNasterii.getYear(); // 2005 - metoda ACCESOR
6 | int luna = ziuaNasterii.getMonthValue(); // 3
7 | int zi = ziuaNasterii.getDayOfMonth(); // 17

```

Listing 4.3: Metode accesori (*getter*) ale clasei `LocalDate`

O metodă *accesor* (*getter*) doar citește o valoare din obiect, fără a-i modifica starea internă. O metodă *mutator* (*setter*), în schimb, modifică starea obiectului — însă, așa cum am văzut deja la `String` (Secțiunea 3.1), nu toate clasele din bibliotecă permit acest lucru:

```

1 | LocalDate maine = ziuaNasterii.plusDays(1); // NU modifica
2 |                                           // ziuaNasterii!
3 |
4 | System.out.println(ziuaNasterii); // 2005-03-17 - neschimbat
5 | System.out.println(maine); // 2005-03-18 - obiect nou

```

Listing 4.4: `LocalDate` este, la rândul său, imutabil

#### Notă

Multe clase de bază din biblioteca standard Java (`String`, `LocalDate`, `BigInteger`, `BigDecimal` — toate întâlnite deja în Capitolul 3) sunt **imutabile**, dintr-un motiv comun: obiectele imutabile sunt mai ușor de raționat, sigure implicit în context concurent (Capitolul 13–Capitolul 14) și pot fi partajate liber, fără riscul ca o parte din cod să modifice pe ascuns starea văzută de altă parte. Nu toate clasele proprii pe care le vom scrie vor fi imutabile, dar principiul merită ținut minte ca opțiune de design implicită, nu ca excepție.

## 4.3 Definirea claselor proprii

Revenim la exemplul `ContBancar` schițat în Secțiunea 4.1 (Figura 4.1) și îl transformăm, pas cu pas, într-o clasă Java completă.

### 4.3.1 Anatomia unei clase

O clasă Java tipică conține trei categorii de membri: **câmpuri** (starea internă), **constructori** (modul de creare a obiectelor) și **metode** (comporta-

**Listing 4.5:** Structura de bază a clasei ContBancar

mentul).

```

1 public class ContBancar {
2
3     // Campuri - starea internă a obiectului
4     private String titular;
5     private double sold;
6
7     // Constructor - modul de creare a unui obiect
8     public ContBancar(String titular, double soldInitial) {
9         this.titular = titular;
10        this.sold = soldInitial;
11    }
12
13    // Metode - comportamentul obiectului
14    public void depune(double suma) {
15        sold = sold + suma;
16    }
17
18    public boolean retrage(double suma) {
19        if (suma > sold) {
20            return false; // fonduri insuficiente
21        }
22        sold = sold - suma;
23        return true;
24    }
25
26    public double getSold() {
27        return sold;
28    }
29 }

```

**Notă**

Câmpurile sunt declarate private — o alegere deliberată, discutată pe larg în Secțiunea 4.4, care împiedică accesul direct din afara clasei. Singurul mod de a citi sau modifica soldul este prin metodele publice ale clasei, nu prin acces direct la câmp.

**4.3.2 Constructori; var; referințe null**

Un *constructor* este o metodă specială, cu același nume ca și clasa, apelată automat la crearea unui obiect cu `new`. Rolul său este să aducă noul obiect într-o stare inițială validă.

**Listing 4.6:** Crearea de obiecte folosind constructorul

```

1 ContBancar cont = new ContBancar("Ana Popescu", 500.0);
2 cont.depone(150.0);
3 boolean succes = cont.retrage(2000.0); // false - fonduri
4                                     // insuficiente

```

Pentru variabile locale (ca `cont` de mai sus), Java permite folosirea lui `var` în locul tipului explicit, atunci când tipul poate fi dedus din partea dreaptă a atribuirii:

**Listing 4.7:** `var` pentru inferența tipului la variabile locale

```

1 var cont = new ContBancar("Ana Popescu", 500.0);
2 // tipul e dedus: ContBancar
3 var suma = 150.0;
4 // tipul e dedus: double

```

**Notă**

`var` **nu** face din Java un limbaj cu tipare dinamică — tipul variabilei este stabilit definitiv la compilare, exact ca înainte; doar scrierea lui explicită devine opțională. `var` nu poate fi folosit pentru câmpuri, parametri de metodă sau tipuri de retur — doar pentru variabile locale, inițializate pe aceeași linie.

O variabilă de tip obiect care nu indică niciun obiect are valoarea specială `null`:

```
1 ContBancar cont = null;
2 System.out.println(cont.getSold());
3 // arunca NullPointerException la executie!
```

Listing 4.8: Referința `null` și pericolul ei**Notă**

Accesarea unui câmp sau apelarea unei metode pe o referință `null` este una dintre cele mai frecvente erori de execuție din Java, atât pentru începători, cât și în cod de producție. Compilatorul **nu** poate detecta, în general, această situație — ea apare doar la execuție, sub forma unei excepții `NullPointerException`. Vom învăța să gestionăm corect astfel de situații în Capitolul 9.

### 4.3.3 Parametri implicați și expliți

În constructorul `ContBancar` de mai sus, cuvântul cheie `this` este esențial: parametrul titular al constructorului și câmpul titular al clasei au **același nume**, iar `this.titular` dezambiguizează explicit că ne referim la câmp, nu la parametru.

```
1 public ContBancar(String titular, double sold) {
2     this.titular = titular;
3     // Observatie:
4     // this.titular -> campul
5     // titular (fara this) -> parametrul
6     this.sold = sold;
7 }
```

Listing 4.9: `this` distinge câmpul de parametrul omonim

Distincția dintre *parametrul implicit* și *parametrii expliți* ai unei metode este următoarea: în interiorul unei metode ne-stactice, `this` se referă la obiectul pe care a fost invocată metoda — acesta este *parametrul implicit*. Ceilalți parametri, listați explicit între paranteze la declararea metodei, sunt *parametrii expliți*.

```
1 // 'suma' e parametru EXPLICIT
2 // 'this' e parametrul IMPLICIT
3 public boolean retrage(double suma) {
4     if (suma > this.sold) {
5         return false;
6     }
7     this.sold = this.sold - suma;
8     return true;
9 }
```

Listing 4.10: Parametrul implicit (`this`) versus parametrii expliți

**Notă**

Scrierea explicită a lui `this` este opțională atunci când nu există ambiguitate de nume (Java o adaugă implicit) — de aceea, în codul scris până acum în acest curs, majoritatea accesărilor de câmpuri nu au folosit `this`. Recomandăm totuși folosirea lui `this` explicit în constructori, exact pentru a rezolva ambiguitatea nume-parametru/nume-câmp ilustrată mai sus.

#### 4.3.4 Suprascrierea constructorilor (overloading)

O clasă poate avea **mai mulți constructori**, cu liste diferite de parametri — fenomen numit *overloading* (supraîncărcare):

**Listing 4.11:** Doi constructori pentru `ContBancar`

```

1 public class ContBancar {
2     private String titular;
3     private double sold;
4
5     public ContBancar(String titular, double soldInitial) {
6         this.titular = titular;
7         this.sold = soldInitial;
8     }
9
10    // Constructor supraîncărcat - sold initial implicit 0
11    public ContBancar(String titular) {
12        this(titular, 0.0);
13        // apelează celălalt constructor al clasei
14    }
15 }
```

**Notă**

Apelul `this(titular, 0.0);` din al doilea constructor invocă **celălalt constructor al aceleiași clase** — o tehnică ce evită duplicarea codului de inițializare. Dacă este folosit, apelul `this(...)` trebuie să fie **prima instrucțiune** din corpul constructorului.

Compilatorul selectează constructorul potrivit pe baza numărului și tipului parametrilor din apel:

**Listing 4.12:** Compilatorul alege constructorul corect, pe baza parametrilor

```

1 ContBancar cont1 = new ContBancar("Ana Popescu", 500.0);
2     // primul constructor
3 ContBancar cont2 = new ContBancar("Ion Ionescu");
4     // al doilea constructor, sold = 0.0
```

#### 4.3.5 Blocuri de inițializare

Pe lângă constructori, Java oferă *blocuri de inițializare* — cod executat automat la construcția fiecărui obiect, util mai ales când mai mulți constructori au nevoie de aceeași logică de inițializare:

**Listing 4.13:** Un bloc de inițializare, executat la fiecare construcție de obiect

```

1 public class ContBancar {
2     private String titular;
3     private double sold;
4     private final String idCont;
5
6     // Bloc de inițializare - executat înaintea oricărui
7     // constructor
```

```

8      {
9          idCont = "CONT-" + System.nanoTime();
10     }
11
12     public ContBancar(String titular, double soldInitial) {
13         this.titular = titular;
14         this.sold = soldInitial;
15     }
16 }

```

#### Notă

În practică, blocurile de inițializare sunt rar folosite — majoritatea logicii de inițializare încapă natural într-un constructor. Le-am inclus aici mai ales pentru a introduce, prin contrast, inițializarea **statică**, mult mai des întâlnită și pe care o vom discuta în Subsecțiunea 4.4.4.

## 4.4 Încapsulare

### Definiție — Încapsulare

*Încapsularea* este principiul conform căruia starea internă a unui obiect este ascunsă de restul programului și poate fi accesată sau modificată **doar** prin metodele publice pe care clasa le expune explicit. Scopul nu este secretomania de dragul ei, ci protejarea obiectului împotriva unor stări invalide, imposibil de prevenit dacă orice cod extern ar putea modifica direct orice câmp.

### 4.4.1 Modificatori de acces

Java oferă patru niveluri de vizibilitate pentru câmpuri, metode și clase:

| Modificator                          | Vizibil din               | Observație              |
|--------------------------------------|---------------------------|-------------------------|
| <code>private</code>                 | doar aceeași clasă        | cel mai restrictiv      |
| <i>(implicit, fără cuvânt cheie)</i> | aceleși pachet            | „package-private”       |
| <code>protected</code>               | aceleși pachet + subclase | detaliat în Capitolul 6 |
| <code>public</code>                  | de oriunde                | cel mai permisiv        |

**Tabela 4.1:** Nivelurile de vizibilitate din Java, de la cel mai restrictiv la cel mai permisiv.

#### Notă

Regula practică de proiectare, pe care o vom aplica sistematic pe tot parcursul cursului: câmpurile unei clase sunt aproape întotdeauna `private`; metodele care formează „interfața publică” a clasei (ceea ce restul programului are voie să folosească) sunt `public`; orice metodă auxiliară, folosită doar intern, ar trebui declarată `private`, nu `public` „de siguranță”. Un modificator mai permisiv decât este necesar reduce libertatea de a schimba ulterior implementarea internă a clasei, fără să afecteze codul care o folosește.

### 4.4.2 Metode private

O metodă `private` este vizibilă doar în interiorul clasei care o declară — utilă pentru a extrage o bucată de logică internă, reutilizată de mai multe metode publice, fără a o expune ca parte din interfața publică a clasei:

**Listing 4.14:** O metodă privată, folosită intern de o metodă publică

```

1 public class ContBancar {
2     private String titular;
3     private double sold;
4
5     public boolean retrage(double suma) {
6         if (!sumaEsteValida(suma)) {
7             return false;
8         }
9         sold = sold - suma;
10        return true;
11    }
12
13    // Metoda privata - detaliu de implementare, nu parte
14    // din interfata publica
15    private boolean sumaEsteValida(double suma) {
16        return suma > 0 && suma <= sold;
17    }
18 }
```

#### Notă

Faptul că `sumaEsteValida` este `private` înseamnă că putem oricând să-i schimbăm numele, parametrii sau implementarea, fără să afectăm niciun cod din afara clasei `ContBancar` — nimeni din exterior nu are cum să apeleze direct această metodă. Această libertate de a modifica detaliile interne fără a rupe codul care folosește clasa este beneficiul central al încapsulării.

### 4.4.3 Câmpuri final

Un câmp declarat `final` poate fi atribuit o singură dată — fie direct la declarare, fie în constructor — și rămâne apoi neschimbat pe toată durata de viață a obiectului:

**Listing 4.15:** Câmpuri `final`, atribuite o singură dată, în constructor

```

1 public class ContBancar {
2     private final String titular;
3     // final - nu se schimba dupa construire
4     private double sold;
5     // NU final - soldul se modifica prin
6     // depuneri/retrageri
7
8     public ContBancar(String titular, double soldInitial) {
9         this.titular = titular;
10        // atribuire valida - e in constructor
11        this.sold = soldInitial;
12    }
13
14    public void schimbaTitular(String nouTitular) {
15        this.titular = nouTitular;
16        // EROARE DE COMPILARE - titular e final
17    }
18 }
```

**Notă**

`final` pe un câmp de tip obiect (de exemplu un `String` sau un `array`) garantează doar că **referința** nu se schimbă — nu că obiectul indicat de ea este imutabil. Un câmp `final` de tip `array`, de exemplu, tot permite modificarea elementelor sale individuale; doar reatribuirea întregului `array` eșuează la compilare. Combinați `final` cu o clasă efectiv imutabilă (ca `String`) atunci când aveți nevoie de garanții complete de neschimbare.

#### 4.4.4 Câmpuri statice și constante

```

1 public class ContBancar {
2     private static int numarConturiCreate = 0;
3     // partajat de TOATE instantele
4     private static final double DOBANDA_ANUALA = 0.02;
5     // constanta, la nivel de clasa
6
7     private String titular;
8     private double sold;
9
10    public ContBancar(String titular, double soldInitial) {
11        this.titular = titular;
12        this.sold = soldInitial;
13        numarConturiCreate++;
14        // modifica campul static, comun tuturor
15        // obiectelor
16    }
17
18    public static int getNumarConturiCreate() {
19        return numarConturiCreate;
20    }
21 }

```

```

1 ContBancar cont1 = new ContBancar("Ana Popescu", 500.0);
2 ContBancar cont2 = new ContBancar("Ion Ionescu", 1200.0);
3
4 System.out.println(ContBancar.getNumarConturiCreate()); // 2

```

**Listing 4.16:** Un câmp static, partajat de toate instanțele clasei

**Listing 4.17:** Toate obiectele văd aceeași valoare a câmpului static

**Notă**

Spre deosebire de un câmp obișnuit, un câmp static **nu** există separat pentru fiecare obiect — există o singură copie, asociată clasei, indiferent câte obiecte au fost create. Convenția `static final` pentru o constantă (`DOBANDA_ANUALA` de mai sus) este modul standard de a defini valori care nu se schimbă și nu au sens legate de un obiect anume.

#### 4.4.5 Metode statice; metode factory

O metodă statică este apelată pe clasă, nu pe un obiect anume — de altfel, am folosit deja metode statice fără să le numim explicit astfel: `LocalDate.of(...)` și `LocalDate.now()` din Subsecțiunea 4.2.2 sunt exemple de *metode factory* statice.

```

1 public class ContBancar {
2     private String titular;
3     private double sold;

```

**Listing 4.18:** O metodă factory statică pentru `ContBancar`

```

4
5     private ContBancar(String titular, double soldInitial) {
6         // constructor PRIVAT
7         this.titular = titular;
8         this.sold = soldInitial;
9     }
10
11     // Metoda factory - alternativa publica, cu nume
12     // descriptiv, la new ContBancar(...)
13     public static ContBancar contNou(String titular) {
14         return new ContBancar(titular, 0.0);
15     }
16
17     public static ContBancar contCuDepunereInitiala(String
18     titular, double sumaInitiala) {
19         if (sumaInitiala < 0) {
20             throw new IllegalArgumentException("Suma initiala
21             nu poate fi negativa");
22         }
23         return new ContBancar(titular, sumaInitiala);
24     }
25 }

```

1 ContBancar cont1 = ContBancar.contNou("Ana Popescu");  
2 ContBancar cont2 = ContBancar.contCuDepunereInitiala("Ion  
Ionescu", 1200.0);

**Listing 4.19:** Folosirea metodelor factory, în locul lui new direct

#### Notă

Metodele factory devin utile mai ales atunci când constructorul simplu nu este suficient de expresiv: nume ca contNou sau contCuDepunereInitiala comunică intenția mult mai clar decât ar putea-o face un constructor supraîncărcat (vezi Subsecțiunea 4.3.4), mai ales atunci când există mai mulți constructori cu aceeași listă de tipuri de parametri, situație în care overloading-ul simplu nu ar fi posibil din cauza ambiguității. Observați și că am declarat constructorul `private` — singurul mod de a crea un obiect `ContBancar` din afara clasei este acum prin metodele factory publice.

### 4.4.6 Metoda main

Revenim, cu o înțelegere mai profundă, la metoda cu care a început acest curs (vezi Secțiunea 2.1):

**Listing 4.20:** Semnătura metodei main, acum pe deplin explicabilă

```

1 public static void main(String[] args) {
2     // ...
3 }

```

Fiecare cuvânt din această semnătură are acum sens: `public` astfel încât JVM-ul (din afara clasei) să o poată apela; `static` pentru că JVM-ul o invocă fără să existe încă vreun obiect construit — nu poate exista un `this` implicit înainte ca vreo instanță a clasei să fi fost creată; `void` pentru că metoda nu returnează nimic apelantului (JVM-ul); `String[] args` pentru argumentele primite din linia de comandă la lansarea programului.

#### Notă

Deoarece `main` este `static`, corpul ei **nu** poate accesa direct câmpuri sau metode ne-stactice ale clasei — exact ca orice altă metodă statică.

Dacă un program are nevoie de comportament ne-static, main trebuie mai întâi să construiască un obiect (`new NumeClasa(...)`) și să apeleze metode pe acel obiect, așa cum am făcut deja în exemplele cu `ContBancar` din acest capitol.

## 4.5 Records

Multe clase există doar pentru a transporta câteva date înrudite, fără comportament complex — o pereche de coordonate, un rezultat de calcul, o tranzacție. Pentru acest tipar frecvent, scrierea manuală a constructorului, a metodelor accesori, și a metodelor `equals`/`hashCode`/`toString` (pe care le vom detalia riguros în Capitolul 6) devine repetitivă. Un record generează automat tot acest cod repetitiv, pornind doar de la lista câmpurilor.

### 4.5.1 Conceptul de record

#### Definiție — Record

Un *record* este o formă specială, concisă, de clasă imutabilă, destinată să transporte date. Declararea câmpurilor (numite, într-un record, *componente*) generează automat un constructor, metode accesori (fără prefixul `get`), și implementări corecte pentru `equals`, `hashCode` și `toString`.

```
1 public record Tranzactie(String tip, double suma, LocalDate
   data) {
2 }

1 Tranzactie t = new Tranzactie("depunere", 150.0, LocalDate.
   now());
2
3 System.out.println(t.tip()); // "depunere" - accesori
4                               // generat, FARA prefixul get
5 System.out.println(t.suma()); // 150.0
6 System.out.println(t);        // Tranzactie[tip=depunere,
7                               // suma=150.0, data=2026-...]
```

Listing 4.21: Un record pentru o tranzacție bancară

Listing 4.22: Folosirea unui record: constructor și accesorii generate automat

#### Notă

Comparativ cu o clasă obișnuită de dimensiune similară (ca `ContBancar` din Subsecțiunea 4.3.1), un record de o singură linie înlocuiește zeci de linii de cod repetitiv: constructor, accesorii, `equals`, `hashCode`, `toString`. Prețul acestei concizii este imutabilitatea impusă — un record **nu** poate avea câmpuri mutabile sau metode `set...`; dacă aveți nevoie de un obiect mutabil (ca `ContBancar`, al cărui sold se schimbă), un record nu este potrivit, iar o clasă obișnuită rămâne alegerea corectă.

### 4.5.2 Constructori canonici, compacti și personalizați

Constructorul generat automat, care atribuie fiecare componentă exact în ordinea declarată, se numește *constructor canonic*. Acesta poate fi **rescris explicit** atunci când e nevoie de validare la construcție:

Listing 4.23: Un constructor compact, pentru validare

```

1 public record Tranzactie(String tip, double suma, LocalDate
   data) {
2
3     // Constructor COMPACT - fara lista de parametri, fara
4     // atribuirii explicite
5     public Tranzactie {
6         if (suma <= 0) {
7             throw new IllegalArgumentException("Suma
   tranzactiei trebuie sa fie pozitiva");
8         }
9         tip = tip.toLowerCase();
10        // se poate normaliza o componenta inainte de
11        // atribuire
12    }
13 }

```

#### Notă

Constructorul *compact* nu repetă lista de parametri (aceasta rămâne cea din declarația record-ului) și nu conține atribuirii explicite `this.suma = suma;` — acestea se execută automat, după corpul constructorului compact. Este locul potrivit pentru validare și normalizare a datelor primite, nu pentru logică arbitrară.

Pe lângă constructorul canonic (eventual rescris compact), un record poate avea și constructori suplimentari, la fel ca o clasă obișnuită (vezi Subsecțiunea 4.3.4) — cu condiția ca fiecare dintre ei să delege, în cele din urmă, către constructorul canonic:

Listing 4.24: Un constructor suplimentar, care delegă către cel canonic

```

1 public record Tranzactie(String tip, double suma, LocalDate
   data) {
2
3     public Tranzactie {
4         if (suma <= 0) {
5             throw new IllegalArgumentException("Suma
   tranzactiei trebuie sa fie pozitiva");
6         }
7     }
8
9     // Constructor suplimentar - data implicita este
10    // data curenta
11    public Tranzactie(String tip, double suma) {
12        this(tip, suma, LocalDate.now());
13        // delegare obligatorie catre constructorul canonic
14    }
15 }

```

#### Rezumat

- O clasă are trei categorii de membri: câmpuri (starea internă), constructori (crearea obiectelor) și metode (comportamentul).
- `this` este parametrul implicit al oricărei metode ne-stactice; `this(...)` din interiorul unui constructor apelează un alt constructor al aceleiași clase (overloading).
- Încapsularea ascunde starea internă (câmpuri de regulă private) în spatele unei interfețe publice de metode — protejează obiectul de stări invalide și permite schimbarea liberă a implementării interne.

- ▶ Un câmp/metodă static aparține clasei, nu unei instanțe anume; metodele `factory` statice oferă o alternativă mai expresivă la constructori pentru crearea de obiecte.
- ▶ Metoda `main` este static tocmai pentru că JVM-ul o invocă înainte ca vreun obiect al clasei să existe.
- ▶ Un record este o clasă imutabilă concisă pentru transportul de date, cu constructor/accesori/`equals`/`hashCode`/`toString` generate automat; constructorul compact permite validare fără a repeta lista de parametri.

### Exerciții

1. Adăugați la clasa `ContBancar` un câmp static `numarConturi`. Create și o metodă statică ce îl returnează. Verificați, printr-un mic program, că valoarea crește corect la fiecare cont nou creat.
2. Transformați constructorul public al clasei `ContBancar` într-unul privat și oferiți în locul lui două metode `factory`: `contNou(titular)` (sold inițial 0) și `contCuDepunereInitia-la(titular, suma)` (cu validare: suma nu poate fi negativă).
3. Definiți un record `Punct(double x, double y)` și adăugați-i o metodă (nu o componentă) `distanțaLaOrigine()`, care calculează distanța euclidiană până la origine. (Indicație: un record poate avea și metode obișnuite, pe lângă componente.)
4. Scrieți un constructor compact pentru recordul `Tranzactie` care validează, în plus față de exemplul din curs, că `tip` este una dintre valorile "depunere" sau "retragere", altfel aruncă o excepție.
5. Explicați, în cuvinte proprii, de ce `ContBancar` (cu soldul care se modifică prin depuneri/retrageri) nu ar putea fi rescris ca record, în timp ce `Tranzactie` (care reprezintă un eveniment petrecut deja, neschimbător) este un candidat potrivit.



# Obiecte și clase (II): pachete, JAR, Javadoc

# 5

## Introducere

Am construit deja clase complete în Capitolul 4, dar toate au trăit izolat, într-un singur fișier. Pe măsură ce un proiect crește dincolo de câteva clase, avem nevoie de un mod de a le organiza (*pachete*), de a le distribui ca o singură unitate (*fișiere JAR*) și de a le documenta pentru alți programatori care le vor folosi, fără să le citească sursa (*Javadoc*). Capitolul se încheie cu câteva recomandări de design, adunate din experiența practică a comunității Java.

## 5.1 Pachete

### Definiție — Pachet

Un *pachet* (*package*) este un spațiu de nume (*namespace*) care grupează clase înrudite și previne coliziunile de nume între biblioteci diferite. Două clase cu același nume simplu (de exemplu `List`) pot co-exista fără ambiguitate, atât timp cât aparțin unor pachete diferite (`java.util.List` versus `java.awt.List`).

### 5.1.1 Denumirea și organizarea pachetelor

Convenția Java pentru numele de pachete este domeniul de internet al organizației, scris invers, urmat de numele proiectului:

```
1 package ro.unibv.cursjava.bancar;  
2  
3 public class ContBancar {  
4     // ...  
5 }
```

**Listing 5.1:** Declararea pachetului, ca primă linie a fișierului sursă

### Notă

Declarația `package` trebuie să fie **prima** instrucțiune din fișierul sursă (înaintea oricărui `import` sau a clasei propriu-zise). Numele complet al clasei `ContBancar` de mai sus devine, de fapt, `ro.unibv.cursjava.bancar.ContBancar` — numele „simplu” pe care l-am folosit până acum era doar o prescurtare convenabilă.

Structura de directoare pe disc **trebuie** să reflecte exact structura pachetului — aceasta nu este o convenție opțională, ci o cerință impusă de compilator:

```
1 src/  
2   ro/  
3     unibv/  
4       cursjava/  
5         bancar/  
6           ContBancar.java  
7           Tranzactie.java
```

**Listing 5.2:** Structura de directoare corespunzătoare pachetului `ro.unibv.cursjava.bancar`

### 5.1.2 Importuri de clase; `import static`

Pentru a folosi o clasă dintr-un alt pachet, avem nevoie fie de numele ei complet, fie de o declarație `import`, care permite folosirea numelui simplu:

**Listing 5.3:** Import de o singură clasă versus import de tot pachetul

```
1 | import java.time.LocalDate;           // import specific, o
   | singura clasa
2 | import java.util.*;                  // import cu wildcard -
   | TOATE clasele din java.util
3 |
4 | LocalDate azi = LocalDate.now(); // fara acest import, ar
   | trebui java.time.LocalDate.now()
```

#### Notă

Importul cu wildcard (`import java.util.*;`) nu are niciun impact asupra performanței la execuție și nici nu importă subpachete (`java.util.concurrent` rămâne neimportat) — afectează doar lizibilitatea codului sursă. Multe echipe și IDE-uri preferă totuși importuri explicite, câte unul de fiecare clasă, tocmai pentru a face vizibilă, dintr-o privire, proveniența fiecărui nume folosit în fișier.

O formă specială, `import static`, permite folosirea directă a câmpurilor și metodelor statice ale unei clase, fără a mai prefixa numele clasei:

**Listing 5.4:** `import static`, pentru acces direct la membri statici

```
1 | import static java.lang.Math.sqrt;
2 | import static java.lang.Math.PI;
3 |
4 | double diagonala = sqrt(2) * 5; // fara aceasta, ar trebui
   | Math.sqrt(2)
5 | double arie = PI * 5 * 5;       // fara aceasta, ar trebui
   | Math.PI
```

#### Notă

`import static` este folosit cu moderație în practică — util mai ales pentru constante matematice (`Math.PI`) sau pentru biblioteci de testare, unde metode precum `assertEquals` sunt apelate atât de des încât prefixul repetat ar îngreuna lectura. Folosit excesiv, poate face neclar de unde provine o metodă sau o constantă folosită într-un fișier.

### 5.1.3 Class path; compilare cu pachete

Atunci când clasele sunt organizate în pachete, compilarea și execuția din linia de comandă necesită puțin mai multă grijă decât în Secțiunea 2.1:

**Listing 5.5:** Compilare și execuție pentru clase organizate în pachete

```
1 | $ javac -d bin src/ro/unibv/cursjava/bancar/ContBancar.java
2 | $ java -cp bin ro.unibv.cursjava.bancar.ContBancar
```

Opțiunea `-d bin` indică unde să fie plasate fișierele `.class` compilate (respectând structura de pachete); opțiunea `-cp` (*class path*) indică JVM-ului unde să caute clasele compilate la execuție. Numele complet, cu pachet, este obligatoriu la execuție — `java ContBancar` (fără prefix) ar eșua cu o eroare de tipul `ClassNotFoundException`.

**Notă**

În practică, la acest curs veți folosi aproape întotdeauna un IDE (vezi Secțiunea 1.4) sau un instrument de build (Maven/Gradle), care gestionează automat structura de directoare, compilarea și class path-ul. Este totuși important să înțelegeți ce se întâmplă în spate, mai ales atunci când apar erori de tipul `ClassNotFoundException` sau `NoClassDefFoundError` — ambele semnaleză, de regulă, o problemă de class path, nu o eroare de logică a programului.

### 5.1.4 Acces la nivel de pachet

Revenim la tabelul modificatorilor de acces din Subsecțiunea 4.4.1: nivelul implicit (fără niciun cuvânt cheie) este vizibil **doar** în cadrul aceleiași pachet — de aici numele informal „package-private”.

```

1 package ro.unibv.cursjava.bancar;
2
3 // fara 'public' - vizibila doar in acest pachet
4 class UtilitarValidare {
5
6     // fara modificador - package-private
7     static boolean sumaEsteValida(double suma) {
8         return suma > 0;
9     }
10 }
```

**Listing 5.6:** O clasă și o metodă vizibile doar în cadrul pachetului

**Notă**

Vizibilitatea la nivel de pachet este utilă pentru clase și metode „de sprijin”, folosite de mai multe clase din același pachet, dar care nu ar trebui expuse codului din afara acelui pachet — un compromis între `private` (prea restrictiv, dacă mai multe clase din pachet au nevoie de acces) și `public` (prea permisiv, dacă utilizarea e strict internă pachetului).

## 5.2 Fișiere JAR

**Definiție — JAR**

Un fișier *JAR* (*Java ARchive*) este o arhivă, în format ZIP, care grupează într-un singur fișier clase compilate (`.class`), resurse (imagini, fișiere de configurare) și metadata despre conținutul arhivei. Este formatul standard de distribuire a bibliotecilor și aplicațiilor Java.

### 5.2.1 Crearea unui JAR

```
$ jar cf biblioteca.jar -C bin .
```

**Listing 5.7:** Crearea unui fișier JAR din clase deja compilate

Opțiunile folosite mai sus: `c` (*create*, creează o arhivă nouă), `f` (*file*, numele fișierului rezultat urmează imediat), `-C bin .` (schimbă directorul curent la `bin` și arhivează tot conținutul de acolo, păstrând structura de pachete).

## 5.2.2 Manifestul și JAR-uri executabile

Fiecare JAR conține un fișier special, META-INF/MANIFEST.MF, cu metadate despre arhivă. Pentru un JAR **executabil** (rulabil direct cu `java -jar`), manifestul trebuie să indice clasa cu metoda `main`:

**Listing 5.8:** Conținutul unui fișier manifest minimal

**Listing 5.9:** Crearea unui JAR executabil, cu manifest generat automat

```
1 Main-Class: ro.unibv.cursjava.bancar.ContBancar
```

```
1 $ jar cfe aplicatie.jar ro.unibv.cursjava.bancar.ContBancar -  
  C bin .  
2 $ java -jar aplicatie.jar
```

Opțiunea suplimentară `e` (*entry point*) generează automat manifestul potrivit, fără a mai fi nevoie să scriem manual fișierul MANIFEST.MF.

### Notă

Un JAR executabil rulat cu `java -jar aplicatie.jar` **ignoră** orice opțiune `-cp` dată în linia de comandă — singura sursă de `class path` suplimentar, în acest caz, este atributul `Class-Path` din manifest. Aceasta e o sursă frecventă de confuzie pentru cineva obișnuit deja cu `-cp` din Subsecțiunea 5.1.3.

## 5.2.3 JAR-uri multi-release

Bibliotecile care doresc să profite de funcționalități noi ale platformei, păstrând totodată compatibilitate cu versiuni mai vechi de Java, pot folosi un *JAR multi-release* — o singură arhivă ce conține variante alternative ale unor clase, selectate automat de JVM în funcție de versiunea de Java folosită la execuție:

**Listing 5.10:** Structura unui JAR multi-release (schematic)

```
1 biblioteca.jar  
2 META-INF/  
3   MANIFEST.MF (contine: Multi-Release: true)  
4   versions/  
5     17/  
6       ro/exemplu/Util.class (varianta pentru  
7                               Java 17+)  
8     21/  
9       ro/exemplu/Util.class (varianta pentru  
10                              Java 21+, foloseste  
11                              virtual threads)  
12 ro/  
13   exemplu/  
14     Util.class (varianta implicita,  
15                  compatibila cu orice versiune)
```

### Notă

JAR-urile multi-release sunt o funcționalitate destinată în principal autorilor de biblioteci, nu codului aplicație obișnuit — le menționăm aici pentru exhaustivitate, dar nu le vom folosi activ în acest curs. Este suficient să recunoașteți conceptul dacă îl întâlniți în structura unei dependențe externe.

## 5.3 Comentarii de documentație (Javadoc)

Am menționat deja, în Subsecțiunea 2.1.1, existența comentariilor de forma `/** ... */`. Această secțiune arată de ce sunt speciale: instrumentul javadoc, inclus în JDK, le extrage automat și generează din ele documentație HTML navigabilă — același format familiar din documentația oficială a bibliotecii standard Java.

### 5.3.1 Sintaxa comentariilor Javadoc

```

1  /**
2  * Reprezinta un cont bancar simplu, cu operatii de depunere
   * si retragere.
3  *
4  * @author Ana Popescu
5  * @since 1.0
6  */
7  public class ContBancar {
8
9      /**
10     * Depune o suma in cont.
11     *
12     * @param suma suma de depus; trebuie sa fie strict
       pozitiva
13     * @throws IllegalArgumentException daca suma nu este
       pozitiva
14     */
15     public void depune(double suma) {
16         if (suma <= 0) {
17             throw new IllegalArgumentException("Suma trebuie
       sa fie pozitiva");
18         }
19         sold = sold + suma;
20     }
21 }

```

**Listing 5.11:** Un comentariu Javadoc complet, pentru o clasă și o metodă

#### Notă

Un comentariu Javadoc trebuie plasat **imediat înaintea** elementului pe care îl documentează (clasă, metodă, câmp sau constructor), fără nicio altă linie între ele — altfel instrumentul javadoc nu îl asociază corect cu elementul respectiv. Prima propoziție a comentariului joacă un rol special: devine rezumatul afișat în tabelele de sumar ale documentației generate, deci merită scrisă cu grijă, ca o frază completă și autonomă.

### 5.3.2 Etichete (tag-uri) comune

Javadoc recunoaște un set de *etichete* speciale, fiecare începând cu @, care structurează informația documentată:

Pe lângă etichetele de bloc din Tabela 5.1, Javadoc recunoaște și etichete *inline*, folosite direct în text, delimitate de acolade:

```

1  /**
2  * Retrage o suma din cont, daca fondurile sunt suficiente.
3  *
4  * <p>Aceasta metoda este complementara metodei {@link #depune
   (double)}.

```

**Listing 5.12:** Etichete Javadoc inline: `{@code ...}` și `{@link ...}`

**Tabela 5.1:** Etichetele Javadoc cel mai frecvent folosite.

| Etichetă              | Semnificație                                               |
|-----------------------|------------------------------------------------------------|
| @param nume descriere | descrie un parametru al metodei/constructorului            |
| @return descriere     | descrie valoarea returnată (absentă la metode void)        |
| @throws Tip descriere | descrie o excepție pe care metoda o poate arunca           |
| @see referinta        | trimite către un element înrudit (altă clasă/metodă)       |
| @author nume          | autorul clasei                                             |
| @since versiune       | versiunea la care elementul a fost introdus                |
| @deprecated motiv     | marchează elementul ca depășit, cu alternativa recomandată |

```

5  * Apelul {@code retrage(0)} este permis si nu modifica soldul
6  *
7  * @param suma suma de retras
8  * @return {@code true} daca retragerea a reusit, {@code false}
9  *   altfel
10 * /
11 public boolean retrage(double suma) {
12     // ...
13     return true;
14 }

```

**Notă**

`{@code text}` afișează textul cu font monospace, fără a-l interpreta ca HTML — util pentru fragmente scurte de cod menționate în proza documentației. `{@link element}` generează un hyperlink către elementul referit (o altă metodă, clasă sau câmp) în documentația HTML rezultată. Observați și că, în corpul comentariului, se pot folosi etichete HTML simple (precum `<p>` de mai sus) — documentația generată este, la urma urmei, o pagină HTML.

**5.3.3 Generarea documentației****Listing 5.13:** Generarea documentației HTML din linia de comandă

```

1 javac -d bin src/ro/unitbv/mip/banking/ContBancar.java
2 javadoc -d docs -author -version src/ro/unitbv/mip/banking/*.java

```

Opțiunea `-d docs` indică directorul unde se generează paginile HTML; `-author` și `-version` instruiesc javadoc să includă în documentație informațiile din etichetele `@author` și, respectiv, dintr-o eventuală etichetă de versiune. Rezultatul este un set de pagini HTML interconectate, identice ca structură cu documentația oficială a bibliotecii standard Java — de altfel, chiar aceasta este generată cu același instrument.

**Notă**

IDE-urile moderne (Secțiunea 1.4) afișează automat conținutul comentariilor Javadoc sub formă de *tooltip*, atunci când poziționați cursorul deasupra numelui unei clase sau metode — inclusiv pentru clase din biblioteca standard sau din biblioteci externe. Documentați-vă clase-

le și metodele publice consecvent: beneficiul nu este vizibil doar în paginile HTML generate, ci în fiecare moment în care voi sau colegii voștri folosiți acel cod din IDE.

## 5.4 Recomandări de design pentru clase

### 5.4.1 Principii generale

- **Păstrați datele private.** Orice câmp public poate fi modificat necontrolat din afara clasei, anulând avantajele încapsulării (Secțiunea 4.4). Excepție rezonabilă: câmpurile public static final (constante).
- **Inițializați întotdeauna datele.** Java oferă valori implicite pentru câmpuri neinițializate explicit (0, false, null), dar a te baza pe acestea, în loc să inițializezi explicit prin constructor, face codul mai greu de urmărit.
- **Nu folosiți prea multe tipuri primitive într-o clasă.** Un grup de câmpuri înrudite (de exemplu strada, oras, codPostal) ar trebui extras într-o clasă proprie (Adresa), în loc să rămână câmpuri separate ale clasei originale.
- **Nu toate câmpurile au nevoie de accesorii și mutatori individuali.** Expunerea necondiționată a unui accesor și a unui mutator pentru fiecare câmp privat este echivalentă, practic, cu a face acel câmp public — gândiți-vă la ce operații are sens să fie permise din exterior, nu la a expune mecanic fiecare câmp.
- **Împărțiți clasele și metodele care au responsabilități prea largi.** O clasă sau o metodă ar trebui să aibă o singură responsabilitate clară, ușor de descris într-o propoziție.
- **Alegeți nume de clase (substantive) și metode (verbe) care reflectă clar responsabilitatea lor,** urmând euristică din Subsecțiunea 4.1.2.

Aceste recomandări urmează îndeaproape principiile de design consacrate în literatura de referință a limbajului [1].

### 5.4.2 Greșeli frecvente de evitat

```

1 public class Echipa {
2     private List<String> membri;
3
4     public List<String> getMembri() {
5         return membri;
6         // PERICULOS - expune referinta interna directa!
7     }
8 }
```

Listing 5.14: O eroare frecventă: expunerea unei referințe interne mutabile

#### Notă

Codul de mai sus pare încapsulat (câmpul `membr`i este `private`), dar `getMembri()` returnează chiar referința internă, nu o copie — orice cod extern poate modifica direct lista, ocolind complet controlul clasei `Echipa`. Această capcană se numește *expunerea reprezentării interne* (*representation exposure*). Vom relua acest subiect riguros, alături de soluțiile corecte (vizualizări imutabile, copii defensive), în Capitolul

11, când vom discuta pe larg colecțiile.

#### Rezumat

- ▶ Un pachet organizează clase înrudite într-un spațiu de nume; structura de directoare trebuie să reflecte exact structura pachetului.
- ▶ `import` permite folosirea numelui simplu al unei clase din alt pachet; `import static` face același lucru pentru membri statici, dar trebuie folosit cu moderație.
- ▶ Nivelul de acces implicit (fără cuvânt cheie) este „package-private” — vizibil doar în cadrul aceluiași pachet, util pentru cod de sprijin ne-public.
- ▶ Un fișier JAR grupează clase compilate într-o singură arhivă; un manifest cu `Main-Class` face din arhivă un JAR executabil, rulabil cu `java -jar`.
- ▶ Comentariile Javadoc (`/** ... */`), plasate imediat înaintea elementului documentat, sunt extrase de instrumentul javadoc în documentație HTML navigabilă; etichetele `@param`, `@return`, `@throws` structurează informația, iar `{@code}`/`{@link}` sunt etichete inline.
- ▶ Proiectarea bună a claselor pornește de la încapsulare strictă, responsabilități unice, clare, și evitarea expunerii directe a stării mutabile interne.

#### Exerciții

1. Reorganizați clasa `ContBancar` din Capitolul 4 într-un pachet `ro.unitbv.mip.banking`, respectând structura de directoare corespunzătoare, apoi compilați și rulați-o din linia de comandă, fără IDE.
2. Creați un JAR executabil pentru un mic program cu metodă `main`, folosind opțiunea `e` a comenzii `jar`, și verificați-l cu `java -jar`.
3. Adăugați comentarii Javadoc complete (inclusiv `@param`, `@return`, `@throws` unde e cazul) pentru toate metodele publice ale clasei `ContBancar`, apoi generați documentația HTML cu `javadoc` și inspectați rezultatul într-un browser.
4. Identificați, într-un cod propriu scris la un laborator anterior, un exemplu de „expunere a reprezentării interne” (o metodă care returnează direct o referință mutabilă privată). Propuneți o corectare.
5. Discutați: de ce credeți că biblioteca standard Java preferă, pentru multe clase, mai multe metode accesori specifice (de exemplu `getYear()`, `getMonthValue()` la `LocalDate`, Subsecțiunea 4.2.2) în loc de un singur accesori care returnează un obiect intern complet?

### Introducere

De-a lungul capitolelor anterioare am construit clasa `ContBancar` pas cu pas, dar am tratat-o mereu ca pe o entitate izolată. În practică, multe clase dintr-un domeniu au variante specializate ale unui concept mai general: un cont de economii este, la urma urmei, tot un cont bancar, dar cu o regulă suplimentară (dobândă). *Moștenirea* este mecanismul Java prin care exprimăm exact această relație „este un” (*is-a*), menționată deja, în treacăt, în Subsecțiunea 4.1.2. Vom vedea și cum moștenirea deschide calea către *polimorfism* — capacitatea codului de a trata uniform obiecte de tipuri înrudite, dar cu comportamente diferite.

## 6.1 Superclase și subclase

### 6.1.1 Definirea unei subclase

Presupunem că vrem un cont de economii: identic cu `ContBancar` (titular, sold, depunere, retragere), dar cu o rată a dobânzii și o metodă suplimentară de aplicare a acesteia.

```
1 public class ContEconomii extends ContBancar {
2
3     private double rataDobanda;
4
5     public ContEconomii(
6         String titular,
7         double soldInitial,
8         double rataDobanda
9     ) {
10         // apeleaza constructorul din ContBancar
11         super(titular, soldInitial);
12         this.rataDobanda = rataDobanda;
13     }
14
15     public void aplicaDobanda() {
16         double dobanda = getSold() * rataDobanda;
17         depune(dobanda);
18     }
19 }
```

Listing 6.1: O subclasă, definită cu `extends`

### Notă

`ContEconomii` **moștenește** automat toți membrii publici și protejați ai lui `ContBancar` — câmpurile `titular` și `sold` (private, deci inaccesibile direct, dar manipulabile prin metodele moștenite `depune/retrage/getSold`) și metodele corespunzătoare. Nu trebuie să re-scriem nimic din codul deja scris în Capitolul 4 — exact acesta este beneficiul central al moștenirii: reutilizarea codului existent, combinată cu extinderea lui controlată.

### 6.1.2 Apelul `super(...)` și constructorii în subclase

Un constructor de subclasă trebuie, direct sau indirect, să inițializeze și partea moștenită de la superclasă. Apelul `super(...)` face exact acest lucru, invocând constructorul superclasei:

Listing 6.2: `super(...)` trebuie să fie prima instrucțiune a constructorului

```

1 public ContEconomii(
2     String titular,
3     double soldInitial,
4     double rataDobanda
5 ) {
6     // OBLIGATORIU prima instructiune, daca e folosita
7     super(titular, soldInitial);
8     this.rataDobanda = rataDobanda;
9 }

```

#### Notă

Dacă un constructor de subclasă **nu** conține explicit un apel `super(...)`, compilatorul inserează automat un apel implicit către constructorul **fără argumente** al superclasei. Dacă superclasa nu are un asemenea constructor (de exemplu, `ContBancar` are doar constructori cu parametri, din Subsecțiunea 4.3.4), compilarea eșuează — subclasa este atunci **obligată** să apeleze explicit unul dintre constructorii existenți ai superclasei, cu argumentele potrivite.

### 6.1.3 Suprascrierea metodelor

O subclasă poate *suprascrie* (*override*) o metodă moștenită, oferind o implementare proprie, cu aceeași semnătură:

Listing 6.3: Suprascrierea unei metode moștenite

```

1 public class ContEconomii extends ContBancar {
2     private double rataDobanda;
3     private static final double PENALIZARE_RETRAGERE = 0.01;
4
5     // ...
6
7     @Override
8     public boolean retrage(double suma) {
9         double sumaCuPenalizare = suma * (1 +
10             PENALIZARE_RETRAGERE);
11         return super.retrage(sumaCuPenalizare);
12         // apeleaza versiunea din ContBancar
13     }
14 }

```

#### Notă

Adnotarea `@Override` este **opțională** din perspectiva compilatorului, dar foarte recomandată: dacă semnătura metodei nu se potrivește exact cu vreo metodă din superclasă (de exemplu, ați greșit tipul unui parametru), compilatorul semnalează eroare la acest punct, în loc să creeze din greșeală o metodă nouă, complet separată, care doar *pare* să suprascrîe ceva. În interiorul metodei suprascrise, `super.retrage(...)` apelează explicit versiunea din superclasă — util atunci când noua implementare trebuie să *extindă* comportamentul original, nu să-l înlocuiască în întregime.

## 6.2 Polimorfism

### Definiție — Polimorfism

*Polimorfismul* este proprietatea prin care o variabilă de tipul unei superclase poate referi obiecte ale oricărei subclase, iar apelul unei metode suprascrise se rezolvă, la execuție, către implementarea specifică a tipului **real** al obiectului — nu a tipului declarat al variabilei.

### 6.2.1 Legarea dinamică (*dynamic dispatch*)

```
1 ContBancar cont = new ContEconomii("Ana Popescu", 1000.0,
2     0.02);
3 cont.retrage(200.0);
4 // executa ContEconomii.retrage, NU ContBancar.retrage!
```

**Listing 6.4:** O variabilă de tip superclasă poate referi un obiect de tip subclasă

### Notă

Deși variabila `cont` este declarată de tip `ContBancar`, obiectul la care aceasta se referă este, la execuție, un `ContEconomii`. Apelul `cont.retrage(200.0)` execută versiunea **suprascrisă** din `ContEconomii`, nu cea din `ContBancar` — acest mecanism, numit *legare dinamică* (*dynamic dispatch*), este determinat de tipul **real** al obiectului, stabilit la crearea lui cu `new`, nu de tipul declarat al variabilei care îl referă.

Polimorfismul devine util mai ales atunci când tratăm uniform o colecție de obiecte de tipuri înrudite, dar diferite:

```
1 ContBancar[] conturi = {
2     new ContBancar("Ion Ionescu", 500.0),
3     new ContEconomii("Ana Popescu", 1000.0, 0.02)
4 };
5
6 for (ContBancar c : conturi) {
7     c.retrage(100.0);
8     // fiecare obiect executa PROPRIA sa versiune de retrage
9 }
```

**Listing 6.5:** Tratare uniformă, prin polimorfism, a unor obiecte de tipuri diferite

### 6.2.2 Cast-uri între clase înrudite; `instanceof` și `pattern matching`

Uneori avem nevoie de funcționalitatea specifică unei subclase, deși lucrăm cu o variabilă declarată de tip superclasă. Un *cast* explicit face conversia posibilă, dar doar dacă obiectul **este cu adevărat** de tipul respectiv la execuție:

```
1 ContBancar cont = new ContEconomii("Ana Popescu", 1000.0,
2     0.02);
3 if (cont instanceof ContEconomii economii) {
4     // pattern matching: verifica SI converteste
5     economii.aplicaDobanda();
6     // acum putem apela metoda specifica ContEconomii
7 }
```

**Listing 6.6:** Cast explicit către o subclasă, cu verificare `instanceof`

**Notă**

Forma modernă a lui `instanceof` (folosită mai sus) combină verificarea de tip cu declararea unei variabile noi (economii), deja convertite la tipul verificat — variabila este disponibilă doar în ramura unde condiția e adevărată. Un cast „orb”, fără verificare `instanceof` în prealabil, care se dovedește greșit la execuție, aruncă o excepție `ClassCastException`; vom discuta gestionarea corectă a unor astfel de excepții în Capitolul 9. Vom relua pattern matching-ul, în forma sa completă și mai avansată, în Capitolul 8.

### 6.2.3 Modificatorul `protected`; clase și metode `final`

Revenim la ultimul nivel de acces din Tabela 4.1, rămas deocamdată neexplicat: `protected`. Un membru `protected` este vizibil în același pachet (ca și cel implicit, din Subsecțiunea 5.1.3) și, **în plus**, din orice subclasă, indiferent de pachetul acesteia:

**Listing 6.7:** Un câmp `protected`, accesibil direct din subclasă

```

1 public class ContBancar {
2     protected double sold;
3     // vizibil în subclase, chiar din alte pachete
4
5     // ...
6 }
7
8 public class ContEconomii extends ContBancar {
9     public void aplicaDobandaDirect(double rata) {
10         sold = sold * (1 + rata);
11         // acces DIRECT la câmpul protected moștenit
12     }
13 }
```

**Notă**

Folosirea lui `protected` pentru câmpuri este, în general, o alegere de design discutabilă — expune starea internă a superclasei subclaselor sale, exact tipul de dependență fragilă pe care încapsularea (Secțiunea 4.4) urmărește să-l evite. Preferați metode `protected` (care pot valida sau controla accesul) în locul câmpurilor `protected` directe, ori de câte ori este posibil.

Modificatorul `final`, întâlnit deja pentru câmpuri (Subsecțiunea 4.4.3), se aplică și claselor și metodelor, cu un sens înrudit, dar distinct:

**Listing 6.8:** Clase și metode `final` — interzic moștenirea, respectiv suprascrierea

```

1 public final class ContEconomii extends ContBancar {
2     // aceasta clasa NU mai poate fi extinsa de nimeni
3 }
4
5 public class ContBancar {
6     public final double getSold() {
7         // aceasta metoda NU mai poate fi suprascrisa
8         return sold;
9     }
10 }
```

**Notă**

O clasă `final` nu poate avea subclase; o metodă `final` nu poate fi suprascrisă de nicio subclasă (dar rămâne moștenită normal, ca orice altă metodă). Aceste restricții sunt utile atunci când comportamentul unei clase/metode trebuie garantat neschimbat — de exemplu, clasa `String` din biblioteca standard este, ea însăși, declarată `final`, exact pentru a proteja garanțiile de imutabilitate discutate în Subsecțiunea 3.1.3.

## 6.3 Clasa *Object* – superclasa cosmică

Fiecare clasă din Java, chiar dacă nu scriem explicit `extends`, moștenește implicit din clasa `Object` — rădăcina comună a oricărei ierarhii de clase Java. `Object` definește câteva metode pe care orice obiect le are automat, chiar dacă nu le suprascriem: printre acestea, `equals`, `hashCode` și `toString`.

### 6.3.1 `equals`

Am văzut deja, în Subsecțiunea 3.1.4, distincția dintre `==` (compară referințe) și `equals` (compară conținut). Implementarea implicită a lui `equals`, moștenită din `Object`, se comportă însă identic cu `==`:

```
1 ContBancar cont1 = new ContBancar("Ana Popescu", 500.0);
2 ContBancar cont2 = new ContBancar("Ana Popescu", 500.0);
3
4 System.out.println(cont1.equals(cont2));
5 // false - doua obiecte DIFERITE!
```

**Listing 6.9:** Implementarea implicită a `equals` compară referințe, nu conținut

Pentru ca două conturi cu aceleași date să fie considerate egale, trebuie să suprascriem `equals` explicit:

```
1 @Override
2 public boolean equals(Object obj) {
3     if (this == obj) {
4         return true; // acelasi obiect - scurtatura,
5                     // evident egale
6     }
7     if (!(obj instanceof ContBancar alt)) {
8         return false; // obj e null SAU nu e un ContBancar
9     }
10    return this.titular.equals(alt.titular)
11           && Double.compare(this.sold, alt.sold) == 0;
12 }
```

**Listing 6.10:** Suprascrierea lui `equals`, pentru egalitate bazată pe conținut

**Notă**

Verificarea `obj instanceof ContBancar alt` rezolvă simultan două probleme: returnează `false` dacă `obj` este `null` (operatorul `instanceof` este definit să returneze `false` pentru `null`, nu să arunce excepție) și returnează `false` dacă `obj` nu este de tipul așteptat. Comparăm câmpurile de tip `double` cu `Double.compare`, nu cu `==`, din cauza cazurilor speciale ale reprezentării în virgulă mobilă (`NaN`, zero pozitiv/negativ) pe care `==` nu le tratează corect.

**Definiție — Contractul metodei equals**

Orice implementare a lui `equals` trebuie să respecte cinci proprietăți: *reflexivitate* (`a.equals(a)` e mereu `true`), *simetrie* (`a.equals(b)` implică `b.equals(a)`), *tranzitivitate* (dacă `a.equals(b)` și `b.equals(c)`, atunci `a.equals(c)`), *consistență* (rezultatul nu se schimbă între apeluri repetate, dacă obiectele nu se modifică) și `a.equals(null)` trebuie să returneze mereu `false`.

**6.3.2 hashCode**

Metoda `hashCode` returnează un cod numeric (`int`) asociat unui obiect, folosit intern de colecțiile bazate pe tabele de dispersie (`HashMap`, `HashSet` — discutate pe larg în Capitolul 11–Capitolul 12).

**Listing 6.11:** Suprascrierea lui `hashCode`, coerentă cu `equals`

```
1 @Override
2 public int hashCode() {
3     return Objects.hash(titular, sold);
4 }
```

**Definiție — Contractul equals–hashCode**

Dacă două obiecte sunt egale conform `equals`, ele **trebuie** să aibă același `hashCode`. Reciproca nu este obligatorie: două obiecte cu același `hashCode` nu trebuie neapărat să fie egale (o *coliziune* de hash este permisă, doar mai puțin eficientă).

**Notă**

O greșeală frecventă este suprascrierea lui `equals` **fără** suprascrierea corespunzătoare a lui `hashCode`. Rezultatul: două obiecte considerate egale de `equals` pot ajunge cu `hashCode` diferit, ceea ce rupe complet comportamentul unui `HashSet/HashMap` care le conține (un obiect „egal” cu altul deja adăugat poate să nu fie găsit, fiindcă tabela de dispersie caută mai întâi în funcție de `hashCode`). Regula practică: suprascriseți **întotdeauna** ambele metode împreună, niciodată doar una.

**6.3.3 toString**

Am întâlnit deja consecința practică a implementării implicite a lui `toString` în Subsecțiunea 2.6.4: afișarea unui obiect fără `toString` suprascris produce un șir de forma `NumeClasa@codHash`, nu o reprezentare utilă.

**Listing 6.12:** Suprascrierea lui `toString`, pentru o reprezentare textuală utilă

```
1 @Override
2 public String toString() {
3     return "ContBancar[titular=" + titular + ", sold=" + sold
4         + "]";
5 }
```

**Listing 6.13:** `toString` este apelat automat de `println` și de concatenarea cu `+`

```
1 ContBancar cont = new ContBancar("Ana Popescu", 500.0);
2 System.out.println(cont);
3 // ContBancar[titular=Ana Popescu, sold=500.0]
```

**Notă**

Nu este o coincidență că reprezentarea de mai sus seamănă cu formatul generat automat pentru recorduri (Subsecțiunea 4.5.1, Capitolul 4) — acel format este, de fapt, exact ce generează automat compilatorul pentru un record, motiv suplimentar pentru care recordurile sunt atât de convenabile pentru clase simple, purtătoare de date. Pentru o clasă obișnuită precum ContBancar, această comoditate trebuie scrisă manual, o singură dată.

## 6.4 ArrayList generic; autoboxing

### 6.4.1 ArrayList de bază

Spre deosebire de un array (Secțiunea 2.6), a cărei dimensiune este fixată la creare, un ArrayList este o listă care crește și scade dinamic, pe măsură ce adăugăm sau eliminăm elemente:

```

1 import java.util.ArrayList;
2
3 ArrayList<ContBancar> conturi = new ArrayList<>();
4
5 conturi.add(new ContBancar("Ana Popescu", 500.0));
6 conturi.add(new ContEconomii("Ion Ionescu", 1000.0, 0.02));
7
8 System.out.println(conturi.size());
9 // 2 - metoda, NU proprietate (spre deosebire
10 // de array.length)
11
12 ContBancar primul = conturi.get(0);
13 conturi.remove(0); // elimina primul element;
14 // lista se restrange automat

```

Listing 6.14: ArrayList, o listă cu dimensiune variabilă

**Notă**

Notăția ArrayList<ContBancar> folosește un *parametru de tip* — ContBancar între paranteze unghiulare — care instruește compilatorul să accepte în această listă **doar** obiecte ContBancar (sau subclase ale sale, precum ContEconomii, prin polimorfism). Aceasta este prima noastră întâlnire practică cu *genericile* Java; vom înțelege riguros mecanismul din spatele lor abia în Capitolul 10, dar folosirea de bază, ca mai sus, nu necesită acea înțelegere completă.

O diferență sintactică importantă față de array-uri: ArrayList folosește metode (size(), get(i), add(...), remove(i)), nu operatorul de indexare [] și nici proprietatea .length.

### 6.4.2 Autoboxing și clase wrapper

Parametrul de tip al unui ArrayList (și, în general, al oricărui tip generic) **nu poate fi un tip primitiv** — ArrayList<int> este o eroare de compilare. Pentru fiecare tip primitiv, Java oferă o *clasă wrapper* corespunzătoare, care „împachetează” o valoare primitivă într-un obiect:

**Tabela 6.1:** Câteva clase wrapper corespunzătoare tipurilor primitive (lista completă are opt perechi, câte una pentru fiecare tip din Tabela 2.1).

| Primitiv | Clasă wrapper |
|----------|---------------|
| int      | Integer       |
| double   | Double        |
| boolean  | Boolean       |
| char     | Character     |
| long     | Long          |

**Listing 6.15:** ArrayList de numere întregi, folosind clasa wrapper Integer

```

1 import java.util.ArrayList;
2
3 ArrayList<Integer> numere = new ArrayList<>();
4
5 numere.add(10); // autoboxing: int 10 - devine automat
6                 // Integer.valueOf(10)
7 numere.add(20);
8
9 int primul = numere.get(0); // unboxing: Integer - devine
10                            // automat int

```

#### Notă

Conversia automată, în ambele sensuri, între un tip primitiv și clasa sa wrapper corespunzătoare se numește *autoboxing* (primitiv → obiect) și *unboxing* (obiect → primitiv); compilatorul o inserează automat, fără sintaxă explicită. Comoditatea are însă un cost ascuns: *unboxing*-ul unei referințe null aruncă `NullPointerException` — de exemplu, dacă un `ArrayList<Integer>` conține o valoare null (posibil, fiindcă `Integer` e un tip obiect) și o extragem într-un `int`, programul eșuează la execuție.

#### Notă

O capcană mai subtilă: `Integer` este, la rândul său, un tip obiect, deci `==` pe două variabile `Integer` compară referințe, exact ca la `String` (Subsecțiunea 3.1.4). Java păstrează, din motive de performanță, un cache intern pentru valorile `Integer` mici (-128 până la 127), astfel încât `Integer.valueOf(100) == Integer.valueOf(100)` este `true`, dar `Integer.valueOf(200) == Integer.valueOf(200)` este `false` — un comportament inconsecvent în aparență, cauzat de un detaliu de implementare. Folosiți întotdeauna `equals` pentru compararea valorilor `Integer` (sau a oricărei alte clase wrapper), niciodată `==`.

## 6.5 Metode cu număr variabil de argumente (*varargs*)

Uneori numărul de argumente ale unei metode nu este cunoscut dinainte — de exemplu, o metodă care calculează suma unui număr oarecare de depuneri. Java oferă sintaxa *varargs* (...) exact pentru acest caz:

**Listing 6.16:** O metodă *varargs*, care acceptă un număr oarecare de argumente

```

1 public static double sumaTotala(double... sume) {
2     double total = 0;
3     for (double suma : sume) {
4         total += suma;
5     }
6     return total;

```

```

7 | }
1 | double t1 = sumaTotala(); // 0 argumente - valid, total = 0
2 | double t2 = sumaTotala(100.0); // 1 argument
3 | double t3 = sumaTotala(100.0, 250.0, 75.5); // 3 argumente

```

**Listing 6.17:** Apeluri ale unei metode varargs, cu număr diferit de argumente

#### Notă

În interiorul metodei, parametrul varargs (sume de mai sus) se comportă exact ca un array obișnuit (`double[]`) — de altfel, aceasta este și implementarea reală: la compilare, argumentele individuale sunt automat împachetate într-un array. O metodă poate avea **cel mult un** parametru varargs, și acesta trebuie să fie **ultimul** din lista de parametri. Ați mai folosit deja o metodă varargs, fără să știți: `String.format/printf` (Subsecțiunea 3.3.2) acceptă un număr variabil de argumente exact prin acest mecanism.

## 6.6 Clase abstracte, enum avansat, clase sealed

### 6.6.1 Clase abstracte

O *clasă abstractă* declară structura comună a unei familii de subclase, dar lasă neimplementate una sau mai multe metode — acestea devin obligatorii pentru orice subclasă concretă:

```

1 | public abstract class Forma {
2 |
3 |     public abstract double arie(); // FARA implementare -
4 |                                   // doar semnatura
5 |
6 |     public String descriere() {
7 |         return "Aceasta forma are aria " + arie();
8 |         // poate apela metoda abstracta
9 |     }
10 | }

```

**Listing 6.18:** O clasă abstractă, cu o metodă abstractă

```

1 | public class Cerc extends Forma {
2 |     private double raza;
3 |
4 |     public Cerc(double raza) {
5 |         this.raza = raza;
6 |     }
7 |
8 |     @Override
9 |     public double arie() {
10 |         return Math.PI * raza * raza;
11 |     }
12 | }
13 |
14 | public class Dreptunghi extends Forma {
15 |     private double latime, inaltime;
16 |
17 |     public Dreptunghi(double latime, double inaltime) {
18 |         this.latime = latime;
19 |         this.inaltime = inaltime;
20 |     }
21 |
22 |     @Override
23 |     public double arie() {

```

**Listing 6.19:** Subclase concrete, obligate să implementeze metoda abstractă

Listing 6.20: O clasă abstractă nu poate fi instanțiată direct

```

24         return latime * inaltime;
25     }
26 }

1 Forma f = new Forma();
2     // EROARE DE COMPILARE - clasa abstracta
3 Forma f2 = new Cerc(5.0);
4     // valid - Cerc e o subclasa CONCRETA
5 System.out.println(f2.descriere());
6     // Aceasta forma are aria 78.53...

```

#### Notă

O clasă abstractă poate avea, la fel ca oricare alta, câmpuri, constructori și metode complet implementate (precum `descriere()` de mai sus) — diferența esențială este că **nu poate fi instanțiată** cu `new` direct, doar prin intermediul subclaselor sale concrete. Alegeți o clasă abstractă (în locul unei interfețe, discutate în Capitolul 7) atunci când subclasele trebuie să partajeze și cod comun, nu doar o semnătură de metodă.

## 6.6.2 Clase enumerate, revizitate

Am introdus tipurile enum superficial în Subsecțiunea 2.3.2, ca o alternativă tipizată la constante întregi. Un enum Java poate fi însă mult mai bogat: fiecare constantă poate avea propriile câmpuri, iar tipul poate avea constructori și metode, exact ca o clasă:

Listing 6.21: Un enum cu câmpuri și constructor propriu

```

1 public enum TipTranzactie {
2     DEPUNERE("Depunere in cont"),
3     RETRAGERE("Retragere din cont");
4
5     private final String descriere;
6
7     TipTranzactie(String descriere) {
8         // constructorul unui enum e IMPLICIT private
9         this.descriere = descriere;
10    }
11
12    public String getDescriere() {
13        return descriere;
14    }
15 }

```

Listing 6.22: Folosirea constantelor enumerate cu câmpuri proprii

```

1 TipTranzactie t = TipTranzactie.DEPUNERE;
2 System.out.println(t.getDescriere()); // "Depunere in cont"
3 System.out.println(t.name());
4     // "DEPUNERE" - numele constantei, ca String
5 System.out.println(t.ordinal());
6     // 0 - pozitia in ordinea declararii

```

#### Notă

Constructorul unui enum este apelat automat, o singură dată pentru fiecare constantă, la încărcarea clasei — nu poate fi apelat explicit cu `new`, motiv pentru care este implicit `private` (chiar dacă am fi scris `public`, compilatorul l-ar trata la fel). Recordul `Tranzactie` din Subsecțiunea 4.5.1 (Capitolul 4) folosea un simplu `String` pentru câmpul `tip`; un enum precum `TipTranzactie` ar fi, în practică, o alegere

mai sigură — elimină posibilitatea unei valori greșit scrise (de exemplu "depunre"), imposibilă cu un enum, dar perfect validă sintactic cu un String simplu.

Fiecare constantă poate, la nevoie, să-și suprascrie propriul comportament, printr-un corp de clasă anonim:

```

1 public enum TipTranzactie {
2     DEPUNERE {
3         @Override
4         public double aplica(double sold, double suma) {
5             return sold + suma;
6         }
7     },
8     RETRAGERE {
9         @Override
10        public double aplica(double sold, double suma) {
11            return sold - suma;
12        }
13    };
14
15    public abstract double aplica(double sold, double suma);
16 }

```

Listing 6.23: Comportament diferit per constantă, printr-o metodă abstractă în enum

### 6.6.3 Clase sealed

O clasă (sau interfață) sealed restricționează explicit ce anume poate să o extindă — un compromis între o clasă obișnuită (oricine o poate extinde) și o clasă final (nimeni nu o poate extinde):

```

1 public abstract sealed class Forma permits Cerc, Dreptunghi {
2
3     public abstract double arie();
4 }
5
6 public final class Cerc extends Forma {
7     // ...
8 }
9
10 public final class Dreptunghi extends Forma {
11     // ...
12 }

```

Listing 6.24: O ierarhie sealed, cu sub-clase permise explicit

#### Notă

Clauza `permits Cerc, Dreptunghi` declară explicit lista completă de subclase directe autorizate — orice altă clasă care ar încerca `extends Forma` produce eroare de compilare. Fiecare subclasă permisă trebuie, la rândul ei, să fie declarată `final` (nu mai poate fi extinsă), `sealed` (își definește propria listă restrânsă de subclase) sau `non-sealed` (redevine complet deschisă extinderii, explicit).

#### Notă

Beneficiul principal al claselor sealed devine vizibil abia în combinație cu pattern matching-ul exhaustiv (Capitolul 8): deoarece compilatorul cunoaște toate subclasele posibile ale unei ierarhii sealed, poate verifica la compilare dacă un `switch` pe tipul obiectului acoperă toate

cazurile posibile, fără a mai necesita o ramură default de siguranță.

### Rezumat

- ▶ `extends` creează o subclasă, care moștenește membrii publici/-protejați ai superclasei; `super(...)` invocă explicit constructorul superclasei, iar `super.metoda(...)` invocă explicit versiunea suprascrisă.
- ▶ Polimorfismul rezolvă apelurile de metode suprascrise pe baza tipului **real** al obiectului (legare dinamică), nu a tipului declarat al variabilei; `instanceof` cu `pattern matching` permite acces sigur la funcționalitatea specifică unei subclase.
- ▶ Fiecare clasă moștenește implicit din `Object`; suprascrierea corectă a lui `equals` trebuie însoțită mereu de suprascrierea lui `hashCode`, iar `toString` oferă o reprezentare textuală utilă în locul celei implicite.
- ▶ `ArrayList` oferă o listă cu dimensiune variabilă; parametrul său de tip nu poate fi un tip primitiv, de unde necesitatea claselor wrapper și a conversiei automate (autoboxing/unboxing).
- ▶ `Varargs (tip... nume)` permite un număr variabil de argumente, tratate în corpul metodei ca un array obișnuit.
- ▶ O clasă abstractă lasă una sau mai multe metode neimplementate și nu poate fi instanțiată direct; un `enum` poate avea câmpuri, constructori și metode proprii, inclusiv comportament diferit per constantă; o clasă `sealed` restrânge explicit lista subclaselor permise.

### Exerciții

1. Adăugați la `ContEconomii` o suprascriere corectă a lui `toString`, care să includă și rata dobânzii, pe lângă câmpurile moștenite de la `ContBancar`.
2. Scrieți o metodă statică `varargs soldMediu(ContBancar... conturi)` care calculează soldul mediu al unui număr oarecare de conturi primite ca argumente.
3. Adăugați o a treia formă geometrică, `Triunghi` (cu bază și înălțime), la ierarhia `Forma` din Subsecțiunea 6.6.1. Dacă ierarhia este `sealed` (Subsecțiunea 6.6.3), ce trebuie modificat pentru ca noua clasă să fie acceptată?
4. Rescrieți enumul `TipTranzactie` astfel încât fiecare constantă să știe să valideze dacă o sumă este acceptabilă pentru tipul respectiv de tranzacție (de exemplu, o retragere nu poate depăși un plafon zilnic fix, diferit de cel al unei depuneri).
5. Explicați, folosind exemplul `Integer.valueOf(100) == Integer.valueOf(100)` versus `Integer.valueOf(200) == Integer.valueOf(200)`, de ce concluzia corectă nu este „`==` funcționează uneori pentru `Integer`”, ci „`==` nu trebuie folosit niciodată pentru compararea claselor wrapper”.

## Introducere

Clasele abstracte (vezi Subsecțiunea 6.6.1) ne permit să exprimăm un comportament comun, parțial implementat, pe care subclasele îl completează. Uneori însă avem nevoie doar de un **contract** — o promisiune despre ce metode va oferi o clasă, fără nicio implementare comună, și fără constrângerea unei singure superclase (Java nu permite moștenire multiplă de clase). Acesta este rolul *interfețelor*. Vom vedea apoi cum interfețele cu o singură metodă abstractă deschid calea spre *expresii lambda* — o sintaxă concisă, esențială pentru stilul modern de programare Java — și, în final, cum putem defini clase în interiorul altor clase.

## 7.1 Interfețe

### 7.1.1 Conceptul de interfață

O *interfață* declară un set de metode fără implementare (similar metodelor abstracte din Subsecțiunea 6.6.1), pe care orice clasă care o **implementează** trebuie să le definească:

```
1 public interface Notificabil {
2     void notifica(String mesaj);
3 }
4
5 public class ContBancar implements Notificabil {
6     // ... campurile si metodele deja cunoscute ...
7
8     @Override
9     public void notifica(String mesaj) {
10         System.out.println("Notificare pentru " + titular + "
11         : " + mesaj);
12     }
13 }
```

Listing 7.1: O interfață simplă și o clasă care o implementează

## Notă

Spre deosebire de `extends` (o singură superclasă permisă), o clasă poate implementa **oricâte** interfețe simultan, separate prin virgulă (`implements Notificabil, Comparable<ContBancar>`). Aceasta este calea prin care Java compensează absența moștenirii multiple de clase: o clasă are o singură superclasă, dar poate respecta oricâte „contracte” (interfețe) simultan.

Toate metodele unei interfețe sunt, implicit, `public` și `abstracte` — nu este nevoie (deși e permis) să scrieți explicit aceste cuvinte cheie:

```
1 public interface Notificabil {
2     public abstract void notifica(String mesaj);
3     // identic cu declaratia de mai sus
4 }
```

Listing 7.2: Interfața de mai sus, cu modificatorii implicați scriși explicit

### 7.1.2 Metode default, static și private în interfețe

Începând cu Java 8, o interfață poate oferi și metode cu implementare completă — *metode default* — pe care clasele care implementează interfața le moștenesc automat, fără obligația de a le rescrie:

Listing 7.3: O metodă default, cu implementare comună în interfață

```

1 public interface Notificabil {
2     void notifica(String mesaj);
3
4     default void notificaUrgent(String mesaj) {
5         // metoda CU implementare
6         notifica("URGENT: " + mesaj);
7     }
8 }
```

#### Notă

O clasă care implementează `Notificabil` moștenește automat `notificaUrgent`, construită pe baza metodei abstracte `notifica` pe care ea însăși o implementează — fără să fi scris nicio linie de cod suplimentară. Dacă dorește, clasa poate oricând suprascrie și metoda default, exact ca pe o metodă moștenită dintr-o superclasă.

O interfață poate avea și metode static (apelate pe numele interfeței, ca metode utilitare înrudite conceptual cu ea) și metode private (cod auxiliar, folosit doar intern de metodele default ale aceleiași interfețe, invizibil din exterior):

Listing 7.4: Metode static și private într-o interfață

```

1 public interface Notificabil {
2     void notifica(String mesaj);
3
4     static Notificabil implicit() {
5         // metoda STATICA - Notificabil.implicit()
6         return mesaj ->
7             System.out.println("Notificare: " + mesaj);
8     }
9
10    private String formateaza(String mesaj) {
11        // metoda PRIVATA - doar uz intern
12        return "[" + mesaj.toUpperCase() + "];"
13    }
14
15    default void notificaFormat(String mesaj) {
16        notifica(formateaza(mesaj));
17        // foloseste metoda privata
18    }
19 }
```

#### Notă

Metoda static `Notificabil implicit()` de mai sus folosește o expresie lambda (`mesaj -> ...`) pentru a construi rapid o implementare a interfeței — vom explica riguros această sintaxă în Secțiunea 7.2, imediat după această secțiune; pentru moment, e suficient să recunoașteți că o interfață cu o singură metodă abstractă poate fi implementată extrem de concis.

### 7.1.3 Rezolvarea conflictelor între interfețe

Dacă o clasă implementează două interfețe care oferă metode default cu aceeași semnătură, apare un conflict pe care compilatorul **nu** îl poate rezolva singur:

```

1 public interface A {
2     default void salut() {
3         System.out.println("Salut din A");
4     }
5 }
6
7 public interface B {
8     default void salut() {
9         System.out.println("Salut din B");
10    }
11 }
12
13 public class C implements A, B {
14     // OBLIGATORIU - altfel eroare de compilare
15     @Override
16     public void salut() {
17         // alegem explicit implementarea din A
18         A.super.salut();
19     }
20 }

```

Listing 7.5: Conflict între metode default din interfețe diferite

#### Notă

Compilatorul obligă clasa C să suprascrie explicit `salut()`, tocmai pentru că nu poate decide singură care dintre cele două implementări default moștenite ar trebui folosită. Sintaxa `NumeInterfața.super.metoda()` permite invocarea explicită a implementării dintr-o interfață anume, similar cu `super.metoda()` pentru superclase (Subsecțiunea 6.1.3).

### 7.1.4 Interfețe ca *callback*-uri; Comparator

O utilizare extrem de frecventă a interfețelor este ca *parametru* al unei metode — codul apelant furnizează o implementare concretă, iar metoda o invocă la momentul potrivit (un *callback*). Cel mai comun exemplu din biblioteca standard este `Comparator`, folosit pentru a sorta obiecte după un criteriu specificat de apelant:

```

1 import java.util.ArrayList;
2 import java.util.Comparator;
3
4 ArrayList<ContBancar> conturi = new ArrayList<>();
5 conturi.add(new ContBancar("Ion Ionescu", 1200.0));
6 conturi.add(new ContBancar("Ana Popescu", 500.0));
7
8 conturi.sort(new Comparator<ContBancar>() {
9     @Override
10    public int compare(ContBancar c1, ContBancar c2) {
11        return Double.compare(c1.getSold(), c2.getSold());
12    }
13 });

```

Listing 7.6: Sortarea unei liste de conturi, folosind un `Comparator` explicit

**Notă**

Expresia `new Comparator<ContBancar>() { ... }` este o *clasă anonimă* — o clasă fără nume, definită și instanțiată pe loc, care implementează interfața `Comparator`. Vom reveni riguros asupra claselor anonime în Secțiunea 7.4, dar exemplul de mai sus va deveni mult mai concis odată ce introducem expresiile lambda, chiar în secțiunea următoare.

**Notă**

O clasă poate implementa `Comparable<T>` (o singură metodă, `compareTo`) pentru a defini o „ordine naturală” intrinsecă (de exemplu, conturile ordonate implicit după sold); `Comparator<T>`, în schimb, definește o ordine **externă**, la alegerea codului apelant, utilă atunci când aveți nevoie de mai multe criterii de sortare diferite pentru aceeași clasă, fără să modificați clasa însăși.

### 7.1.5 Clonarea obiectelor

Interfața `Cloneable`, din biblioteca standard, marchează o clasă drept „clonabilă” prin metoda moștenită `clone()` din `Object`. Menționăm acest mecanism mai degrabă pentru completitudine istorică decât ca recomandare practică:

**Notă**

Mecanismul `Cloneable/clone()` din Java este, prin consensul larg al comunității, considerat o greșeală de design a bibliotecii standard — comportamentul implicit copiază câmpurile *superficial* (o copie *shallow*, care nu duplică obiectele referite de câmpuri), semnătura metodei este neintuitivă, iar gestionarea corectă necesită cod boilerplate substanțial. Alternativa modernă, recomandată, este un **constructor de copiere** explicit (`public ContBancar(ContBancar altul) { ... }`) sau, pentru clase purtătoare de date, un record (Subsecțiunea 4.5.1), care oferă imutabilitate în locul necesității de a clona.

## 7.2 Expresii Lambda

### 7.2.1 Sintaxă

O *expresie lambda* este o formă concisă de a scrie implementarea unei interfețe cu o **singură** metodă abstractă, fără sintaxa greoaie a unei clase anonime:

**Listing 7.7:** Exemplul cu `Comparator` din Subsecțiunea 7.1.4, rescris cu lambda

```
1 | conturi.sort((c1, c2) -> Double.compare(c1.getSold(), c2.
   |   getSold()));
```

**Notă**

Comparați acest cod cu clasa anonimă echivalentă din Subsecțiunea 7.1.4: aceeași logică, dar fără numele interfeței, fără `@Override`, fără numele metodei, fără acolade suplimentare pentru corpul metodei (când corpul e o singură expresie). Compilatorul deduce toate aceste detalii din context — variabila `conturi.sort(...)` așteaptă un `Com-`

parator<ContBancar>, deci lambda-ul este interpretat automat ca o implementare a metodei compare.

Sintaxa generală a unei lambda este (parametri) -> corp, cu câteva variații:

```
1 // Un singur parametru - parantezele sunt optionale
2 mesaj -> System.out.println(mesaj);
3
4 // Fara parametri
5 () -> System.out.println("Salut!");
6
7 // Mai multi parametri - parantezele sunt OBLIGATORII
8 (a, b) -> a + b;
9
10 // Corp cu mai multe instructiuni - acolade si return explicit
11 (a, b) -> {
12     int suma = a + b;
13     return suma * 2;
14 };
15
16 // Tipurile parametrilor pot fi scrise explicit (rar necesar,
17 // compilatorul le deduce)
18 (int a, int b) -> a + b;
```

Listing 7.8: Variații de sintaxă lambda

## 7.2.2 Interfețe funcționale

### Definiție — Interfață funcțională

O *interfață funcțională* este o interfață cu **exact o** metodă abstractă (poate avea oricâte metode default/static). Doar interfețele funcționale pot fi implementate printr-o expresie lambda — compilatorul are nevoie să știe exact ce metodă „completează” lambda-ul.

```
1 @FunctionalInterface
2 public interface Notificabil {
3     void notifica(String mesaj); // singura metoda abstracta
4
5     default void notificaUrgent(String mesaj) {
6         // default - NU conteaza la numarare
7         notifica("URGENT: " + mesaj);
8     }
9 }

1 Notificabil notificador = mesaj ->
2     System.out.println("Notificare: " + mesaj);
3 notificador.notifica("Sold insuficient");
```

Listing 7.9: O interfață funcțională proprie, adnotată explicit

Listing 7.10: Implementarea interfeței funcționale de mai sus, printr-o lambda

### Notă

Adnotarea `@FunctionalInterface` este **opțională**, dar recomandată: instruieste compilatorul să verifice explicit că interfața are exact o metodă abstractă, semnalând eroare dacă cineva adaugă din greșeală o a doua metodă abstractă, ceea ce ar rupe compatibilitatea cu lambda existente. Este exact analogul lui `@Override` din Subsecțiunea 6.1.3: nu schimbă comportamentul, dar transformă o greșeală silențioasă într-o eroare de compilare vizibilă imediat.

### 7.2.3 Interfețe funcționale standard (`java.util.function`)

Pentru a evita definirea repetată a unor interfete funcționale foarte comune, biblioteca standard oferă, în pachetul `java.util.function`, un set de interfete funcționale generice, gata de folosit:

Tabela 7.1: Interfețele funcționale generice cele mai frecvent folosite din `java.util.function`.

| Interfață                        | Metodă abstractă               | Rol                                  |
|----------------------------------|--------------------------------|--------------------------------------|
| <code>Predicate&lt;T&gt;</code>  | <code>boolean test(T t)</code> | testează o condiție                  |
| <code>Function&lt;T,R&gt;</code> | <code>R apply(T t)</code>      | transformă un T într-un R            |
| <code>Consumer&lt;T&gt;</code>   | <code>void accept(T t)</code>  | consumă o valoare, fără rezultat     |
| <code>Supplier&lt;T&gt;</code>   | <code>T get()</code>           | furnizează o valoare, fără parametri |

Listing 7.11: Folosirea interfetelor funcționale standard, cu lambda

```

1 import java.util.function.Predicate;
2 import java.util.function.Consumer;
3
4 Predicate<ContBancar> areSoldMare = cont ->
5     cont.getSold() > 1000.0;
6 System.out.println(areSoldMare.test(conturi.get(0)));
7 // true sau false
8
9 Consumer<ContBancar> afiseaza = cont ->
10     System.out.println(cont);
11 afiseaza.accept(conturi.get(0));

```

#### Notă

Nu este nevoie să definiți propriile interfete funcționale (ca `Notificabil` mai devreme) pentru majoritatea situațiilor comune — `Predicate`, `Function`, `Consumer` și `Supplier` acoperă cele mai frecvente tipare. Aceste interfete devin deosebit de importante în Capitolul 15–Capitolul 16, unde stau la baza întregului API de procesare a colecțiilor prin operații de tip `filter/map/forEach`.

## 7.3 Referințe la metode și constructori; lambda și `this`

### 7.3.1 Referințe la metode

Atunci când corpul unei lambda constă doar într-un apel către o metodă deja existentă, notația `::` oferă o formă și mai concisă, numită *referință la metodă*:

Listing 7.12: Referință la o metodă statică, echivalentă cu o lambda

```

1 import java.util.function.Function;
2
3 Function<String, Integer> parseaza1 = s ->
4     Integer.parseInt(s); // lambda obisnuita
5 Function<String, Integer> parseaza2 = Integer::parseInt;
6 // referinta la metoda - echivalent

```

Cel mai relevant exemplu pentru firul `ContBancar` al acestui curs este referința la o metodă de instanță, folosită direct ca `Comparator`, în locul lambda-ului scris manual în Subsecțiunea 7.1.4:

```

1 import java.util.Comparator;
2
3 conturi.sort(Comparator.comparingDouble(ContBancar::getSold))
  ;

```

Listing 7.13: Comparator construit dintr-o referință la metodă

Tabela 7.2: Cele trei forme de referințe la metode.

| Formă                                   | Exemplu                          | Echivalent lambda                          |
|-----------------------------------------|----------------------------------|--------------------------------------------|
| Metodă statică                          | <code>Integer::parseInt</code>   | <code>s -&gt; Integer.parseInt(s)</code>   |
| Metodă de instanță, obiect fix          | <code>System.out::println</code> | <code>x -&gt; System.out.println(x)</code> |
| Metodă de instanță, obiect ca parametru | <code>ContBancar::getSold</code> | <code>c -&gt; c.getSold()</code>           |

#### Notă

Diferența dintre a doua și a treia formă din Tabela 7.2 este subtilă, dar importantă: `System.out::println` referă metoda unui obiect deja fixat (`System.out`); `ContBancar::getSold` referă o metodă de instanță **nelegată** de niciun obiect anume — obiectul devine primul parametru al funcției rezultate, exact cum `Comparator.comparingDouble` are nevoie de o funcție care primește un `ContBancar` și returnează un `double`.

### 7.3.2 Referințe la constructori

Notăția `NumeClasa::new` referă un constructor, ca și cum ar fi o metodă — utilă atunci când o interfață funcțională așteaptă o funcție „creatoare” de obiecte:

```

1 import java.util.function.BiFunction;
2
3 BiFunction<String, Double, ContBancar> creeazaCont =
  ContBancar::new;
4
5 ContBancar cont = creeazaCont.apply("Ana Popescu", 500.0); //
  apeleaza constructorul

```

Listing 7.14: Referință la constructor, folosită ca fabrică de obiecte

#### Notă

`BiFunction<T, U, R>` este o rudă a lui `Function<T, R>` din Tabela 7.1, dar cu doi parametri de intrare (`T` și `U`) în loc de unul singur — pachetul `java.util.function` conține numeroase asemenea variante (`BiConsumer`, `BiPredicate`, `UnaryOperator` etc.), pentru diverse combinații de număr de parametri și tip de rezultat. Nu este nevoie să le memorați pe toate; recunoașteți-le după nevoie, pornind de la cele patru fundamentale deja introduse.

### 7.3.3 Domeniul de vizibilitate; lambda și this

O expresie lambda poate folosi variabile din contextul înconjurător — fenomen numit *captură* — dar doar dacă acestea sunt *efectiv finale* (valoarea lor nu se schimbă după inițializare, chiar dacă nu sunt declarate explicit `final`):

```

1 double rataDobandaStandard = 0.02;
2 // niciodata reatribuita dupa acest punct

```

Listing 7.15: O lambda captează o variabilă locală efectiv finală

Listing 7.16: Eroare de compilare: variabila capturată nu este efectiv finală

```

3
4 Function<Double, Double> calculDobanda = sold -> sold *
    rataDobandaStandard;

1 double rata = 0.02;
2 rata = 0.03; // reatribuire - rata NU mai e efectiv finala
3
4 Function<Double, Double> calculDobanda = sold -> sold * rata;
5 // EROARE DE COMPILARE

```

#### Notă

Această restricție există pentru că o lambda poate „supraviețui” contextului în care a fost creată (de exemplu, dacă e stocată într-un câmp sau returnată dintr-o metodă) — permițând captura unei variabile mutabile ar crea ambiguitate despre *ce valoare* anume ar trebui folosită la momentul execuției lambda-ului, posibil mult mai târziu și dintr-un cu totul alt context.

O diferență importantă, ușor de trecut cu vederea, între lambda și clasele anonime din Subsecțiunea 7.1.4: **this în interiorul unei lambda** se referă la instanța clasei **înconjurătoare**, nu la lambda însăși (o lambda nu are propria identitate de obiect, în sensul lui **this**):

Listing 7.17: **this** într-o lambda referă obiectul înconjurător

```

1 public class ContBancar {
2     private String titular;
3
4     public Runnable creeazaSarcina() {
5         return () -> System.out.println(this.titular);
6         // 'this' = obiectul ContBancar
7     }
8 }

```

#### Notă

Într-o clasă anonimă echivalentă, **this** s-ar fi referit la instanța clasei anonime însăși, nu la **ContBancar** înconjurător — pentru a accesa obiectul exterior dintr-o clasă anonimă, ar fi fost nevoie de sintaxa **ContBancar.this.titular**. Această diferență de comportament este un motiv suplimentar pentru care lambda este, în general, opțiunea mai simplă și mai predictibilă, ori de câte ori interfața are o singură metodă abstractă.

## 7.4 Clase imbricate

O clasă poate fi definită **în interiorul** altei clase — utilă atunci când o clasă auxiliară are sens doar în contextul clasei care o conține, fără a merita propriul fișier separat sau propria vizibilitate la nivel de pachet.

### 7.4.1 Clase imbricate statice

O clasă imbricată **static** nu are nevoie de o instanță a clasei exterioare — se comportă, practic, ca orice altă clasă, doar că numele ei este „găzduit” de clasa exterioară:

Listing 7.18: O clasă imbricată statică, folosită ca detaliu intern de implementare

```

1 public class ListaLegata<T> {
2

```

```

3     private static class Nod<T> {
4         T valoare;
5         Nod<T> urmator;
6
7         Nod(T valoare) {
8             this.valoare = valoare;
9         }
10    }
11
12    private Nod<T> prim;
13
14    public void adauga(T valoare) {
15        Nod<T> nodNou = new Nod<>(valoare);
16        nodNou.urmator = prim;
17        prim = nodNou;
18    }
19 }

```

#### Notă

Clasa `Nod` este un detaliu de implementare al `ListaLegata` — nimeni din afara clasei nu are nevoie să știe de existența ei, motiv pentru care este declarată `private static`. Numele complet al clasei este `ListaLegata.Nod`, dar codul din interiorul lui `ListaLegata` o poate folosi direct, ca `Nod`. Folosiți clase imbricate statice, în locul unor clase separate la nivel de pachet, ori de câte ori clasa auxiliară nu are sens independent de clasa care o conține.

### 7.4.2 Clase interne (non-stactice)

Spre deosebire de o clasă imbricată statică, o clasă *internă* (fără `static`) este legată de o instanță **anume** a clasei exterioare și îi poate accesa direct câmpurile, fără nicio calificare specială:

```

1 public class ListaLegata<T> {
2     private Nod<T> prim;
3
4     // Clasa INTERNA - NU are 'static' - fiecare instanta e
5     // legata de un ListaLegata anume
6     public class Iterator {
7         private Nod<T> curent = prim; // acces direct la
8         // campul clasei exterioare!
9
10        public boolean areUrmator() {
11            return curent != null;
12        }
13
14        public T urmator() {
15            T valoare = curent.valoare;
16            curent = curent.urmator;
17            return valoare;
18        }
19    }
20
21    public Iterator iterator() {
22        return new Iterator();
23        // creata in contextul acestei instante de
24        // ListaLegata
25    }
26 }

```

Listing 7.19: O clasă internă, cu acces direct la câmpurile clasei exterioare

**Notă**

O clasă internă nu poate fi instanțiată independent de o instanță a clasei exterioare — sintaxa completă, rar necesară explicit, este `listaLegata.new Iterator()`. În exemplul de mai sus, metoda `iterator()` creează implicit acest „atașament” la instanța curentă, motiv pentru care `Iterator` poate accesa direct câmpul `prim` al acelei instanțe specifice, fără nicio referință explicită.

### 7.4.3 Clase locale și anonime

O *clasă locală* este definită în interiorul unei metode, vizibilă doar în acel bloc de cod — rar folosită în practică, dar utilă atunci când o clasă auxiliară e necesară o singură dată, într-un singur loc:

**Listing 7.20:** O clasă locală, definită în interiorul unei metode

```

1 public void proceseazaConturi(List<ContBancar> conturi) {
2     class Sumar {
3         // clasa LOCALA - vizibila doar in aceasta metoda
4         double totalSold = 0;
5         int numarConturi = 0;
6     }
7
8     Sumar sumar = new Sumar();
9     for (ContBancar c : conturi) {
10         sumar.totalSold += c.getSold();
11         sumar.numarConturi++;
12     }
13     System.out.println("Total: " + sumar.totalSold + " pe " +
14         sumar.numarConturi + " conturi");
15 }
```

O *clasă anonimă*, deja întâlnită în Subsecțiunea 7.1.4, duce conceptul mai departe: nici măcar nu are un nume, fiind definită și instanțiată într-o singură expresie, direct la locul de folosire.

**Notă**

Ierarhia de concizie, de la cea mai explicită la cea mai concisă formă, pentru implementarea unei interfețe cu o singură metodă, este: clasă separată (fișier propriu) → clasă imbricată statică → clasă locală → clasă anonimă → expresie lambda (doar pentru interfețe funcționale). Alegeți forma cea mai simplă care rămâne totuși clară — pentru interfețe funcționale, lambda este aproape întotdeauna alegerea potrivită; pentru interfețe cu mai multe metode abstracte, o clasă anonimă (sau una imbricată/locală) rămâne singura opțiune.

### 7.4.4 Records imbricate

Un record (Subsecțiunea 4.5.1) poate fi, la rândul lui, imbricat într-o altă clasă:

**Listing 7.21:** Un record imbricat, pentru date interne unei clase

```

1 public class ContBancar {
2
3     public record Extras(String descriere, double suma) {
4     }
5
6     private List<Extras> istoricTranzactii = new ArrayList
7     <>();
8 }
```

```

7
8 public void depune(double suma) {
9     sold = sold + suma;
10    istoricTranzactii.add(new Extras("depunere", suma));
11 }
12 }

```

### Notă

Spre deosebire de o clasă imbricată obișnuită, un record imbricat este **implicit static**, chiar dacă nu scrieți explicit `static` — nu poate exista o versiune „internă” (non-statică) a unui record, întrucât un record este, prin natura sa, o clasă purtătoare de date imutabilă, fără nevoie de o legătură implicită cu o instanță exterioară.

### Rezumat

- ▶ O interfață declară un contract de metode; o clasă poate implementa oricâte interfețe simultan, spre deosebire de moștenirea unică de clase.
- ▶ Metodele default oferă implementare comună, moștenită automat; conflictele între metode default din interfețe diferite trebuie rezolvate explicit, cu `Interfata.super.metoda()`.
- ▶ O expresie lambda implementează concis o interfață funcțională (exact o metodă abstractă); referințele la metode (`::`) sunt o formă și mai concisă, atunci când corpul lambda e doar un apel de metodă existentă.
- ▶ O lambda poate captura variabile locale **efectiv finale**; `this` într-o lambda se referă la obiectul înconjurător, spre deosebire de o clasă anonimă, unde `this` se referă la instanța anonimă însăși.
- ▶ Clasele imbricate statice sunt independente de o instanță a clasei exterioare; clasele interne (non-statice) sunt legate de o instanță anume și îi pot accesa direct câmpurile; clasele locale și anonime au domeniu de vizibilitate restrâns la o singură metodă/expresie.
- ▶ Un record imbricat este implicit static.

### Exerciții

1. Definiți o interfață funcțională `ValidatorTranzactie` cu o metodă boolean `esteValida(double suma)`, apoi scrieți trei implementări diferite ale ei: una ca clasă separată, una ca clasă anonimă, una ca lambda.
2. Rescrieți sortarea listei de conturi din Subsecțiunea 7.3.1 astfel încât să sorteze **descrescător** după `sold`, folosind `Comparator.comparingDouble(...).reversed()`.
3. Folosind o referință la constructor (`ContBancar::new`), scrieți o mică metodă care primește o listă de perechi (nume, sold) și construiește o listă de obiecte `ContBancar` corespunzătoare.
4. Adăugați la `ListaLegata` din Subsecțiunea 7.4.1 o metodă `marime()` care parcurge lista folosind clasa internă `Iterator` definită în Subsecțiunea 7.4.2 și numără elementele.
5. Explicați, în cuvinte proprii, de ce un record imbricat este întotdeauna implicit static, în timp ce o clasă imbricată obișnuită poate fi fie statică, fie internă (non-statică), la alegerea programatorului.



## Introducere

Am folosit deja forma modernă a lui `instanceof` în Subsecțiunea 6.2.2, care combină verificarea de tip cu declararea unei variabile convertite. În acest capitol extindem acest mecanism — numit generic *pattern matching* — către formele sale complete: deconstruirea recordurilor direct în tiparul de *matching*, și un `switch` capabil să testeze tipuri, nu doar valori constante. Vom vedea și cum clasele `sealed` din Subsecțiunea 6.6.3 devin, în acest context, un instrument al compilatorului pentru a garanta exhaustivitatea. Capitolul se încheie cu *reflecția* — capacitatea unui program Java de a-și inspecta și manipula propriile clase la runtime.

## 8.1 Pattern matching cu `instanceof`; deconstruirea recordurilor

### 8.1.1 Recapitulare: `instanceof` cu pattern matching

Reamintim forma introdusă în Subsecțiunea 6.2.2: `instanceof` verifică tipul și, dacă verificarea reușește, declară automat o variabilă deja convertită la acel tip:

```
1 Object obj = new ContEconomii("Ana Popescu", 1000.0, 0.02);
2
3 if (obj instanceof ContEconomii economii) {
4     // 'economii' e deja de tip ContEconomii
5     economii.aplicaDobanda();
6 }
```

**Listing 8.1:** Forma de bază, deja cunoscută, a pattern matching-ului cu `instanceof`

### 8.1.2 Deconstruirea recordurilor (*record patterns*)

Pentru un record (Subsecțiunea 4.5.1), pattern matching-ul poate merge un pas mai departe: în loc să declare o singură variabilă pentru întregul record, poate **deconstrui** direct componentele acestuia:

```
1 Object obj = new ContBancar.Extras("depunere", 150.0);
2
3 if (obj instanceof ContBancar.Extras(String descriere, double
4     suma)) {
5     System.out.println(descriere + ": " + suma + " lei");
6     // 'descriere' si 'suma' sunt disponibile direct,
7     // fara obj.descriere()/obj.suma()
8 }
```

**Listing 8.2:** Deconstruirea recordului `Extras`, definit în Subsecțiunea 7.4.4

## Notă

Tiparul `ContBancar.Extras(String descriere, double suma)` verifică simultan că `obj` este de tipul `Extras` și extrage direct cele două componente ale sale, în variabile locale noi. Aceasta este o comoditate semnificativă față de deconstruirea manuală (`var e = (Extras) obj; String descriere = e.descriere(); etc.`), mai ales pentru

recorduri cu multe componente.

### 8.1.3 Pattern-uri imbricate

Componentele unui record pattern pot fi, la rândul lor, alte pattern-uri (inclusiv alte record patterns), permițând deconstruirea unor structuri de date imbricate într-un singur pas:

**Listing 8.3:** Pattern-uri imbricate, pentru structuri de date compuse

```

1 public record Punct(double x, double y) {
2 }
3
4 public record Segment(Punct inceput, Punct sfarsit) {
5 }
6
7 Object obj = new Segment(new Punct(0, 0), new Punct(3, 4));
8
9 if (obj instanceof Segment(Punct(double x1, double y1), Punct
10 (double x2, double y2))) {
11     double lungime = Math.sqrt(Math.pow(x2 - x1, 2) + Math.
12     pow(y2 - y1, 2));
13     System.out.println("Lungime segment: " + lungime);
14 }
```

#### Notă

Pattern-ul de mai sus deconstruiește un Segment direct în cele patru coordonate ale capetelor sale, într-o singură verificare instanceof — fără nicio variabilă intermediară pentru Punct. Puteți folosi var în locul tipului explicit pentru oricare dintre componente (Punct(var x1, var y1)), atunci când tipul e evident din context.

## 8.2 Pattern matching cu switch

### 8.2.1 Switch pe tipuri

Am introdus expresiile switch moderne în Subsecțiunea 2.4.4, cu ramuri bazate pe valori constante. Un switch poate testa acum și **tipul** unei valori, folosind aceeași sintaxă de pattern matching din Secțiunea 8.1:

**Listing 8.4:** switch pe tipuri, cu deconstruire, pe ierarhia sealed din Subsecțiunea 6.6.3

```

1 public static double descrieArie(FormaGeometrica forma) {
2     return switch (forma) {
3         case Cerc c -> Math.PI * c.raza() * c.raza();
4         case Dreptunghi(double latime, double inaltime) ->
5             latime * inaltime;
6         case Triunghi t -> t.arie();
7     };
8 }
```

#### Notă

Observați combinarea celor două tehnici din acest capitol: ramura pentru Dreptunghi folosește direct un record pattern, deconstruind lățimea și înălțimea, în timp ce ramurile pentru Cerc și Triunghi folosesc forma simplă, cu o singură variabilă. Fiecare ramură poate alege independent nivelul de deconstruire de care are nevoie.

### 8.2.2 Gărzi (*guards*)

O *gardă* adaugă o condiție suplimentară unui pattern, folosind cuvântul cheie `when`:

```
1 public static String clasificaCont(ContBancar cont) {
2     return switch (cont) {
3         case ContEconomii ce when ce.getSold() > 10000.0 -> "
Cont economii premium";
4         case ContEconomii ce -> "Cont economii standard";
5         case ContBancar cb when cb.getSold() < 0 -> "Cont in
descoperit";
6         default -> "Cont standard";
7     };
8 }
```

Listing 8.5: O gardă `when`, pentru o condiție suplimentară pe lângă tip

#### Notă

Fără gardă, un pattern de tip `ContEconomii ce` se potrivește cu **ori**ce obiect de acel tip, indiferent de valorile sale interne. Garda `when ce.getSold() > 10000.0` restrânge potrivirea doar la acele obiecte care, în plus față de tip, satisfac și condiția specificată. Ordinea ramurilor contează: prima ramură care se potrivește (tip și gardă, dacă există) este cea executată.

### 8.2.3 Exhaustivitate și dominanță

Pentru un `switch` pe o ierarhie sealed (Subsecțiunea 6.6.3), compilatorul poate verifica **exhaustivitatea**: dacă toate subclasele posibile sunt acoperite, clauza `default` devine inutilă și poate fi omisă:

```
1 public static double descrieArie(FormaGeometrica forma) {
2     // FormaGeometrica e sealed, cu exact 3 subclase permise
3     return switch (forma) {
4         case Cerc c -> Math.PI * c.raza() * c.raza();
5         case Dreptunghi d -> d.latime() * d.inaltime();
6         case Triunghi t -> t.arie();
7         // FARA default -- compilatorul stie ca acestea sunt
8         // SINGURELE subclase posibile
9     };
10 }
```

Listing 8.6: `switch` exhaustiv pe o ierarhie sealed, fără `default`

#### Notă

Acesta este exact beneficiul promis în Subsecțiunea 6.6.3: dacă adăugați o a patra formă geometrică la lista `permits`, dar uitați să adăugați o ramură corespunzătoare în acest `switch`, compilatorul semnalează eroare imediat, în loc să lăsați un caz netratat descoperit abia la execuție. Pentru o ierarhie **obișnuită** (nu sealed), compilatorul nu poate face această garanție — oricine ar putea adăuga o subclasă nouă, oricând, deci un `default` rămâne obligatoriu.

*Dominanța* este o eroare de compilare distinctă: o ramură anterioară care „acoperă” complet o ramură ulterioară, făcând-o de neatin:

```
1 switch (forma) {
2     // prea generala, primul rand!
3     case FormaGeometrica f -> System.out.println("Orice forma
");
4 }
```

Listing 8.7: Eroare de dominanță – o ramură ulterioară devine imposibil de atins

```

4 // EROARE: niciodată atinsă, dominată de randul de mai sus
5 case Cerc c -> System.out.println("Cerc");
6 }

```

#### Notă

Ordinea ramurilor unui switch pe tipuri contează: o ramură cu un tip mai general (aici, chiar tipul de bază `FormaGeometrica`) trebuie plasată **după** ramurile cu tipuri mai specifice, altfel acestea din urmă nu ar fi niciodată atinse — compilatorul detectează această situație și refuză să compileze codul, în loc să lase o eroare logică ascunsă.

### 8.2.4 Gestionarea null în switch

Un switch clasic (Subsecțiunea 2.5.3) aruncă `NullPointerException` dacă valoarea testată este null. Un switch cu pattern matching permite, în schimb, tratarea explicită a lui null ca pe un caz separat:

**Listing 8.8:** Tratarea explicită a lui null într-un switch cu pattern matching

```

1 public static String descrie(Object obj) {
2     return switch (obj) {
3         case null -> "Nicio valoare";
4         case ContEconomii ce -> "Cont economii: " + ce.
5             getSold();
6         case ContBancar cb -> "Cont standard: " + cb.getSold
7             ();
8         default -> "Tip necunoscut";
9     };
10 }

```

#### Notă

Ramura `case null` trebuie scrisă explicit — fără ea, un switch cu pattern matching aruncă în continuare `NullPointerException` pentru o valoare null, exact ca varianta clasică. Această cerință explicită este intenționată: obligă programatorul să decidă conștient ce se întâmplă în cazul null, în loc să se bazeze pe un comportament implicit ușor de uitat, revenind la discuția despre referințe null din Subsecțiunea 4.3.2.

## 8.3 Reflecție: clasa `Class` și inspecția la runtime

Până acum, structura unei clase (câmpuri, metode, constructori) a fost cunoscută la **compilare** — scriem `cont.getSold()` pentru că știm dinainte că metoda există. *Reflecția* permite unui program să inspecteze și să folosească această structură la **execuție**, chiar și pentru clase necunoscute la momentul scrierii codului.

### 8.3.1 Clasa `Class`

Fiecare clasă și interfață din Java are un obiect `Class` asociat, la runtime, care descrie complet structura ei. Acest obiect poate fi obținut în trei moduri:

```

1 ContBancar cont = new ContBancar("Ana Popescu", 500.0);
2
3 Class<?> c1 = cont.getClass(); // din orice obiect existent
4 Class<?> c2 = ContBancar.class; // literal de clasa,
5                                // cunoscut la compilare
6 // din numele complet, ca String:
7 Class<?> c3 = Class.forName("ro.unitbv.mip.banking.ContBancar
  ");

1 System.out.println(c1.getName());
2 // ro.unitbv.mip.banking.ContBancar
3 System.out.println(c1.getSimpleName()); // ContBancar
4 System.out.println(c1.getSuperclass()); // class
5                                           // java.lang.Object

```

Listing 8.9: Trei moduri de a obține obiectul `Class` al unei claseListing 8.10: Informații de bază obținute dintr-un obiect `Class`**Notă**

A treia formă, `Class.forName(...)`, este singura care acceptă numele clasei ca `String`, cunoscut abia la execuție — de exemplu, citit dintr-un fișier de configurare. Aceasta este exact tehnica prin care multe *framework*-uri încarcă dinamic clase pe care nu le cunosc la momentul propriei compilări, un subiect la care revenim în Subsecțiunea 8.4.3.

### 8.3.2 Analiza câmpurilor și metodelor la runtime

Obiectul `Class` oferă acces la listele complete de câmpuri și metode ale unei clase, indiferent de nivelul lor de acces:

```

1 import java.lang.reflect.Field;
2 import java.lang.reflect.Method;
3
4 Class<?> clasa = ContBancar.class;
5
6 for (Field f : clasa.getDeclaredFields()) {
7     System.out.println(f.getType().getSimpleName() + " " + f.
8         getName());
9 }
10 for (Method m : clasa.getDeclaredMethods()) {
11     System.out.println(m.getName());
12 }

```

Listing 8.11: Enumerarea câmpurilor și metodelor unei clase, prin reflecție

**Notă**

`getDeclaredFields()/getDeclaredMethods()` returnează **toți** membrii declarați direct în clasă, indiferent de modificatorul de acces (inclusiv `private`) — spre deosebire de `getFields()/getMethods()`, care returnează doar membrii `public`, dar incluzând și cei moșteniți din superclase.

Reflecția poate chiar **citi valoarea** unui câmp privat, dintr-un obiect concret, ocolind încapsularea (Secțiunea 4.4):

```

1 Field campSold = ContBancar.class.getDeclaredField("sold");
2 // dezactiveaza temporar verificarea de acces:
3 campSold.setAccessible(true);
4 // citeste direct campul privat 'sold':

```

Listing 8.12: Citirea unui câmp privat prin reflecție, ocolind încapsularea

```
5 | double valoare = (double) campSold.get(cont);
```

#### Notă

setAccessible(true) este o capacitate **puternică și periculoasă**: anulează exact protecția pe care încapsularea (Secțiunea 4.4) o oferă în mod normal. Folosiți-o doar în cod de infrastructură (biblioteci de testare, cadre de serializare/injecție de dependențe), niciodată în codul obișnuit al unei aplicații — accesarea directă a câmpurilor private ale unei clase pe care nu o controlați anulează orice garanție pe care acea clasă încearcă să o mențină despre propria stare internă.

## 8.4 Invocare dinamică; Service Loaders și Proxy

### 8.4.1 Invocarea dinamică a metodelor

**Listing 8.13:** Invocarea unei metode, aleasă și apelată dinamic, prin reflecție

```
1 | import java.lang.reflect.Method;
2 |
3 | Method metodaGetSold = ContBancar.class.getMethod("getSold");
4 | // echivalent cu cont.getSold():
5 | double sold = (double) metodaGetSold.invoke(cont);
6 |
7 | // echivalent cu cont.depune(150.0):
8 | metodaDepune.invoke(cont, 150.0);
```

#### Notă

getMethod(nume, tipuriParametri...) localizează metoda dorită după numele ei și tipurile parametrilor (necesar pentru a distinge între metode supraîncărcate, Subsecțiunea 4.3.4); invoke(obiect, argumente...) o apelează efectiv, pe obiectul specificat. Rezultatul are tipul Object și trebuie convertit explicit (cast) la tipul real returnat, cunoscut de programator, dar nu de compilator.

### 8.4.2 Invocarea dinamică a constructorilor

**Listing 8.14:** Construirea unui obiect dinamic, prin reflecție, fără new

```
1 | import java.lang.reflect.Constructor;
2 |
3 | Constructor<ContBancar> constructor =
4 | ContBancar.class.getConstructor(String.class, double.class);
5 |
6 | ContBancar contNou = constructor.newInstance("Ion Ionescu",
7 | 1200.0);
```

#### Notă

Această tehnică este exact mecanismul prin care multe *framework*-uri (de exemplu, cele de injecție de dependențe sau de serializare JSON/XML) pot crea obiecte ale unor clase scrise de dezvoltatorul aplicației, deși cadrul de lucru însuși a fost compilat cu mult înainte ca aceste clase să existe — cadrul cunoaște doar **numele** clasei (posibil dintr-un fișier de configurare) și folosește reflecția pentru a o instanția la momentul potrivit.

### 8.4.3 Service Loaders și Proxy (context)

Menționăm, pentru completitudine, două mecanisme din biblioteca standard construite pe fundația reflecției, fără a intra în detalii de implementare:

- **ServiceLoader** descoperă, la execuție, toate implementările disponibile ale unei interfețe, listate în fișiere de configurare speciale (`META-INF/services/...`) — mecanismul standard prin care biblioteci și module Java pot oferi implementări *pluggable*, fără ca modulul care le folosește să cunoască dinainte clasele concrete.
- **java.lang.reflect.Proxy** generează, la execuție, o clasă care implementează una sau mai multe interfețe date, redirectionând fiecare apel de metodă către un cod comun de tratare — tehnica din spatele multor cadre de lucru populare (acces la baze de date, testare cu obiecte simulate/*mock*, interceptarea apelurilor pentru logging sau tranzacții).

#### Notă

Ambele mecanisme sunt folosite intens în interiorul bibliotecilor și *framework*-urilor pe care le veți folosi ca dezvoltatori, dar rareori direct, în codul unei aplicații obișnuite. Le menționăm aici tocmai pentru ca, atunci când le veți întâlni (de exemplu, într-un *framework* de injecție de dependențe), să recunoașteți fundația de reflecție pe care se bazează, nu ca pe o „magie” de neînțeles.

#### Rezumat

- Pattern matching-ul cu `instanceof` poate deconstrui direct componentele unui record (*record pattern*), inclusiv imbricat, pentru structuri de date compuse.
- Un `switch` pe tipuri combină testarea tipului cu deconstruirea; gărziile (*when*) adaugă condiții suplimentare; ordinea ramurilor contează, pentru a evita erorile de dominanță.
- Pe o ierarhie sealed, compilatorul poate verifica exhaustivitatea unui `switch`, eliminând nevoia unui `default`; ramura case `null` tratează explicit valorile nule, altfel netratate implicit.
- Reflecția (clasa `Class` și pachetul `java.lang.reflect`) permite inspectarea și manipularea structurii unei clase la execuție: enumerarea câmpurilor/metodelor, citirea câmpurilor private (cu `setAccessible(true)`), invocarea dinamică a metodelor și constructorilor.
- `ServiceLoader` și `Proxy` sunt mecanisme din biblioteca standard, construite pe reflecție, care stau la baza multor cadre de lucru — utile de recunoscut, chiar dacă rar folosite direct.

#### Exerciții

1. Scrieți o metodă `descrieArie(FormaGeometrica forma)` folosind `switch` cu pattern matching și `record patterns`, pentru ierarhia `Cerc/Dreptunghi/Triunghi` din Subsecțiunea 6.6.3, fără clauză `default`.
2. Adăugați o gardă (*when*) la exercițiul anterior, care tratează separat cazul unui `Cerc` cu raza mai mare de 100.
3. Folosind pattern-uri imbricate, scrieți o metodă care primește un `Object` și, dacă acesta este un `Segment` (din Subsecțiunea 8.1.3)

cu ambele capete în origine (0, 0), afișează un mesaj distinct.

4. Scrieți un mic program care, folosind reflecție, enumeră toate metodele publice ale clasei `ContBancar` și afișează numele fiecăreia.
5. Folosind reflecție, construiți dinamic un obiect `ContBancar` (prin `Constructor.newInstance`) și apelați metoda `depune` pe acesta (prin `Method.invoke`), fără a folosi direct `new` sau apelul explicit al metodei.

## Introducere

De-a lungul cursului am întâlnit repetat situații care pot eșua la execuție: un index de array în afara limitelor (Secțiunea 2.6), o referință null apelată (Subsecțiunea 4.3.2), o intrare de la tastatură care nu se poate converti la numărul cerut (Subsecțiunea 3.3.1), un cast eșuat (Subsecțiunea 6.2.2). Am amânat, de fiecare dată, explicarea completă a mecanismului — acesta este momentul. *Excepțiile* sunt modul Java de a semnaliza și gestiona erori la execuție; *aserțiile* și *logging-ul* sunt instrumente complementare, pentru verificarea ipotezelor interne ale programului și, respectiv, pentru înregistrarea structurată a evenimentelor din timpul execuției.

## 9.1 Clasificarea excepțiilor

### 9.1.1 Ierarhia claselor de excepții

Orice condiție excepțională din Java este reprezentată printr-un obiect, instanță a unei clase din ierarhia Throwable:

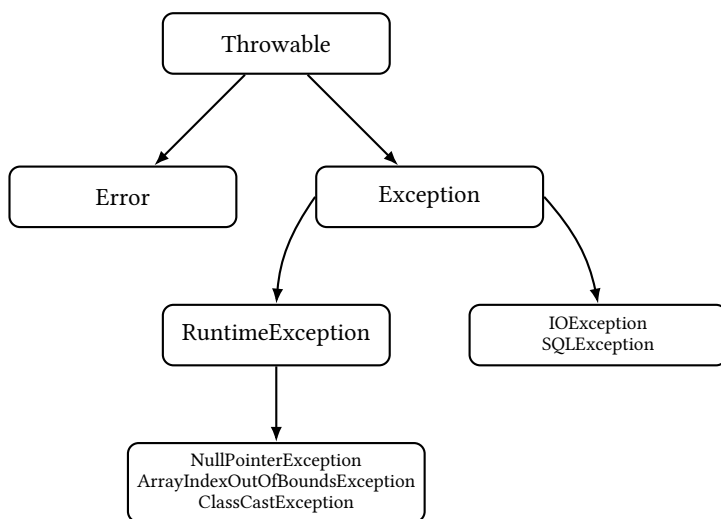


Figura 9.1: Ierarhia (simplificată) a claselor de excepții din Java.

## Notă

Error reprezintă condiții grave, din care programul de obicei nu se poate recupera (de exemplu, memorie insuficientă) — nu sunt gândite să fie capturate de codul obișnuit al aplicației. Exception reprezintă condiții din care programul **poate și ar trebui** să se recupereze; acest capitol se concentrează aproape exclusiv pe această ramură a ierarhiei.

### 9.1.2 Excepții verificate versus neverificate

Java împarte excepțiile din ramura Exception în două categorii, cu tratament diferit din partea compilatorului:

**Definiție — Excepții verificate și neverificate**

O excepție *verificată* (*checked*) — orice subclasă a lui `Exception` care **nu** este și subclasă a lui `RuntimeException` — trebuie fie capturată, fie declarată explicit în semnătura metodei (`throws`), altfel codul nu compilează. O excepție *neverificată* (*unchecked*) — orice subclasă a lui `RuntimeException` — nu impune nicio astfel de obligație compilatorului.

**Notă**

Toate excepțiile întâlnite până acum în acest curs (`NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`, `IllegalArgumentException`) sunt **neverificate** — de aceea codul din capitolele anterioare a compilat fără nicio mențiune de `try/catch` sau `throws`, deși aceste excepții pot apărea la execuție. Excepțiile verificate sunt tipic asociate operațiilor care depind de resurse externe, predispuse la eșec independent de corectitudinea codului (citirea unui fișier, o conexiune de rețea) — vom întâlni exemple concrete în Capitolul 17 și Capitolul 20.

### 9.1.3 Declararea și aruncarea excepțiilor

O metodă care poate arunca o excepție verificată, fără să o captureze ea însăși, trebuie să declare acest lucru explicit, cu clauza `throws`:

**Listing 9.1:** Declararea unei excepții verificate cu `throws`, și aruncarea ei cu `throw`

```
1 public void incarcaDinFisier(String cale) throws IOException
2 {
3     if (cale == null || cale.isBlank()) {
4         // neverificata - fara throws necesar
5         throw new IllegalArgumentException("Calea nu poate fi
6         goala");
7     }
8     // ... cod care poate arunca IOException, propagata mai
9     // departe ...
10 }
```

**Notă**

Clauza `throws` din semnătura metodei este o parte a *contractului public* al acesteia — informează orice apelant că trebuie să se pregătească pentru acea excepție (fie prin capturare, fie prin propagare mai departe, cu propriul `throws`). Excepțiile neverificate **nu** necesită această declarare, deși nimic nu interzice menționarea lor explicită în `throws`, ca formă de documentare.

### 9.1.4 Clase de excepții proprii

O clasă de excepție proprie extinde, de regulă, `Exception` (pentru o excepție verificată) sau `RuntimeException` (pentru una neverificată):

**Listing 9.2:** O excepție verificată proprie, pentru fonduri insuficiente

```
1 public class FonduriInsuficienteException extends Exception {
2     private final double sumaLipsa;
3
4     public FonduriInsuficienteException(String mesaj, double
5     sumaLipsa) {
6         super(mesaj);
7     }
8 }
```

```

6         this.sumaLipsa = sumaLipsa;
7     }
8
9     public double getSumaLipsa() {
10         return sumaLipsa;
11     }
12 }

```

```

1 public void retrage(double suma) throws
    FonduriInsuficienteException {
2     if (suma > sold) {
3         double lipsa = suma - sold;
4         throw new FonduriInsuficienteException("Fonduri
        insuficiente", lipsa);
5     }
6     sold = sold - suma;
7 }

```

**Listing 9.3:** ContBancar.retrage, rescris să arunce excepția proprie, în loc să returneze boolean

### Notă

Comparați această versiune cu cea originală din Subsecțiunea 4.3.1, care returna boolean. Diferența de design este semnificativă: returnarea unui boolean obligă apelantul să **verifice** manual rezultatul (ușor de omis din greșeală), în timp ce o excepție **nu poate fi ignorată** tăcut — fie e capturată, fie propagată, dar codul care o omite complet nu compilează (pentru o excepție verificată). Alegerea între cele două stiluri depinde de context: o excepție e potrivită pentru o condiție cu adevărat „excepțională”; un rezultat boolean/Optional e adesea mai potrivit pentru un eșec așteptat, frecvent, făcând parte din fluxul normal al programului.

## 9.2 Capturarea excepțiilor

### 9.2.1 try/catch/finally

```

1 try {
2     cont.retrage(2000.0);
3     System.out.println("Retragere reusita");
4 } catch (FonduriInsuficienteException e) {
5     System.out.println("Eroare: " + e.getMessage() + " (
        lipsesc " + e.getSumaLipsa() + " lei");
6 } finally {
7     // se executa INTOTDEAUNA, cu sau fara exceptie
8     System.out.println("Operatiune incheiata");
9 }

```

**Listing 9.4:** Structura de bază try/catch/finally

### Notă

Blocul `finally` se execută necondiționat — indiferent dacă blocul `try` a reușit, a aruncat o excepție (capturată sau nu de blocurile `catch`), sau a conținut chiar un `return`. Este locul potrivit pentru eliberarea resurselor (fișiere, conexiuni) care **trebuie** închise, indiferent de rezultat — deși, așa cum vom vedea în Subsecțiunea 9.3.1, Java oferă azi o sintaxă mai sigură pentru exact acest caz.

## 9.2.2 Capturare multiplă (multi-catch)

Mai multe blocuri catch pot trata tipuri diferite de excepții, în ordine, sau un singur bloc poate trata mai multe tipuri simultan, dacă tratamentul e identic:

**Listing 9.5:** Blocuri catch multiple, versus un singur bloc multi-catch

```

1 try {
2     proceseazaTranzactie();
3 } catch (FonduriInsuficienteException e) {
4     System.out.println("Fonduri insuficiente: " + e.
5         getMessage());
6 } catch (IllegalArgumentException e) {
7     System.out.println("Argument invalid: " + e.getMessage())
8     ;
9 }
10 // Echivalent, daca tratamentul ar fi identic pentru ambele
11 // tipuri:
12 try {
13     proceseazaTranzactie();
14 } catch (FonduriInsuficienteException |
15     IllegalArgumentException e) {
16     System.out.println("Eroare la procesare: " + e.getMessage
17         ());
18 }

```

### Notă

Ordinea blocurilor catch contează atunci când tipurile de excepții sunt înrudite prin moștenire: un bloc pentru o superclasă (de exemplu Exception) trebuie plasat **după** blocurile pentru subclase mai specifice, altfel compilatorul semnalează eroare — exact aceeași regulă de dominanță întâlnită deja la switch pe tipuri, în Subsecțiunea 8.2.3.

## 9.2.3 Re-aruncare și *chaining* de excepții

Uneori dorim să capturăm o excepție doar pentru a o „înveli” într-un tip mai relevant pentru contextul curent, păstrând totuși excepția originală ca *cauză*:

**Listing 9.6:** *Chaining*: excepția originală devine cauza celei noi

```

1 public void proceseazaTranzactie() {
2     try {
3         cont.retrage(2000.0);
4     } catch (FonduriInsuficienteException e) {
5         throw new RuntimeException("Tranzactia a esuat", e);
6         // 'e' devine
7         CAUZA
8     }
9 }

```

**Listing 9.7:** Recuperarea cauzei originale, dintr-o excepție înlănțuită

```

1 try {
2     proceseazaTranzactie();
3 } catch (RuntimeException e) {
4     System.out.println("Eroare: " + e.getMessage());
5     System.out.println("Cauzata de: " + e.getCause()); //
6     acceseaza FonduriInsuficienteException originala
7 }

```

**Notă**

Fără *chaining* (adică aruncând o excepție nouă, fără a pasa excepția originală drept cauză), informația despre **motivul real** al eșecului s-ar pierde complet — am ști doar că „tranzacția a eșuat”, nu și de ce. Constructorul cu doi parametri (mesaj, cauza), disponibil pentru majoritatea claselor de excepții din biblioteca standard (și recomandat și pentru cele proprii), păstrează întregul lanț de cauzalitate, vizibil integral în *stack trace*-ul afișat — subiectul secțiunii următoare.

## 9.3 try-with-resources; stack trace; bune practici

### 9.3.1 try-with-resources

Multe resurse (fișiere, conexiuni de rețea, conexiuni la baze de date) trebuie închise **explicit** după utilizare, indiferent dacă operația a reușit sau a aruncat o excepție. Închiderea manuală, într-un bloc *finally* (Subsecțiunea 9.2.1), este verboasă și ușor de scris greșit. Sintaxa *try-with-resources* rezolvă elegant această problemă:

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4
5 try (Scanner cititor = new Scanner(new File("conturi.txt")))
6 {
7     while (cititor.hasNextLine()) {
8         System.out.println(cititor.nextLine());
9     }
10 } catch (FileNotFoundException e) {
11     System.out.println("Fișierul nu a putut fi găsit: " + e.
12         getMessage());
13 }
14 // cititor.close() este apelat AUTOMAT, chiar dacă apare o
15 // excepție în buclă
```

**Listing 9.8:** try-with-resources: resursa se închide automat, garantat

**Notă**

Orice clasă care implementează interfața *AutoCloseable* (o singură metodă abstractă, *close()*) poate fi folosită în paranteza de după *try*. Resursa este închisă automat la ieșirea din bloc, în ordine **inversă** declarării, indiferent dacă blocul s-a încheiat normal sau printr-o excepție — echivalent cu un *finally* scris corect, dar fără riscul de a-l uita sau de a-l scrie greșit. Vom folosi intens această sintaxă în Capitolul 17 (fișiere) și Capitolul 21 (conexiuni JDBC).

```
1 try (Scanner cititor = new Scanner(new File("intrare.txt"));
2     var scriitor = new java.io.PrintWriter("iesire.txt")) {
3     // ambele resurse se închid automat,
4     // în ordine inversă declarării
5 } catch (FileNotFoundException e) {
6     // ...
7 }
```

**Listing 9.9:** Mai multe resurse în același *try*, separate prin punct și virgulă

### 9.3.2 Analiza stack trace-ului

Când o excepție nu este capturată de năicăieri, JVM-ul afișează un *stack trace* — lanțul complet de apeluri de metode active la momentul aruncării excepției — și oprește programul:

**Listing 9.10:** Un stack trace tipic, cu excepție înlănțuită

```

1 Exception in thread "main" java.lang.RuntimeException:
   Tranzactia a esuat
2   at Banca.proceseazaTranzactie(Banca.java:42)
3   at Banca.main(Banca.java:15)
4 Caused by: FonduriInsuficienteException: Fonduri insuficiente
5   at ContBancar.retrage(ContBancar.java:28)
6   at Banca.proceseazaTranzactie(Banca.java:40)
7   ... 1 more

```

#### Notă

Se citește de sus în jos: prima linie identifică tipul excepției și mesajul ei; liniile `at ...` arată lanțul de apeluri, de la locul aruncării (sus) către punctul de intrare al programului (jos). Secțiunea `Caused by:` apare exact datorită mecanismului de *chaining* din Subsecțiunea 9.2.3 — fără el, am vedea doar primul bloc, pierzând informația despre cauza reală (`FonduriInsuficienteException`). Codul propriu poate obține aceleași informații programatic, prin `e.printStackTrace()` sau prin array-ul returnat de `e.getStackTrace()`.

### 9.3.3 Bune practici

#### Notă

Câteva capcane frecvente, de evitat sistematic:

- ▶ Nu „înghiți” tăcut o excepție (`catch (Exception e) { },` bloc `gol`) — programul continuă ca și cum nimic nu s-ar fi întâmplat, iar eroarea reală devine aproape imposibil de diagnosticat ulterior. Cel puțin logați excepția (Subsecțiunea 9.4.2).
- ▶ Nu capturați **Exception** (sau, și mai rău, **Throwable**) generic, decât dacă chiar intenționați să tratați orice eroare posibilă la fel — de obicei ascunde erori de programare (precum `NullPointerException`) sub același tratament ca eșecuri așteptate (precum `FonduriInsuficienteException`).
- ▶ Nu folosiți excepții pentru fluxul normal de control al programului — aruncarea și capturarea unei excepții este semnificativ mai costisitoare decât o structură condițională obișnuită, iar codul devine mai greu de urmărit.
- ▶ Păstrați mesajele de excepție informative, incluzând valorile relevante (de exemplu suma și soldul disponibil, nu doar „eroare la retragere”) — mesajul e adesea singurul indiciu disponibil la depanare.

## 9.4 Aserții și Logging

### 9.4.1 Aserții

O *asertie* verifică o condiție despre care programatorul crede că este întotdeauna adevărată, dacă programul este corect — nu o condiție care

poate eșua din cauza datelor de intrare ale utilizatorului:

```

1 public double calculeazaMedie(double[] valori) {
2     assert valori.length > 0 : "Array-ul nu ar trebui sa fie
   gol aici";
3
4     double suma = 0;
5     for (double v : valori) {
6         suma += v;
7     }
8     return suma / valori.length;
9 }

```

#### Notă

Aserțiile sunt **dezactivate implicit** la execuție — trebuie activate explicit cu opțiunea *-ea* (*enable assertions*) a comenzii `java`. Din acest motiv, o aserție **nu** trebuie folosită pentru validarea datelor primite de la utilizator sau din surse externe (folosiți excepții, ca `IllegalArgumentException`, din Subsecțiunea 9.1.4, pentru acele cazuri) — ci doar pentru verificarea unor ipoteze interne, pe care codul propriu ar trebui să le garanteze deja prin construcție. Un eșec de aserție indică, de regulă, o eroare de programare, nu o intrare invalidă.

**Listing 9.11:** O aserție, pentru o invari-  
antă internă a programului

## 9.4.2 API-ul de logging

Instrucțiunile `System.out.println` folosite pe tot parcursul acestui curs sunt potrivite pentru exemple didactice, dar nepotrivite pentru o aplicație reală: nu pot fi dezactivate selectiv, nu au nivel de severitate, nu pot fi redirectionate ușor către un fișier. API-ul de logging al platformei rezolvă aceste limitări:

```

1 import java.util.logging.Logger;
2 import java.util.logging.Level;
3
4 public class ContBancar {
5     private static final Logger LOG = Logger.getLogger(
6         ContBancar.class.getName());
7
8     public void retrage(double suma) throws
9         FonduriInsuficienteException {
10         LOG.fine("Se incearca retragerea a " + suma + " lei")
11         ;
12
13         if (suma > sold) {
14             LOG.warning("Fonduri insuficiente pentru
15             retragere: " + suma + " > " + sold);
16             throw new FonduriInsuficienteException("Fonduri
17             insuficiente", suma - sold);
18         }
19
20         sold = sold - suma;
21         LOG.info("Retragere reusita: " + suma + " lei");
22     }
23 }

```

**Listing 9.12:** Un logger de bază, cu diver-  
se niveluri de severitate

Tabela 9.1: Nivelurile de severitate ale API-ului de logging, de la cel mai important la cel mai detaliat.

| Nivel             | Utilizare tipică                                                   |
|-------------------|--------------------------------------------------------------------|
| SEVERE            | erori grave, care afectează funcționarea aplicației                |
| WARNING           | situații neobișnuite, dar din care programul se recuperează        |
| INFO              | evenimente importante ale fluxului normal (mesaje pentru operator) |
| CONFIG            | informații de configurare                                          |
| FINE/FINER/FINEST | detalii progresiv mai fine, utile la depanare                      |

### 9.4.3 Configurare: handlers, filtre, formatters

Un Logger nu scrie direct mesajele — le transmite unuia sau mai multor Handler-e, fiecare responsabil de o destinație (consolă, fișier) și, opțional, de un format specific:

Listing 9.13: Configurarea unui logger cu handler de fișier și nivel minim explicit

```

1 import java.util.logging.FileHandler;
2 import java.util.logging.SimpleFormatter;
3
4 Logger log = Logger.getLogger("ro.unitbv.mip.banking");
5 log.setLevel(Level.FINE); // accepta mesaje de la FINE in sus
6
7 FileHandler fisierLog = new FileHandler("aplicatie.log");
8 fisierLog.setFormatter(new SimpleFormatter());
9 log.addHandler(fisierLog);

```

#### Notă

Nivelul setat pe Logger (setLevel) acționează ca un prim filtru: mesajele sub acel nivel sunt ignorate imediat, fără a ajunge la niciun handler. Fiecare Handler poate avea, la rândul lui, propriul nivel și propriul Formatter — de exemplu, un ConsoleHandler la nivel WARNING (doar problemele vizibile imediat pe ecran) combinat cu un FileHandler la nivel FINE (jurnal complet, păstrat pentru investigații ulterioare). Un Filter personalizat poate adăuga criterii suplimentare, dincolo de simplul nivel de severitate, dar rareori este necesar în aplicații obișnuite.

### 9.4.4 O rețetă practică de logging

Tiparul folosit consecvent în cod Java real, pentru capturarea și logarea unei excepții, fără a o „înghiți” tăcut (Subsecțiunea 9.3.3):

Listing 9.14: Tiparul recomandat: logare cu stack trace complet, apoi decizie explicită

```

1 try {
2     cont.retrage(suma);
3 } catch (FonduriInsuficienteException e) {
4     LOG.log(Level.WARNING, "Retragere esuata pentru " + cont.
5         getTitular(), e);
6     // decizie explicita: propagam mai departe, afisam
7     // un mesaj, sau incercam o alternativa
8     throw new RuntimeException("Operatiune bancara esuata", e);
9 }

```

#### Notă

Metoda log(Level, mesaj, exceptie), cu trei argumente, înregistrează atât mesajul descriptiv, cât și **întregul** stack trace al excepției

(inclusiv orice cauză înlănțuită, Subsecțiunea 9.2.3) — spre deosebire de `LOG.warning(mesaj)`, care ar înregistra doar textul, pierzând complet contextul tehnic al erorii.

### 9.4.5 Tehnici de debugging

Pe lângă logging, IDE-urile moderne (Secțiunea 1.4) oferă un depanator (*debugger*) integrat: puncte de întrerupere (*breakpoints*), care opresc execuția la o linie anume, inspecție a valorilor variabilelor în acel moment, execuție pas cu pas (*step over/step into*), și *breakpoint-uri condiționate* (active doar când o expresie dată este adevărată).

#### Notă

Regula practică pentru alegerea instrumentului potrivit: folosiți depanatorul IDE-ului pentru investigarea interactivă, în timp real, a unei erori reproductibile local; folosiți logging-ul pentru înregistrarea persistentă a evenimentelor dintr-un mediu unde nu puteți atașa un depanator (de exemplu, o aplicație deja în producție); evitați `System.out.println` presărat temporar prin cod ca metodă principală de depanare — e ușor de uitat în cod și oferă mult mai puțin control decât oricare dintre celelalte două opțiuni.

#### Rezumat

- ▶ Ierarhia excepțiilor pornește de la `Throwable`, cu ramurile `Error` (necapturabile în practică) și `Exception`; excepțiile **verificate** (subclase ale `Exception`, dar nu ale `RuntimeException`) trebuie capturate sau declarate cu `throws`, spre deosebire de cele **neverificate**.
- ▶ `try/catch/finally` capturează și tratează excepții; `finally` se execută necondiționat; `multi-catch` (`catch (A | B e)`) unifică tratamentul mai multor tipuri înrudite.
- ▶ *Chaining*-ul (constructorul cu cauza) păstrează contextul complet al unei erori atunci când o excepție este „învelită” într-una nouă, mai relevantă contextual.
- ▶ `try-with-resources` garantează închiderea automată a resursei `AutoCloseable`, în ordine inversă declarării, eliminând nevoia unui `finally` scris manual.
- ▶ Nu „înghițiți” excepții tăcut și nu capturați `Exception` generic fără motiv — ambele ascund erori reale.
- ▶ Aserțiile (`assert`) verifică invariante interne, dezactivate implicit la execuție — nu înlocuiesc validarea datelor de intrare cu excepții.
- ▶ API-ul de logging (`Logger`, niveluri de severitate, `Handler`, `Formatter`) înlocuiește `System.out.println` în cod de producție, oferind control fin asupra destinației și volumului de informație înregistrată.

#### Exerciții

1. Rescrieți metoda `contCuDepunereInitiala` din Subsecțiunea 4.4.5 astfel încât să arunce o excepție proprie `SumaInvalidaException` (verificată), în loc de `IllegalArgumentException`, și actualizați codul apelant corespunzător.

2. Scrieți un mic program care citește un fișier text linie cu linie folosind `try-with-resources`, tratând corect cazul în care fișierul nu există.
3. Construiți un lanț de trei excepții (*chaining*) și afișați stack trace-ul complet cu `printStackTrace()`, identificând în el toate secțiunile `Caused by:`.
4. Adăugați un logger la clasa `ContBancar`, cu mesaje la nivel `INFO` pentru fiecare depunere reușită și `WARNING` pentru fiecare retragere eșuată din lipsă de fonduri.
5. Discutați: de ce credeți că aserțiile sunt dezactivate implicit la execuție, spre deosebire de excepții, care sunt întotdeauna active?

# **Programare generică, colecții și concurență**



## Introducere

Am folosit deja `ArrayList<ContBancar>` încă din Subsecțiunea 6.4.1, promițând atunci o explicație riguroasă mai târziu — acesta este momentul. *Programarea generică* permite scrierea de clase și metode care lucrează cu **orice** tip, specificat de codul care le folosește, păstrând totodată siguranța de tip verificată la compilare. Vom vedea de ce genericele există, cum se scriu clase și metode generice proprii, și câteva reguli — uneori surprinzătoare — care decurg din modul în care Java le implementează intern.

## 10.1 Clase și metode generice

### 10.1.1 Motivație: de ce generice?

Înainte de generice (introduse în Java 5), o clasă container care trebuia să funcționeze cu orice tip folosea `Object`, cu cast-uri explicite necesare la fiecare utilizare:

```
1 public class CutieVeche {
2     private Object continut;
3
4     public void seteaza(Object continut) {
5         this.continut = continut;
6     }
7
8     public Object obtine() {
9         return continut;
10    }
11 }
12
13 CutieVeche cutie = new CutieVeche();
14 cutie.seteaza("un text");
15
16 String text = (String) cutie.obtine(); // cast EXPLICIT,
17                                     // necesar, risc de ClassCastException
```

**Listing 10.1:** Stilul dinaintea genericeilor  
– fără siguranță de tip la compilare

## Notă

Problema acestui cod nu este doar verbozitatea cast-ului — este că **nimic** nu împiedică pe cineva să scrie `cutie.seteaza(42)` (un `Integer`, nu un `String`), urmat de cast-ul greșit `(String)` `cutie.obtine()`, care ar arunca `ClassCastException` abia la execuție (Subsecțiunea 6.2.2), deși eroarea de logică era prezentă încă din momentul scrierii codului. Genericele mută exact această verificare de la execuție la compilare.

### 10.1.2 O clasă generică proprie

Rescriem clasa de mai sus, folosind un *parametru de tip* — convențional numit `T` (de la *Type*), deși orice identificator e valid:

**Listing 10.2:** O clasă generică, cu parametru de tip T

```

1 public class Cutie<T> {
2     private T continut;
3
4     public void seteaza(T continut) {
5         this.continut = continut;
6     }
7
8     public T obtine() {
9         return continut;
10    }
11 }

```

**Listing 10.3:** Folosirea clasei generice, fără niciun cast explicit

```

1 Cutie<String> cutie = new Cutie<>();
2 cutie.seteaza("un text");
3
4 String text = cutie.obtine(); // FARA cast - compilatorul
5 // stie deja ca e String
6
7 cutie.seteaza(42); // EROARE DE COMPILARE - Cutie<String>
8 // accepta doar String

```

**Notă**

Acesta este exact mecanismul din spatele lui `ArrayList<ContBancar>`, folosit deja din Subsecțiunea 6.4.1: `ArrayList<T>` este, la rândul ei, o clasă generică din biblioteca standard, definită după exact același tipar ca `Cutie<T>` de mai sus. Notăția `<>` de la instanțiere (*diamond operator*) permite compilatorului să deducă automat parametrul de tip din contextul din stânga atribuirii, fără a-l repeta.

**10.1.3 O metodă generică**

O metodă poate fi generică **independent** de clasa care o conține — parametrul de tip este declarat explicit, între modificatorii metodei și tipul returnat:

**Listing 10.4:** O metodă statică generică, care funcționează cu orice tip de array

```

1 public static <T> T primul(T[] array) {
2     if (array.length == 0) {
3         throw new IllegalArgumentException("array este gol");
4     }
5     return array[0];
6 }

```

**Listing 10.5:** Apelarea unei metode generice – tipul e dedus din argument

```

1 String[] nume = { "Ana", "Ion", "Maria" };
2 String primulNume = primul(nume); // T este dedus ca String
3
4 ContBancar[] conturi = { new ContBancar("Ana Popa", 500.0) };
5 ContBancar primulCont = primul(conturi); // T este dedus ca
6 // ContBancar

```

**Notă**

Declarația `<T>` dinaintea tipului returnat (`public static <T> T primul(...)`) este ceea ce marchează metoda ca generică — fără ea, compilatorul ar căuta o clasă numită literal `T`, inexistentă, și ar semnaliza eroare. Această formă este independentă de eventualii parametri de

tip al clasei care conține metoda; o metodă static generică, în particular, **trebuie** să-și declare propriul parametru de tip, neputând folosi parametrul de tip al clasei (care nu există încă, pentru membri statici, la fel cum am discutat în Subsecțiunea 4.4.5).

## 10.2 Bounds (limite) pentru parametri de tip

### 10.2.1 Bounds simple

Uneori avem nevoie să apelăm metode specifice pe valorile de tip  $T$  — de exemplu, să le comparăm. Un *bound* restrânge tipurile acceptabile pentru  $T$  la cele care respectă o anumită interfață sau extind o anumită clasă:

```
1 public static <T extends Comparable<T>> T maximDintre(T a, T
   b) {
2     return a.compareTo(b) >= 0 ? a : b;
3 }
```

**Listing 10.6:** O metodă generică cu bound:  $T$  trebuie să fie Comparable

```
1 int maxim = maximDintre(10, 20); // Integer implementeaza
   // Comparable<Integer>
2 String maximText = maximDintre("ana", "bob");
3 // String implementeaza Comparable<String>
```

**Listing 10.7:** Folosirea metodei cu bound, cu tipuri care implementează Comparable

#### Notă

Cuvântul cheie folosit este `extends`, chiar și atunci când limita este o interfață (ca `Comparable<T>` de mai sus), nu o clasă — spre deosebire de `implements`, folosit la declararea unei clase (Subsecțiunea 7.1.1). Fără acest bound, expresia `a.compareTo(b)` nu ar compila deloc: compilatorul nu ar avea nicio garanție că un  $T$  oarecare are metoda `compareTo`, întrucât aceasta nu există pe `Object`.

### 10.2.2 Bounds multiple

Un parametru de tip poate avea **mai multe** limite simultan, separate prin `&` — cel mult una poate fi o clasă (și trebuie să fie prima), restul trebuie să fie interfețe:

```
1 public static <T extends Comparable<T> & java.io.Serializable
   > T maximSerializabil(T a, T b) {
2     return a.compareTo(b) >= 0 ? a : b;
3 }
```

**Listing 10.8:** Un bound multiplu:  $T$  trebuie să fie și Comparable, și Serializable

#### Notă

Bounds-urile multiple sunt relativ rar necesare în cod obișnuit de aplicație — apar mai ales în biblioteci generice care trebuie să garanteze simultan mai multe capacități ale tipului parametru (de exemplu, atât ordonare, cât și capacitatea de a fi serializat pe disc sau prin rețea). Este suficient să recunoașteți sintaxa dacă o întâlniți în documentația unei biblioteci.

## 10.3 Type erasure

### 10.3.1 Cum funcționează erasure

Genericile Java sunt o construcție a **compilatorului**, nu a mașinii virtuale: la compilare, toată informația despre parametrii de tip este *ștearsă* (*erased*), înlocuită cu tipul de bound (sau `Object`, dacă nu există bound), iar cast-urile necesare sunt inserate automat de compilator.

**Listing 10.9:** La runtime, toate instanțele `Cutie<T>` au aceeași clasă

```
1 Cutie<String> c1 = new Cutie<>();
2 Cutie<Integer> c2 = new Cutie<>();
3
4 System.out.println(c1.getClass() == c2.getClass());
5 // true - ACEEASI clasa la runtime!
6 System.out.println(c1.getClass().getName());
7 // "Cutie", fara nicio mentiune de String
```

#### Notă

Această comportare poate fi surprinzătoare la prima întâlnire: deși `Cutie<String>` și `Cutie<Integer>` par tipuri diferite la compilare, la execuție există o **singură** clasă `Cutie`, iar parametrul de tip nu mai există în niciun fel în bytecode. Numele „erasure” (*ștergere*) descrie exact acest proces: informația de tip generic ajută compilatorul să verifice corectitudinea codului, apoi dispare complet.

### 10.3.2 Consecințe practice ale erasure

Câteva restricții ale genericilor Java decurg direct din erasure, nu sunt limitări arbitrare:

**Listing 10.10:** Restricții impuse de type erasure

```
1 public class Cutie<T> {
2     // private T[] elemente = new T[10];
3     // EROARE - nu se pot crea array-uri generice
4
5     // private static T ultimaValoare;
6     // EROARE - T nu exista la nivel de clasa (static)
7
8     public boolean verifica(Object obj) {
9         // return obj instanceof Cutie<String>;
10        // EROARE - tipul parametrizat nu exista la runtime
11        return obj instanceof Cutie<?>;
12        // OK - doar tipul BRUT poate fi verificat
13    }
14 }
```

#### Notă

Toate cele trei restricții au aceeași cauză: la execuție, informația `<String>/<Integer>` pur și simplu nu mai există, ștearsă la compilare. Un array `T[]` ar avea nevoie de acea informație pentru a-și crea corect elementele; un câmp static ar fi partajat între **toate** instanțele clasei (`Cutie<String>`, `Cutie<Integer>` etc., care sunt, la runtime, aceeași clasă); un `instanceof` cu tip parametrizat ar necesita o verificare pe care JVM-ul nu o mai poate face. Amintim și restricția deja menționată în Subsecțiunea 6.4.2: un parametru de tip nu poate fi niciodată un tip primitiv, exact din același motiv — clasele wrapper

(Integer, Double etc.) există parțial pentru a face genericele posibile.

### 10.3.3 Reguli de moștenire pentru tipuri generice

O intuiție greșită, frecventă la începători: dacă Integer este subtip de Number (prin autoboxing și ierarhia claselor wrapper), atunci Cutie<Integer> ar trebui să fie subtip de Cutie<Number>. Java respinge explicit această presupunere:

```
1 Cutie<Integer> cutieIntregi = new Cutie<>();
2 cutieIntregi.seteaza(42);
3
4 Cutie<Number> cutieNumere = cutieIntregi;
5 // EROARE DE COMPILARE!
```

**Listing 10.11:** Genericele Java sunt *invariante* – Cutie<Integer> NU e subtip de Cutie<Number>

#### Notă

Această restricție, numită *invariantă*, nu este o limitare arbitrară, ci o protecție necesară. Dacă atribuirea de mai sus ar fi permisă, codul următor ar compila, dar ar corupe silențios tipul conținutului: cutieNumere.seteaza(3.14); (un Double, perfect valid pentru Cutie<Number>) ar insera de fapt un Double într-o cutie pe care cutieIntregi o crede Cutie<Integer>, provocând un ClassCastException mult mai târziu, la un obtine() complet nevinovat. Secțiunea următoare arată cum wildcard-urile permit totuși flexibilitate controlată, fără a compromite această siguranță.

## 10.4 Wildcard-uri

### 10.4.1 Wildcard nemărginit (?)

Wildcard-ul ? reprezintă „un tip oarecare, necunoscut” — util atunci când o metodă nu are nevoie să cunoască exact parametrul de tip, ci doar să lucreze cu Cutie de orice fel:

```
1 public static void afiseaza(Cutie<?> cutie) {
2     System.out.println("Conținut: " + cutie.obține());
3     // obține() returnează Object, dar poate fi afisat
4 }
5
6 afiseaza(new Cutie<String>()); // OK
7 afiseaza(new Cutie<Integer>()); // OK, aceeași metodă
```

**Listing 10.12:** Cutie<?> acceptă orice fel de cutie, indiferent de conținut

#### Notă

Spre deosebire de Cutie<Object> (o cutie care conține specific obiecte de tip Object), Cutie<?> acceptă o cutie cu **orice** parametru de tip concret — inclusiv Cutie<String> sau Cutie<Integer>, care **nu** sunt, conform invarianței din Subsecțiunea 10.3.3, subtipuri de Cutie<Object>.

### 10.4.2 Wildcard mărginit superior (? extends)

Atunci când o metodă doar citește valori dintr-o structură generică, un wildcard mărginit superior permite acceptarea oricărei substituții compatibile:

**Listing 10.13:** ? extends Number – putem citi, tratând orice element ca Number

```

1 public static double sumaConturi(List<? extends Number>
    valori) {
2     double suma = 0;
3     for (Number n : valori) {
4         // fiecare element e GARANTAT cel puțin Number
5         suma += n.doubleValue();
6     }
7     return suma;
8 }
9
10 sumaConturi(List.of(100, 200, 300)); // List<Integer> - OK
11 sumaConturi(List.of(19.99, 5.50)); // List<Double> - OK

```

#### Notă

Această formă rezolvă exact problema din Subsecțiunea 10.3.3: `List<? extends Number>` acceptă atât `List<Integer>`, cât și `List<Double>`, fiindcă metoda doar citește elementele (garantat cel puțin `Number`), fără să adauge vreunul nou. Compilatorul interzice explicit adăugarea de elemente într-o listă `? extends` (cu excepția lui `null`) – nu ar putea garanta tipul exact așteptat de lista reală, primită de la apelant.

### 10.4.3 Wildcard mărginit inferior (? super)

Simetric, atunci când o metodă doar scrie (adaugă) valori într-o structură generică, un wildcard mărginit inferior permite acceptarea oricărei destinații suficient de „largi”:

**Listing 10.14:** ? super Integer – putem adăuga `Integer`, în orice listă suficient de generală

```

1 public static void adaugaPrimele(List<? super Integer> lista,
    int n) {
2     for (int i = 1; i <= n; i++) {
3         lista.add(i); // OK - orice lista din ierarhia
4                         // "super Integer" poate primi
5                         // un Integer
6     }
7 }
8
9 List<Number> numere = new ArrayList<>();
10 adaugaPrimele(numere, 5); // List<Number> accepta - Number
11                          // e "super" fata de Integer
12
13 List<Object> obiecte = new ArrayList<>();
14 adaugaPrimele(obiecte, 5); // List<Object> accepta la fel

```

#### Definiție — Principiul PECS

Regula mnemonică folosită frecvent pentru alegerea corectă a wildcard-ului este *PECS*: *Producer Extends, Consumer Super*. Dacă structura produce (oferă) valori pe care le citești, folosiți `? extends`; dacă structura consumă (primește) valori pe care le adăugați, folosiți `? super`.

### 10.4.4 Capturarea wildcard

Uneori o metodă are nevoie să facă **ambele** lucruri — să citească și să rescrie un element — caz în care niciun wildcard simplu nu ar compila. Soluția este o metodă auxiliară privată, generică, care *capturează* tipul necunoscut într-un parametru de tip concret:

```

1 public static void inverseazaPrimele(List<?> lista) {
2     inverseazaAjutor(lista); // deleaga catre metoda
3                               // generica, care CAPTUREAZA
4                               // tipul
5 }
6
7 private static <T> void inverseazaAjutor(List<T> lista) {
8     if (lista.size() < 2) {
9         return;
10    }
11    T primul = lista.get(0);
12    T ultimul = lista.get(lista.size() - 1);
13    lista.set(0, ultimul); // acum e permis - T e un tip
14                          // CONCRET, capturat
15    lista.set(lista.size() - 1, primul);
16 }

```

**Listing 10.15:** Capturarea wildcard, printr-o metodă auxiliară generică

#### Notă

Metoda publică, `inverseazaPrimele(List<?> lista)`, are o semnătură flexibilă, care acceptă orice listă. Corpul ei nu poate face direct `lista.set(0, lista.get(1))` (compilatorul nu poate garanta că elementul citit e compatibil cu `?` necunoscut), dar delegarea către `inverseazaAjutor`, o metodă generică obișnuită (Subsecțiunea 10.1.3), „capturează” tipul necunoscut `?` într-un parametru de tip concret `T` pentru durata aceluși apel — suficient pentru ca operațiile de citire și scriere să fie, din nou, verificate consecvent de compilator.

## 10.5 Restricții și limitări; generice și reflecție

### 10.5.1 Alte restricții

Câteva limitări suplimentare, tot consecințe ale type erasure (Secțiunea 10.3), merită menționate explicit, fiindcă generează frecvent confuzie la prima întâlnire:

```

1 public class Cutie<T> {
2     // private T continut = new T(); // EROARE - nu se poate
3                                     // instantia direct un
4                                     // parametru de tip
5
6     public void proceseaza() {
7         try {
8             // ...
9         } catch (T e) { // EROARE - nu se poate captura o
10            exceptie de tip generic
11                // ...
12        }
13    }
14
15    // Ambele metode de mai jos au ACEEASI semnatura dupa

```

**Listing 10.16:** Restricții suplimentare, toate cauzate de erasure

```

15 // erasure -> EROARE DE COMPILARE
16 // public void adauga(List<String> valori)
17 // public void adauga(List<Integer> valori)
18 }

```

#### Notă

Imposibilitatea de a scrie `new T()` are o soluție uzuală: metoda sau constructorul primește, în plus, un `Supplier<T>` (Subsecțiunea 7.2.3) sau un obiect `Class<T>`, folosit apoi pentru a crea instanțe prin reflecție (Subsecțiunea 8.4.2) — exact tehnica pe care am folosit-o deja acolo, deși fără acest context explicit. Ultima restricție (suprîncărcarea cu tipuri generice diferite) este directă: după erasure, ambele metode `adauga(List)` ar avea identic aceeași semnătură, ceea ce compilatorul respinge ca redeclarare a aceleiași metode.

## 10.5.2 Generice și reflecție

Deși un obiect la runtime nu „știe” cu ce parametru de tip a fost instanțiat (Subsecțiunea 10.3.1), reflecția (Secțiunea 8.3) poate totuși inspecta parametrii de tip **declarați** ai unui câmp sau ai unei metode — această informație este păstrată în fișierul `.class`, separat de tipul propriu-zis al obiectelor la execuție:

Listing 10.17: Inspectarea tipului generic declarat al unui câmp, prin reflecție

```

1 import java.lang.reflect.Field;
2 import java.lang.reflect.ParameterizedType;
3 import java.lang.reflect.Type;
4
5 Field campIstoric = ContBancar.class.getDeclaredField("
    istoricTranzactii");
6 Type tipGeneric = campIstoric.getGenericType();
7
8 if (tipGeneric instanceof ParameterizedType pt) {
9     Type[] argumenteTip = pt.getActualTypeArguments();
10    System.out.println(argumenteTip[0]);
11    // class ContBancar$Extras
12 }

```

#### Notă

Diferența față de Subsecțiunea 10.3.1 este subtilă, dar importantă: câmpul `istoricTranzactii` (definit ca `List<Extras>` în Subsecțiunea 7.4.4) are un tip **declarat** cunoscut, păstrat static în structura clasei — ceea ce erasure elimină este informația despre parametrul de tip al unei **instanțe concrete** de `List` la execuție, nu despre declarația însăși. Această distincție este exact ce fac posibile bibliotecile de serializare JSON/XML sau cele de injecție de dependențe (menționate deja în Subsecțiunea 8.4.3): ele inspectează, prin reflecție, tipul generic **declarat** al unui câmp, pentru a ști ce tip de obiecte să creeze la reconstituirea datelor.

#### Rezumat

- O clasă sau metodă generică lucrează cu un parametru de tip (`T`), verificat la compilare, eliminând nevoia cast-urilor explicite și a riscului de `ClassCastException` la execuție.
- Un *bound* (`T extends ...`) restrânge parametrul de tip la cele

care oferă anumite metode/interfețe; `extends` se folosește chiar și pentru bounds de tip interfață.

- ▶ *Type erasure* șterge, la compilare, toată informația despre parametrii de tip din bytecode — de aici decurg restricțiile privind array-urile generice, câmpurile statice generice, `instanceof` pe tipuri parametrizate, și `new T()`.
- ▶ Genericele Java sunt **invariante**: `Cutie<Integer>` nu este subtip de `Cutie<Number>`, chiar dacă `Integer` este subtip de `Number` — o protecție necesară împotriva corupției silențioase de tip.
- ▶ Wildcard-urile (`?`, `? extends`, `? super`) redau flexibilitate controlată; principiul PECS (*Producer Extends, Consumer Super*) ghidează alegerea corectă. Capturarea wildcard, printr-o metodă auxiliară generică, rezolvă cazurile care necesită simultan citire și scriere.
- ▶ Reflecția poate inspecta parametrii de tip **declarați** (prin `ParameterizedType`), deși nu poate recupera parametrul de tip al unei instanțe concrete la execuție.

### Exerciții

1. Scrieți o clasă generică `Pereche<A, B>`, cu două componente de tipuri diferite, și metode accesori pentru fiecare.
2. Scrieți o metodă generică statică `<T extends Comparable<T>> T minimDinArray(T[] valori)`, care returnează cel mai mic element al unui array.
3. Explicați, cu un exemplu de cod, de ce `List<Cutie<Integer>>` și `List<Cutie<Number>>` sunt tipuri incompatibile, chiar dacă am accepta (ipotetic) ca `Cutie<Integer>` și `Cutie<Number>` să fie compatibile.
4. Scrieți o metodă statică `double sumaVenituri(List<? extends Number> venituri)` și o metodă statică `void adaugaBonusuri(List<? super Integer> lista, int... bonusuri)`, aplicând corect principiul PECS pentru fiecare.
5. Folosind reflecție, inspectați tipul generic declarat al unui câmp propriu de tip `List<String>` dintr-o clasă la alegere, și afișați argumentul de tip extras din `ParameterizedType`.



## Introducere

Am folosit deja `ArrayList` încă din Subsecțiunea 6.4.1, iar genericile din Capitolul 10 ne dau acum instrumentele necesare pentru a înțelege riguros `ArrayList<T>` și rudele sale. Java oferă un întreg *framework de colecții* — un set coerent de interfețe și implementări pentru grupuri de obiecte: liste ordonate, mulțimi fără duplicate, cozi cu disciplină de acces specifică. În acest capitol și în următorul parcurgem acest framework sistematic, pornind de la interfețele care îl organizează.

## 11.1 Framework-ul de colecții

### 11.1.1 Ierarhia interfețelor

Framework-ul de colecții separă strict **interfețele** (ce poate face o colecție) de **implementările** (cum anume face acel lucru) — exact principiul de proiectare orientat pe interfețe din Secțiunea 7.1, aplicat sistematic:

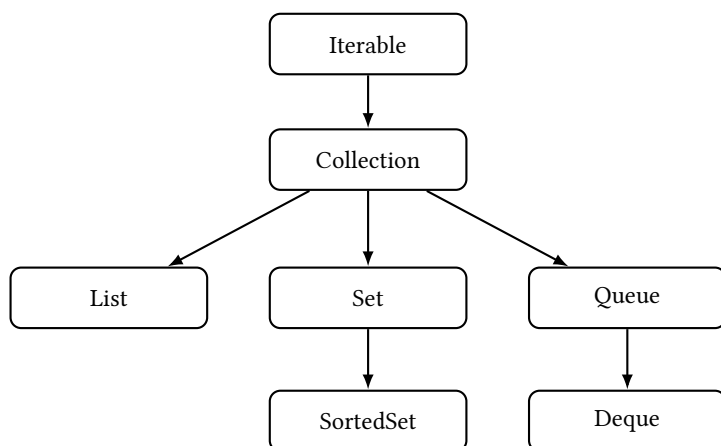


Figura 11.1: Ierarhia (simplificată) a interfețelor din framework-ul de colecții.

## Notă

Codul scris împotriva interfeței `List`, de exemplu, funcționează neschimbat indiferent dacă la execuție primește un `ArrayList` sau un `LinkedList` (Secțiunea 11.2) — exact polimorfismul discutat în Subsecțiunea 6.2.1, aplicat colecțiilor. Declarați variabilele și parametrii de metodă cu tipul de **interfață** (`List<T> lista = new ArrayList<>();`), nu cu tipul concret de implementare, pentru a păstra libertatea de a schimba implementarea mai târziu, fără a afecta restul codului.

### 11.1.2 Iteratori

Interfața `Iterable` (vârful ierarhiei din Figura 11.1) este ceea ce face posibilă bucla `for - each`, folosită deja din Subsecțiunea 2.6.2 pentru array-uri — orice colecție o poate folosi la fel de natural:

**Listing 11.1:** for-each pe o colecție, posibil datorită interfeței Iterable

```
1 List<ContBancar> conturi = new ArrayList<>();
2 conturi.add(new ContBancar("Ana Popescu", 500.0));
3
4 for (ContBancar cont : conturi) {
5     // echivalent cu folosirea explicita a unui Iterator
6     System.out.println(cont);
7 }
```

**Listing 11.2:** Aceeași iterare, scrisă explicit cu un Iterator

```
1 Iterator<ContBancar> it = conturi.iterator();
2 while (it.hasNext()) {
3     ContBancar cont = it.next();
4     System.out.println(cont);
5 }
```

#### Notă

Forma for-each este, la compilare, transformată exact în forma explicită de mai jos — o comoditate sintactică (asemănătoare, ca idee, cu referințele la metode din Subsecțiunea 7.3.1), nu un mecanism separat. Folosiți totuși Iterator explicit atunci când aveți nevoie să **eliminați** elemente în timpul parcurgerii.

**Listing 11.3:** Eliminarea sigură a elementelor, prin metoda remove a iteratorului

```
1 Iterator<ContBancar> it = conturi.iterator();
2 while (it.hasNext()) {
3     ContBancar cont = it.next();
4     if (cont.getSold() < 100.0) {
5         it.remove(); // SIGUR - elimina elementul curent
6                     // din colectie
7     }
8 }
```

#### Notă

Modificarea unei colecții **direct** (`conturi.remove(cont)`) în timp ce o parcurgeți cu for-each aruncă `ConcurrentModificationException` — colecția detectează că a fost modificată „pe la spate” față de iteratorul activ și refuză să continue într-o stare inconsistentă. Metoda `remove()` a iteratorului însuși este singura cale sigură de a elimina elemente în timpul parcurgerii; vom relua acest subiect, în contextul concurenței multi-thread, în Capitolul 13.

## 11.2 Liste

### 11.2.1 ArrayList, revizitat

Am folosit `ArrayList` extensiv, din Subsecțiunea 6.4.1 încoace. Intern, un `ArrayList` este susținut de un array care se realocă și copiază automat atunci când se umple — de aici derivă caracteristicile sale de performanță:

#### Notă

„Amortizat” la `add` înseamnă: majoritatea apelurilor sunt  $O(1)$ , dar ocazional, când array-ul intern se umple, are loc o realocare și copiere completă,  $O(n)$  — însă suficient de rară încât costul mediu, pe termen

| Operație                         | Complexitate                             |
|----------------------------------|------------------------------------------|
| <code>get(index)</code>          | $O(1)$ – acces direct, ca la un array    |
| <code>add(element)</code>        | $O(1)$ amortizat                         |
| <code>add(index, element)</code> | $O(n)$ – necesită deplasarea elementelor |
| <code>remove(index)</code>       | $O(n)$ – necesită deplasarea elementelor |

**Tabela 11.1:** Complexitatea operațiilor uzuale ale `ArrayList`.

lung, rămâne  $O(1)$ . `ArrayList` este, în marea majoritate a situațiilor practice, alegerea implicită corectă pentru o listă în Java.

### 11.2.2 LinkedList

`LinkedList` implementează atât `List`, cât și `Deque` (Secțiunea 11.4), folosind o listă dublu înlănțuită — structură similară celei construite manual în Subsecțiunea 7.4.1, dar cu legături în ambele direcții:

```

1 List<ContBancar> conturi = new LinkedList<>();
2 conturi.add(new ContBancar("Ana Popescu", 500.0));
3
4 LinkedList<ContBancar> listaLegata = new LinkedList<>();
5 listaLegata.addFirst(new ContBancar("Ion Ionescu", 1200.0));
6 // adaugare la INCEPUT,  $O(1)$ 
7 listaLegata.addLast(new ContBancar("Maria Popa", 800.0));
8 // adaugare la SFARSIT,  $O(1)$ 

```

**Listing 11.4:** `LinkedList`, folosit prin interfața `List`

#### Notă

`LinkedList` oferă `addFirst`/`addLast`/`removeFirst`/`removeLast` în timp constant  $O(1)$ , spre deosebire de `ArrayList`, unde inserarea la început este  $O(n)$ . În schimb, `get(index)` pe o `LinkedList` este  $O(n)$  (trebuie parcursă lista de la un capăt), spre deosebire de accesul direct  $O(1)$  al lui `ArrayList`.

#### Notă

În practică, `ArrayList` este preferat în marea majoritate a cazurilor, chiar și atunci când codul face inserări/eliminări frecvente — structura internă modernă a lui `ArrayList` și localitatea sa bună în memoria cache fac adesea diferența reală mai mică decât ar sugera tabelul de complexități. Alegeți `LinkedList` explicit doar atunci când profilați efectiv o aplicație și constatați că operațiile la capete domină, sau când aveți nevoie de comportamentul specific de `Deque` (Secțiunea 11.4).

## 11.3 Mulțimi

Un `Set` garantează absența duplicatelor — adăugarea unui element deja prezent (conform `equals`) nu are niciun efect.

### 11.3.1 HashSet

`HashSet` folosește o tabelă de dispersie internă, exact mecanismul pentru care am suprascris deja `equals` și `hashCode` în Subsecțiunea 6.3.2:

**Listing 11.5:** HashSet elimină automat duplicatele, folosind equals/hashCode deja suprascrise

```
1 Set<ContBancar> conturi = new HashSet<>();
2
3 conturi.add(new ContBancar("Ana Popescu", 500.0));
4 conturi.add(new ContBancar("Ana Popescu", 500.0));
5 // acelasi titular SI sold
6
7 System.out.println(conturi.size());
8 // 1 - al doilea a fost respins ca duplicat!
```

## Notă

Cele două obiecte adăugate mai sus sunt instanțe **diferite** din perspectiva ==, dar HashSet le tratează ca duplicate fiindcă equals (suprascris în Subsecțiunea 6.3.1) le consideră egale. Acesta este exact motivul pentru care contractul equals–hashCode din Subsecțiunea 6.3.2 este obligatoriu: fără un hashCode coerent cu equals, HashSet nu ar găsi corect elementele deja existente, iar deduplicarea de mai sus ar eșua silențios.

## Notă

HashSet **nu** garantează nicio ordine de iterare — ordinea observată depinde de detalii interne de implementare (distribuția valorilor de hash) și nu trebuie asumată niciodată în cod. Operațiile de bază (add, contains, remove) au complexitate medie  $O(1)$ , semnificativ mai rapidă decât căutarea liniară  $O(n)$  într-o listă.

### 11.3.2 TreeSet

Spre deosebire de `HashSet`, un `TreeSet` păstrează elementele **ordonate** permanent, folosind fie ordinea naturală a elementelor (`Comparable`), fie un `Comparator` explicit, exact distincția introdusă în Subsecțiunea 7.1.4. Implementăm acum ordinea naturală pentru `ContBancar`, promișă atunci:

**Listing 11.6:** ContBancar capătă o ordine naturală, prin Comparable

```
1 public class ContBancar implements Notificabil, Comparable<
2     ContBancar> {
3     // ... campurile si metodele deja cunoscute ...
4
5     @Override
6     public int compareTo(ContBancar altul) {
7         return Double.compare(this.sold, altul.sold);
8         // ordine naturala: dupa sold
9     }
10 }
```

**Listing 11.7:** TreeSet menține automat ordinea naturală

[illegible]

**Notă**

Pentru o ordine diferită, fără a modifica deloc clasa `ContBancar`, un `TreeSet` poate primi un `Comparator` explicit la construcție: `new TreeSet<>(Comparator.comparing(ContBancar::getTitular))` — exact aceeași alegere între `Comparable` (ordine intrinsecă, unică) și `Comparator` (ordine externă, oricâte variante) discutată deja în Subsecțiunea 7.1.4. Operațiile de bază ale unui `TreeSet` au complexitate  $O(\log n)$ , mai lentă decât `HashSet`, dar cu beneficiul ordinii garantate; interfața `NavigableSet` adaugă metode utile precum `first()`, `last()`, `headSet(limita)`, `tailSet(limita)`.

## 11.4 Cozi

### 11.4.1 Queue și Deque

Un `Queue` respectă, implicit, disciplina FIFO (*first-in, first-out*): elementele sunt extrase în ordinea în care au fost adăugate. Un `Deque` (*double-ended queue*) generalizează acest concept, permițând inserare/extragere la ambele capete:

```
1 Queue<String> cozi = new LinkedList<>();
2
3 cozi.offer("prima tranzactie"); // adauga la coada
4 cozi.offer("a doua tranzactie");
5
6 String urmatoarea = cozi.poll(); // extrage SI elimina capul
7                                // cozii - "prima tranzactie"
8 String varf = cozi.peek();      // citeste capul cozii, FARA
9                                // sa il elimine
```

**Listing 11.8:** Queue, cu metodele sale specifice

**Notă**

Metodele `Queue`-ului vin în două variante: `offer/poll/peek` returnează o valoare specială (`false` sau `null`) dacă operația nu poate fi efectuată (coadă plină sau goală); variantele mai vechi, `add/remove/element`, aruncă o excepție în aceleași situații. Preferați `offer/poll/peek` în cod obișnuit — evitați excepții pentru situații care sunt adesea parte firească a fluxului normal (o coadă goală nu este neapărat o eroare de program).

`ArrayDeque`, o alternativă la `LinkedList` pentru implementarea `Deque`, folosește intern un array circular, mai eficient decât o listă înlanțuită pentru majoritatea utilizărilor tipice de stivă sau coadă:

```
1 Deque<String> stiva = new ArrayDeque<>();
2
3 stiva.push("prima operatie");
4 stiva.push("a doua operatie");
5
6 String ultima = stiva.pop(); // "a doua operatie" - ultima
7                             // adaugata, prima extrasa (LIFO)
```

**Listing 11.9:** `ArrayDeque`, folosit ca stivă (LIFO)

### 11.4.2 PriorityQueue

Un `PriorityQueue` nu respectă ordinea de inserare, ci extrage mereu elementul **minim**, conform ordinii naturale (`Comparable`) sau unui `Com-`

**Listing 11.10:** PriorityQueue extrage mereu contul cu soldul minim, indiferent de ordinea adăugării

parator explicit — util pentru procesarea elementelor în ordinea priorității, nu a sosirii:

```
1 PriorityQueue<ContBancar> coadaProcesare = new PriorityQueue
  <>();
2 // foloseste Comparable, din 11.3.2
3
4 coadaProcesare.offer(new ContBancar("Ion Ionescu", 1200.0));
5 coadaProcesare.offer(new ContBancar("Ana Popescu", 500.0));
6 coadaProcesare.offer(new ContBancar("Maria Popa", 800.0));
7
8 while (!coadaProcesare.isEmpty()) {
9     ContBancar urmatorul = coadaProcesare.poll();
10    // mereu cel cu soldul cel mai MIC ramas
11    System.out.println(urmatorul);
12 }
13 // Ordinea afisata: Ana Popescu (500), Maria Popa (800),
14 // Ion Ionescu (1200)
```

#### Notă

O capcană frecventă: **iterarea directă** (cu for-each) peste un PriorityQueue **nu** garantează ordinea sortată — doar succesiunea de apeluri poll() garantează extragerea în ordine crescătoare. Intern, PriorityQueue folosește o structură de tip *heap*, optimizată pentru extragerea rapidă a minimumului ( $O(\log n)$  per operație), nu pentru parcurgere ordonată completă — dacă aveți nevoie și de iterare ordonată, TreeSet (Subsecțiunea 11.3.2) este alegerea potrivită.

## 11.5 Algoritmi pe colecții

Clasa utilitară Collections (nu confundați cu interfața Collection, la singular) oferă algoritmi generici, aplicabili oricărei liste, complementând clasa Arrays deja folosită în Subsecțiunea 2.6.4.

### 11.5.1 Sortare și shuffling

**Listing 11.11:** Sortarea unei liste, cu ordinea naturală sau cu un Comparator explicit

```
1 List<ContBancar> conturi = new ArrayList<>(List.of(
2     new ContBancar("Ion Ionescu", 1200.0),
3     new ContBancar("Ana Popescu", 500.0)
4 ));
5
6 Collections.sort(conturi); // foloseste Comparable - ordinea
7                             // naturala, dupa sold
8 conturi.sort(Comparator.comparing(ContBancar::getTitular));
9 // echivalent modern, cu Comparator explicit
10
11 Collections.shuffle(conturi); // amesteca elementele in
12                               // ordine aleatorie
```

#### Notă

Metoda Collections.sort(lista) și metoda de instanță lista.sort(comparator) (disponibilă direct pe List, din Java 8) sunt complementare: prima folosește ordinea naturală (Comparable, Subsecțiunea 11.3.2), a doua acceptă orice Comparator explicit, la fel ca la TreeSet. Collections.shuffle este util mai ales pentru testa-

re (amestecarea unor date de test) sau pentru aplicații care au nevoie explicit de ordine aleatorie (jocuri de cărți, de exemplu).

### 11.5.2 Căutare binară

La fel ca `Arrays.binarySearch` din Subsecțiunea 2.6.4, `Collections.binarySearch` localizează rapid un element într-o listă **deja sortată**:

```
1 Collections.sort(conturi); // PRECONDITIE obligatorie -
2 // lista trebuie sortata dupa Comparable
3
4 int pozitie = Collections.binarySearch(conturi, new
5     ContBancar("Cont-cautat", 800.0));
```

**Listing 11.12:** Căutare binară pe o listă, cu aceeași convenție de returnare ca la array-uri

#### Notă

Rezultatul respectă aceeași convenție întâlnită deja la `Arrays.binarySearch`: un index nenegativ înseamnă poziția găsită; o valoare negativă indică absența elementului, codificând totodată poziția unde acesta ar **trebui** inserat pentru a păstra ordinea. Aplicarea acestei metode pe o listă **nesortată** nu generează nicio eroare, dar produce un rezultat nedefinit, fără sens — responsabilitatea sortării prealabile revine integral programatorului.

### 11.5.3 Operații bulk

Operațiile *bulk* acționează asupra unei întregi colecții deodată, comparându-o cu o altă colecție sau cu o condiție:

```
1 List<ContBancar> toateConturile = new ArrayList<>(conturi);
2 List<ContBancar> conturiInactive = List.of(conturi.get(0));
3
4 toateConturile.addAll(altaListaDeConturi);
5 // adauga TOATE elementele celeilalte colectii
6 toateConturile.removeAll(conturiInactive);
7 // elimina orice element prezent SI in conturiInactive
8 toateConturile.retainAll(conturiInactive);
9 // pastreaza DOAR elementele prezente si in conturiInactive
10 toateConturile.removeIf(cont -> cont.getSold() < 100.0);
11 // elimina pe baza unei conditii (Predicate)
```

**Listing 11.13:** Operații bulk: `addAll`, `removeAll`, `retainAll`, `removeIf`

#### Notă

`removeIf`, introdus în Java 8, acceptă direct un `Predicate<T>` (Subsecțiunea 7.2.3) și este preferabil unei bucle manuale cu `Iterator.remove()` (Subsecțiunea 11.1.2) pentru acest tipar frecvent — mai concis și mai puțin predispus la erori. `removeAll`/`retainAll` se bazează, la rândul lor, pe `equals` pentru a determina apartenența unui element la cealaltă colecție, revenind încă o dată la importanța contractului `equals`-`hashCode` din Subsecțiunea 6.3.2.

### 11.5.4 Conversie colecții și array-uri

```
1 List<ContBancar> conturi = new ArrayList<>();
2 // ... populare ...
3
```

**Listing 11.14:** Conversie de la `List` la array, și invers

```

4 ContBancar[] array = conturi.toArray(new ContBancar[0]);
5 // forma clasica, sigura
6
7 List<ContBancar> dinArray = Arrays.asList(array);
8 // vizualizare - dimensiune FIXA!
9 List<ContBancar> copieMutabila = new ArrayList<>(Arrays.
10    asList(array));
11 // copie CU ADEVARAT mutabila

```

### Notă

`toArray(new ContBancar[0])` pare, la prima vedere, inefficient (crează un array de dimensiune 0, doar pentru a-i indica metodei tipul dorit) — dar este exact tehnica prin care ocolim restricția din Subsecțiunea 10.3.2, care interzice crearea directă de array-uri generice (`new T[n]`). JVM-ul folosește acest array doar pentru a determina tipul exact al array-ului de rezultat, prin reflecție. `Arrays.asList(array)` returnează o vizualizare de dimensiune fixă peste array-ul original (elementele pot fi modificate prin `set`, dar nu se pot adăuga/elimina) — pentru o listă complet mutabilă, independentă, înfășurați rezultatul într-un `new ArrayList<>(...)`, ca mai sus.

### Rezumat

- ▶ Framework-ul de colecții separă interfețele (`Collection`, `List`, `Set`, `Queue`) de implementările concrete; codul ar trebui scris folosind adesea interfețe, nu implementări.
- ▶ `for-each` este echivalent cu folosirea explicită a unui `Iterator`; eliminarea elementelor în timpul parcurgerii necesită `Iterator.remove()`, altfel se aruncă `ConcurrentModificationException`.
- ▶ `ArrayList` oferă acces  $O(1)$ , dar inserare/ștergere  $O(n)$  la poziții arbitrare; `LinkedList` inversează acest compromis, dar rareori merită în practică.
- ▶ `HashSet` elimină duplicatele folosind `equals/hashCode`, fără ordine garantată; `TreeSet` păstrează o ordine (naturală sau `Comparator`), cu cost logaritmic.
- ▶ `Queue/Deque` respectă discipline FIFO/LIFO explicite (`offer/poll/peek`); `PriorityQueue` extrage mereu minimumul, dar nu garantează ordine la iterare directă.
- ▶ `Collections` oferă algoritmi generici (sortare, shuffling, căutare binară, operații bulk); conversia listă↔array necesită atenție la mutabilitate (`Arrays.asList` produce o vizualizare de dimensiune fixă).

### Exerciții

1. Adăugați cinci obiecte `ContBancar` într-un `HashSet`, inclusiv două cu exact aceleași date (titular și sold), și verificați că mulțimea rezultată conține doar patru elemente.
2. Folosind `TreeSet` cu un `Comparator` explicit (nu ordinea naturală deja implementată), afișați conturile ordonate alfabetic după titular.
3. Implementați o mică simulare de coadă la ghișeu, folosind `Queue<String>` (nume de clienți), cu operațiile `offer` la sosire și `poll` la deservire.
4. Scrieți o metodă care primește o `List<ContBancar>` și elimină,

folosind `removeIf`, toate conturile cu soldul sub o valoare dată ca parametru.

5. Explicați, în cuvinte proprii, de ce `Arrays.asList(array)` nu permite `add/remove`, deși permite `set`, și ce relație are acest comportament cu array-ul original.



# Colecții (II): Maps. Vizualizări.

## Colecții imutabile

# 12

### Introducere

Capitolul anterior a acoperit colecțiile care rețin elemente individuale — liste, mulțimi, cozi. Un Map rezolvă o problemă diferită, dar la fel de frecventă: asocierea unei *chei* cu o *valoare*, pentru căutare rapidă după cheie. Încheie capitolul cu instrumente pentru colecții *imutabile* și *vizualizări* — moduri de a expune o colecție în siguranță, fără a permite modificarea ei neintenționată.

## 12.1 Map: operații de bază și actualizare

### 12.1.1 Operații de bază

Reprezentăm banca noastră ca un Map, cu titularul drept cheie și contul asociat drept valoare:

```
1 Map<String, ContBancar> banca = new HashMap<>();
2
3 banca.put("Ana Popescu", new ContBancar("Ana Popescu", 500.0)
4 );
5 banca.put("Ion Ionescu", new ContBancar("Ion Ionescu",
6 1200.0));
7
8 ContBancar cont = banca.get("Ana Popescu");
9 // gaseste contul, sau null daca lipseste
10 boolean exista = banca.containsKey("Maria Popa");
11 // false
12 banca.remove("Ion Ionescu");
13 // elimina intrarea
14 int numarConturi = banca.size();
```

Listing 12.1: Operațiile de bază ale unui HashMap

### Notă

La fel ca la HashSet (Subsecțiunea 11.3.1), cheile unui HashMap se bazează pe equals/hashCode pentru localizare rapidă — String, folosit mai sus ca cheie, are deja ambele metode corect implementate în bibliotecă, exact motivul pentru care este o alegere atât de frecventă pentru chei. Dacă ați folosi ContBancar însuși ca cheie, ar folosi implementarea equals/hashCode din Subsecțiunea 6.3.2, scrisă deja în Capitolul 6.

### 12.1.2 Metode de actualizare convenabile

Câteva metode, introduse în Java 8, simplifică tipare de actualizare frecvente, care altfel ar necesita o verificare explicită containsKey urmată de put:

```
1 double sold = banca.containsKey("Ana Popescu")
2   ? banca.get("Ana Popescu").getSold()
3   : 0.0;
4
5 // Echivalent, mult mai concis:
```

Listing 12.2: getOrDefault și putIfAbsent

Listing 12.3: computeIfAbsent, pentru gruparea valorilor pe chei

```

6 double soldDefault = banca.getOrDefault("Ana Popescu", new
  ContBancar("", 0.0)).getSold();
7
8 banca.putIfAbsent("Ana Popescu", new ContBancar("Ana Popescu"
  , 0.0));
9 // NU suprascrie daca exista deja

```

Listing 12.4: merge, pentru acumularea valorilor pe cheie

```

1 Map<String, List<String>> tranzactiiPeCont = new HashMap<>();
2
3 // Adauga o tranzactie, creand automat lista daca nu exista
4 // deja pentru acea cheie
5 tranzactiiPeCont.computeIfAbsent("Ana Popescu",
6   cheie -> new ArrayList<>())
7   .add("depunere: 150.0");

```

```

1 Map<String, Double> totalDepuneriPeCont = new HashMap<>();
2
3 totalDepuneriPeCont.merge("Ana Popescu", 150.0, Double::sum);
4 // 150.0 (prima depunere)
5 totalDepuneriPeCont.merge("Ana Popescu", 75.0, Double::sum);
6 // 225.0 (acumulat)

```

**Notă**

computeIfAbsent este exact tiparul de care aveți nevoie ori de câte ori grupați elemente pe o cheie derivată (aici, lista tranzacțiilor fiecărui titular) — fără el, codul echivalent ar necesita o verificare explicită `if (!map.containsKey(cheie)) map.put(cheie, new ArrayList<>());` înainte fiecărei adăugări. `merge(cheie, valoare, functie)` combină valoarea existentă (dacă există) cu cea nouă, folosind funcția dată (aici, `Double::sum`, o referință la metodă din Subsecțiunea 7.3.1) — sau inserează direct valoarea nouă, dacă cheia nu există încă.

### 12.1.3 Vizualizări ale unui Map

Un Map oferă trei vizualizări pentru parcurgere, fiecare potrivită unui scop diferit:

Listing 12.5: Cele trei vizualizări ale unui Map: chei, valori, perechi

```

1 for (String titular : banca.keySet()) { // doar cheile
2     System.out.println(titular);
3 }
4
5 for (ContBancar cont : banca.values()) { // doar valorile
6     System.out.println(cont.getSold());
7 }
8
9 for (Map.Entry<String, ContBancar> intrare : banca.entrySet()) {
10     // perechi cheie-valoare
11     System.out.println(intrare.getKey() + " -> " + intrare.
12         getValue().getSold());
13 }

```

**Notă**

Parcurgerea prin `entrySet()` este preferabilă atunci când aveți nevoie și de cheie, și de valoare — o singură parcurgere a structurii interne, spre deosebire de a itera `keySet()` și a apela `banca.get(cheie)`.

pentru fiecare cheie (o a doua căutare, redundantă, pentru fiecare element). Aceste vizualizări sunt *legate* de map-ul original: modificările prin vizualizare (de exemplu, `banca.keySet().remove(...)`) se reflectă direct în map-ul de bază.

## 12.2 Variante specializate de Map/Set

### 12.2.1 LinkedHashMap și LinkedHashSet

Am menționat deja, în Subsecțiunea 11.3.1, că HashSet (și, la fel, HashMap) nu garantează nicio ordine de iterare. LinkedHashMap/LinkedHashSet rezolvă exact această limitare, păstrând **ordinea de inserare**:

```
1 Map<String, ContBancar> bancaOrdonata = new LinkedHashMap<>()
  ;
2
3 bancaOrdonata.put("Ion Ionescu", cont1);
4 bancaOrdonata.put("Ana Popescu", cont2);
5
6 for (String titular : bancaOrdonata.keySet()) {
7     System.out.println(titular);
8     // GARANTAT: "Ion Ionescu" inaintea lui "Ana Popescu"
9 }
```

Listing 12.6: LinkedHashMap păstrează ordinea de inserare a cheilor

#### Notă

Costul acestei garanții suplimentare este mic — LinkedHashMap menține, intern, o listă dublu înălțuită peste structura de hash obișnuită, cu o penalizare de memorie și performanță minoră față de HashMap. Folosiți-o ori de câte ori ordinea de inserare contează pentru logica sau afișarea aplicației, dar nu aveți nevoie de o ordine **sortată** (caz în care TreeMap, mai jos, este alegerea potrivită).

### 12.2.2 TreeMap

TreeMap este, pentru Map, echivalentul lui TreeSet (Subsecțiunea 11.3.2): păstrează cheile ordonate, natural sau printr-un Comparator:

```
1 Map<String, ContBancar> bancaSortata = new TreeMap<>();
2
3 bancaSortata.put("Ion Ionescu", cont1);
4 bancaSortata.put("Ana Popescu", cont2);
5
6 for (String titular : bancaSortata.keySet()) {
7     System.out.println(titular);
8     // "Ana Popescu" inaintea lui "Ion Ionescu" - ordine
9     // alfabetica
10 }
```

Listing 12.7: TreeMap menține cheile în ordine sortată

#### Notă

Cheile unui TreeMap trebuie să fie Comparable (sau trebuie furnizat un Comparator la construcție) — String implementează deja `Comparable<String>` din bibliotecă, motiv pentru care exemplul de mai sus funcționează fără nicio configurare suplimentară. La fel ca TreeSet, operațiile de bază au complexitate  $O(\log n)$ , mai lentă decât  $O(1)$ .

mediu al lui `HashMap`.

### 12.2.3 EnumMap și EnumSet

Când cheile sunt constante ale unui tip `enum` — ca `TipCont`, introdus în Subsecțiunea 6.6.2 — `EnumMap` oferă o implementare specializată, mai eficientă decât un `HashMap` obișnuit:

**Listing 12.8:** `EnumMap`, specializat pentru chei de tip `enum`

```
1 Map<TipCont, Double> comisioanePeTip = new EnumMap<>(TipCont.  
    class);  
2  
3 comisioanePeTip.put(TipCont.STANDARD, 5.0);  
4 comisioanePeTip.put(TipCont.ECONOMII, 0.0);  
5 comisioanePeTip.put(TipCont.PREMIUM, 15.0);  
6  
7 double comision = comisioanePeTip.get(TipCont.PREMIUM);  
8 // 15.0
```

#### Notă

Intern, `EnumMap` este implementat printr-un simplu array, indexat direct după poziția ordinală a constantei `enum` (`ordinal()`) — mult mai rapid și mai compact în memorie decât o tabelă de dispersie, exact fiindcă mulțimea completă a cheilor posibile este cunoscută dinainte, la definirea `enum`-ului. `EnumSet` oferă aceeași optimizare pentru o `Set` de constante `enum`. Folosiți-le implicit ori de câte ori cheile/elementele sunt, prin natura lor, valori ale unui `enum`.

### 12.2.4 WeakHashMap și IdentityHashMap (context)

Menționăm, pentru completitudine, două variante specializate, rar folosite direct în cod de aplicație obișnuit:

- **WeakHashMap** reține cheile prin *referințe slabe* — o intrare dispare automat din map atunci când cheia sa nu mai este referită de nicio altă parte a programului, util pentru implementarea unor cache-uri care nu trebuie să prevină eliberarea memoriei obiectelor-cheie.
- **IdentityHashMap** compară cheile prin `==` (identitate de referință), nu prin `equals` — util în situații foarte specifice (de exemplu, unelte de analiză a graficelor de obiecte), unde egalitatea logică obișnuită ar produce rezultate greșite.

#### Notă

Ambele sunt instrumente de nișă, nu alegeri implicite — pentru marea majoritate a codului de aplicație, `HashMap`, `LinkedHashMap` sau `Treemap` rămân alegerile corecte. Le menționăm aici pentru ca, atunci când le veți întâlni (de exemplu, într-o bibliotecă de cache-uri), să recunoașteți diferența de comportament față de un `Map` obișnuit.

## 12.3 Copii și vizualizări imutabile

### 12.3.1 Colecții mici, imutabile prin construcție

Metodele factory statice `List.of`, `Set.of` și `Map.of`, introduse în Java 9, creează colecții **imutabile**, direct din valorile date:

```
1 List<String> zile = List.of("Luni", "Marti", "Miercuri");
2 Set<String> orase = Set.of("Brasov", "Sibiu", "Cluj");
3 Map<String, Double> comisioane = Map.of("STANDARD", 5.0, "
    PREMIUM", 15.0);
4
5 zile.add("Joi"); // arunca UnsupportedOperationException -
6                 // lista e imutabila
```

**Listing 12.9:** Colecții imutabile, create direct din valori

#### Notă

Orice încercare de modificare (`add`, `remove`, `set`, `put`) asupra colecțiilor create astfel aruncă `UnsupportedOperationException` la execuție — compilatorul nu poate detecta această restricție static, întrucât tipul declarat rămâne interfața obișnuită (`List<String>`), nu un tip special „imutabil”. Aceste metode nu acceptă elemente `null`; folosiți-le pentru date constante, cunoscute la scrierea codului, similar constantelor static `final` din Subsecțiunea 4.4.4.

### 12.3.2 Copii și vizualizări imutabile ale colecțiilor existente

Pentru o colecție deja existentă, mutabilă, două tehnici diferite produc rezultate aparent similare, dar semantic distincte:

```
1 List<ContBancar> conturiMutabile = new ArrayList<>();
2 conturiMutabile.add(new ContBancar("Ana Popescu", 500.0));
3
4 List<ContBancar> vizualizare = Collections.unmodifiableList(
5     conturiMutabile);
6
7 List<ContBancar> copieIndependenta = List.copyOf(
8     conturiMutabile);
9
10 conturiMutabile.add(new ContBancar("Ion Ionescu", 1200.0));
11
12 System.out.println(vizualizare.size());
13 // 2 - reflecta schimbarea din original!
14 System.out.println(copieIndependenta.size());
15 // 1 - neschimbata, complet independenta
```

**Listing 12.10:** Vizualizare imutabilă (legată de original) versus copie imutabilă (independentă)

#### Notă

`Collections.unmodifiableList` (și variantele sale pentru `Set/Map`) produce doar o **vizualizare** care interzice modificarea prin acea referință, dar rămâne legată de colecția originală — dacă originalul se schimbă (prin altă referință, ca `conturiMutabile` de mai sus), vizualizarea reflectă imediat schimbarea. `List.copyOf` (Java 10+), în schimb, produce o copie complet **independentă** și imutabilă, neafectată de modificări ulterioare ale colecției sursă. Alegeți `copyOf` atunci când aveți nevoie de o garanție reală de neschimbare (de exemplu, la returnarea unei colecții interne dintr-o

metodă publică, evitând exact problema de *expunere a reprezentării interne* din Subsecțiunea 5.4.2).

### 12.3.3 Subranges

O *vizualizare* poate expune și doar o **porțiune** a unei colecții, fără a o copia:

**Listing 12.11:** `subList`, o vizualizare asupra unei porțiuni a listei

```
1 List<ContBancar> toateConturile = new ArrayList<>(List.of(
    cont1, cont2, cont3, cont4));
2
3 List<ContBancar> primeleDoua = toateConturile.subList(0, 2);
4 // vizualizare, NU copie
5
6 primeleDoua.clear(); // elimina efectiv primele doua
7 // elemente din toateConturile!
```

Pentru colecții ordonate (`TreeMap`/`TreeSet`, Subsecțiunea 12.2.2), vizualizările pe intervale folosesc **ordinea** elementelor, nu poziția lor:

**Listing 12.12:** Vizualizări pe interval de valori, pentru colecții sortate

```
1 TreeMap<String, ContBancar> bancaSortata = new TreeMap<>(
    banca);
2
3 SortedMap<String, ContBancar> primaJumatateAlfabet =
    bancaSortata.headMap("M");
4 // chei < "M"
5 SortedMap<String, ContBancar> aDouaJumatate = bancaSortata.
    tailMap("M");
6 // chei >= "M"
```

#### Notă

La fel ca `subList`, `headMap`/`tailMap`/`subMap` returnează **vizualizări**, nu copii — modificările prin ele se reflectă în map-ul original, și invers. Această comportare consecventă (vizualizare, nu copie, ca implicit al metodelor de „extragere de porțiune”) este intenționată în întreg framework-ul de colecții, dar merită reținută explicit, fiindcă intuiția „aceasta arată ca o copie” este o presupunere greșită frecventă.

### 12.3.4 Vizualizări inversate și sincronizate

**Listing 12.13:** O vizualizare cu ordine inversată, și una sincronizată pentru acces concurent

```
1 List<ContBancar> conturiInverse = toateConturile.reversed();
2 // ordine INVERSA, vizualizare
3
4 List<ContBancar> conturiSincronizate =
5     Collections.synchronizedList(new ArrayList<>());
6 // sigura pentru acces din mai multe thread-uri
```

#### Notă

`Collections.synchronizedList`/`synchronizedMap`/`synchronizedSet` adaugă sincronizare de bază peste o colecție obișnuită, nesigură implicit în context concurent — vom relua acest subiect riguros, alături de alternative mai performante, în Capitolul 13–Capitolul 14. Pentru moment, e suficient să recunoașteți că nicio colecție întâlnită până acum (`ArrayList`, `HashMap` etc.) nu este sigură implicit pentru acces simultan din mai multe fire de execuție.

## 12.4 Colecții legacy (context istoric)

Câteva clase, prezente în Java încă dinaintea framework-ului de colecții modern (introdus în Java 2), rămân în bibliotecă din motive de compatibilitate, dar sunt considerate azi depășite:

- **Hashtable** este predecesorul istoric al lui `HashMap`, sincronizat implicit (deci mai lent, chiar și în cod `single-thread`) — folosiți `HashMap` sau, pentru context concurent, alternativele din Capitolul 14.
- **Stack** extinde de fapt `Vector` (un alt predecesor istoric al lui `ArrayList`), moștenind metode nepotrivite unei stive curate — folosiți `ArrayDeque` (Subsecțiunea 11.4.1) ca stivă modernă, mai eficientă și mai coerentă.
- **BitSet** rămâne, spre deosebire de celelalte două, o alegere legitimă și azi — un vector eficient de biți, potrivit pentru mulțimi dense de numere întregi mici (de exemplu, marcarea unor indici dintr-un interval cunoscut), mult mai compact în memorie decât un `HashSet<Integer>` echivalent.

### Notă

Recunoașteți aceste clase dacă le întâlniți în cod mai vechi (multe proiecte Java aflate încă în producție au fost scrise înainte de alternativele moderne), dar evitați-le în cod nou, cu excepția `BitSet`, pentru cazul specific descris mai sus.

### Rezumat

- `Map` asociază chei cu valori; `getOrDefault`, `putIfAbsent`, `computeIfAbsent` și `merge` simplifică tipare frecvente de actualizare; `entrySet()` e preferat când e nevoie de ambele, cheie și valoare.
- `LinkedHashMap/LinkedHashSet` păstrează ordinea de inserare; `TreeMap/TreeSet` păstrează o ordine sortată; `EnumMap/EnumSet` sunt optimizate specific pentru chei/elemente de tip `enum`.
- `List.of/Set.of/Map.of` creează colecții mici, imutabile prin construcție; `List.copyOf` produce o copie independentă, spre deosebire de `Collections.unmodifiableList`, care rămâne o vizualizare legată de colecția originală.
- `subList`, `headMap/tailMap` și vizualizările inversate/sincronizate sunt, implicit, **vizualizări** legate de colecția sursă, nu copii independente.
- `Hashtable` și `Stack` sunt clase legacy, de evitat în cod nou; `BitSet` rămâne o alegere validă pentru mulțimi dense de întregi mici.

### Exerciții

1. Folosind `computeIfAbsent`, scrieți o metodă care grupează o listă de `ContBancar` într-un `Map<TipCont, List<ContBancar>>`, pe baza tipului fiecărui cont.
2. Rescrieți banca ca `TreeMap`, apoi extrageți, cu `headMap/tailMap`, doar conturile ai căror titulari încep cu literele A–M.
3. Explicați, cu un exemplu de cod, diferența de comportament dintre `Collections.unmodifiableList(lista)` și

`List.copyOf(lista)`, atunci când lista originală este modificată ulterior.

4. Scrieți o metodă publică `getConturi()` într-o clasă `Banca`, care returnează în siguranță lista internă de conturi, fără a permite apelantului să o modifice direct (evitând expunerea reprezentării interne, Subsecțiunea 5.4.2).
5. Folosind un `EnumMap<TipCont, Double>`, calculați soldul total al conturilor, grupat pe tip de cont, pornind de la o listă mixtă de `ContBancar`.

## Introducere

Toate programele scrise până acum în acest curs au rulat *secvențial*: o instrucțiune se execută, apoi următoarea. *Concurența* permite executarea mai multor fluxuri de instrucțiuni — *thread-uri* — aparent simultan, fie prin adevărat paralelism (pe procesoare multi-core), fie prin comutare rapidă între thread-uri pe un singur nucleu [1]. Acest capitol introduce mecanismul de bază al thread-urilor și instrumentele moderne de gestionare a lor; capitolul următor tratează sincronizarea — coordonarea corectă a thread-urilor care partajează date.

## 13.1 Pornirea thread-urilor; stările unui thread

### 13.1.1 Crearea și pornirea unui thread

Cel mai simplu mod de a crea un thread este să furnizați un `Runnable` — o interfață funcțională deja familiară din Subsecțiunea 7.2.2, cu o singură metodă `run()`, fără parametri și fără valoare de retur:

```
1 Runnable sarcina = () -> {
2     System.out.println("Procesare pe thread-ul: " + Thread.
3         currentThread().getName());
4 };
5 Thread thread = new Thread(sarcina);
6 thread.start(); // porneste thread-ul
7                // NU apelati direct run()!
```

Listing 13.1: Crearea și pornirea unui thread, cu un `Runnable` exprimat ca lambda

## Notă

Apelul `thread.start()` este esențial diferit de `thread.run()`: primul creează efectiv un nou fir de execuție, care rulează `Runnable`-ul **concurent** cu thread-ul apelant; al doilea ar executa pur și simplu metoda `run()` **secvențial**, pe thread-ul curent, exact ca orice apel obișnuit de metodă — fără niciun beneficiu de concurență. Această confuzie este o greșeală frecventă la prima întâlnire cu thread-urile.

## Notă

O alternativă istorică, extinderea directă a clasei `Thread` (`class SarcinaMea extends Thread`), este posibilă, dar descurajată: exact ca la orice altă clasă (Secțiunea 6.1), extinderea `Thread` consumă singura moștenire de clasă disponibilă, împiedicând clasa să mai extindă altceva. Implementarea `Runnable` (sau, mai simplu, o expresie lambda, ca mai sus) evită această limitare.

### 13.1.2 Stările unui thread

Un thread trece printr-un set bine definit de stări pe durata vieții sale:

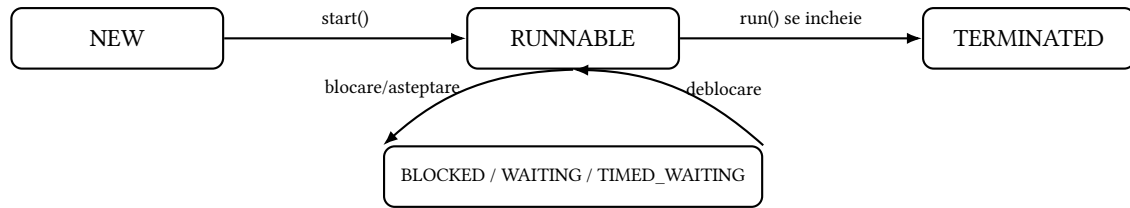


Figura 13.1: Ciclul de viață (simplificat) al unui thread Java.

#### Notă

RUNNABLE nu înseamnă „rulează efectiv chiar acum” — înseamnă doar „eligibil pentru execuție”; planificatorul JVM/sistemului de operare decide, la nivel foarte fin, care thread rulează efectiv pe fiecare nucleu, la fiecare moment. Un thread ajunge în BLOCKED atunci când așteaptă acces la o resursă blocată de alt thread (Capitolul 14), în WAITING/TIMED\_WAITING atunci când așteaptă explicit un semnal sau un interval de timp (de exemplu, `Thread.sleep(...)` sau `join()`).

### 13.1.3 Întreruperea thread-urilor; daemon threads; priorități

Java nu oferă o modalitate de a **forța** oprirea unui thread din exterior — din motive de siguranță (un thread oprit brusc ar putea lăsa date într-o stare inconsistentă). În schimb, `interrupt()` *semnalează* o cerere de oprire, pe care thread-ul o poate observa și trata cooperant:

Listing 13.2: Întreruperea cooperantă a unui thread

```

1 Thread thread = new Thread(() -> {
2     while (!Thread.currentThread().isInterrupted()) {
3         // ... lucru repetitiv ...
4     }
5     System.out.println("Thread oprit cooperant");
6 });
7
8 thread.start();
9 thread.interrupt(); // SEMNALEAZA intreruperea,
10                    // nu opreste forat

```

#### Notă

Metodele care „așteaptă” (`Thread.sleep`, `wait`, `join`) reacționează automat la întrerupere, aruncând `InterruptedException` (o excepție verificată, Subsecțiunea 9.1.2) — codul care le apelează trebuie fie să o propage mai departe, fie să restabilească explicit statusul de întrerupere (`Thread.currentThread().interrupt();`) în blocul catch, pentru a nu „înghiți” tăcut semnalul de oprire, exact greșeala discutată în Subsecțiunea 9.3.3.

Un thread *daemon* nu împiedică oprirea programului — JVM-ul se oprește automat când ultimul thread **ne-daemon** se termină, indiferent câte thread-uri daemon mai rulează:

Listing 13.3: Un thread daemon, pentru sarcini de fundal

```

1 Thread threadFundal = new Thread(() -> { /* monitorizare
2     continua */ });
3 threadFundal.setDaemon(true); // programul se poate incheia
4                                // fara sa astepte acest thread
5 threadFundal.start();

```

**Notă**

Prioritățile de thread (`setPriority`, valori între 1 și 10) sunt doar o *sugestie* pentru planificatorul sistemului de operare, nu o garanție — comportamentul exact variază semnificativ între platforme, iar codul corect **nu** ar trebui să depindă de o anumită prioritate pentru corectitudine, ci doar, eventual, pentru optimizări minore de performanță.

## 13.2 Virtual threads

### 13.2.1 Motivație: platform threads versus virtual threads

Thread-urile discutate până acum — numite, prin contrast, *platform threads* — sunt legate direct de thread-uri ale sistemului de operare, o resursă relativ costisitoare: crearea a mii de platform threads simultan epuizează rapid memoria și degradează performanța. *Virtual threads* (stabilizate în Java 21) sunt gestionate de JVM însăși, mult mai ieftine, permițând milioane de thread-uri concurente:

```
1 Thread virtual = Thread.ofVirtual().start(() -> {
2     System.out.println("Ruleaza pe: " + Thread.currentThread());
3 });
4
5 virtual.join(); // asteapta terminarea, la fel ca la un
6                // platform thread obisnuit
```

Listing 13.4: Crearea unui virtual thread

**Notă**

Din perspectiva codului scris în Subsecțiunea 13.1.1, un virtual thread se comportă identic cu un platform thread — aceeași clasă `Thread`, aceleași metode (`start`, `join`, `interrupt`). Diferența este **internă**: JVM-ul multiplexează un număr mare de virtual threads peste un număr mic de platform threads reale, suspendând eficient un virtual thread blocat (de exemplu, pe o operație de rețea) fără a bloca thread-ul de sistem de operare subiacent.

**Notă**

Virtual threads aduc beneficii majore pentru sarcini **dominate de așteptare** (*I/O-bound*) — de exemplu, mii de conexiuni de rețea simultane (Capitolul 20), fiecare așteptând răspunsul unui server. Pentru sarcini **dominate de calcul** (*CPU-bound*), unde procesorul este ocupat continuu, virtual threads nu oferă niciun avantaj față de platform threads — numărul de nuclee fizice rămâne limitarea reală, indiferent de câte thread-uri (virtuale sau nu) sunt create.

## 13.3 Coordonarea task-urilor: Callable/Future; ExecutorService

### 13.3.1 Callable și Future

Spre deosebire de `Runnable`, care nu returnează nimic, `Callable<V>` este o interfață funcțională a cărei metodă returnează o valoare (și poate

**Listing 13.5:** Callable, o alternativă la Runnable care returnează un rezultat

arunca o excepție verificată):

```
1 Callable<Double> calculSold = () -> {
2     double total = 0;
3     for (ContBancar cont : conturi) {
4         total += cont.getSold();
5     }
6     return total;
7 };
```

Rezultatul unui Callable lansat asincron este reprezentat printr-un Future<V> — o „promisiune” a unei valori care va deveni disponibilă mai târziu:

**Listing 13.6:** Obținerea rezultatului dintr-un Future

```
1 Future<Double> viitorSold = executor.submit(calculSold);
2     // pornit ASINCRON, vezi 13.3.2
3
4 // ... alt cod, executat concurent cu calculul de mai sus ...
5
6 double sold = viitorSold.get(); // BLOCHEAZA pana cand
7     // rezultatul e gata
```

#### Notă

get() blochează thread-ul apelant până când calculul asincron se termină — dacă acel calcul a aruncat o excepție, get() o relansează, învelită într-o ExecutionException (mecanismul de *chaining* din Subsecțiunea 9.2.3). Există și o variantă cu timeout, get(timp, unitate), care aruncă TimeoutException dacă rezultatul nu e gata în intervalul dat, evitând o așteptare nedefinită.

### 13.3.2 ExecutorService

Crearea manuală a câte unui Thread pentru fiecare sarcină, ca în Subsecțiunea 13.1.1, devine greu de gestionat la scară — un ExecutorService administrează un grup (*pool*) de thread-uri, reutilizate pentru mai multe sarcini succesive:

**Listing 13.7:** Crearea și folosirea unui ExecutorService

```
1 ExecutorService executor = Executors.newFixedThreadPool(4);
2     // 4 thread-uri, reutilizate
3
4 Future<Double> viitorSold = executor.submit(calculSold);
5 executor.execute(() -> System.out.println("Sarcina fara
6     rezultat"));
7     // ca Runnable, fara Future
8 executor.shutdown(); // OBLIGATORIU - altfel thread-urile
9     // pool-ului raman active la infinit
```

**Tabela 13.1:** Cele mai frecvente metode factory pentru crearea unui ExecutorService.

| Metodă factory                    | Comportament                                                |
|-----------------------------------|-------------------------------------------------------------|
| newFixedThreadPool(n)             | exact n thread-uri, reutilizate                             |
| newCachedThreadPool()             | crează thread-uri după nevoie, le elimină după inactivitate |
| newSingleThreadExecutor()         | un singur thread, sarcinile se execută strict secvențial    |
| newVirtualThreadPerTaskExecutor() | câte un virtual thread (Secțiunea 13.2) per sarcină         |

**Notă**

Uitarea apelului `shutdown()` este o greșeală frecventă — thread-urile unui `ExecutorService` nu sunt daemon (Subsecțiunea 13.1.3) implicit, deci un program care creează un executor și nu îl închide explicit nu se va termina niciodată, chiar dacă toate sarcinile utile au fost deja executate. `newVirtualThreadPerTaskExecutor()` combină simplitatea API-ului `ExecutorService` cu eficiența virtual threads din Secțiunea 13.2 — o combinație din ce în ce mai recomandată pentru sarcini I/O-bound.

### 13.3.3 Invocarea unui grup de task-uri

Pentru a lansa mai multe sarcini simultan și a aștepta **toate** rezultatele, `invokeAll` este mai convenabil decât `submit` repetat:

```

1 List<Callable<Double>> sarciniPeGrup = List.of(
2     () -> calculSoldPentruGrup(conturiGrupa1),
3     () -> calculSoldPentruGrup(conturiGrupa2),
4     () -> calculSoldPentruGrup(conturiGrupa3)
5 );
6
7 List<Future<Double>> rezultate = executor.invokeAll(
8     sarciniPeGrup);
9 // asteapta TOATE
10
11 double totalGeneral = 0;
12 for (Future<Double> f : rezultate) {
13     totalGeneral += f.get();
14 }

```

Listing 13.8: `invokeAll`, pentru un grup de sarcini executate concurrent

**Notă**

`invokeAll` blochează până când **toate** sarcinile se termină (fie cu succes, fie prin excepție), returnând o listă de `Future`-uri, în aceeași ordine ca lista de `Callable`-uri primită. `invokeAny`, în schimb, returnează direct rezultatul **primei** sarcini care se termină cu succes, anulând restul — util atunci când mai multe strategii alternative rezolvă aceeași problemă și e suficient primul rezultat obținut.

## 13.4 Thread-local variables; scoped values; fork-join

### 13.4.1 ThreadLocal

Un `ThreadLocal<T>` oferă fiecărui thread propria copie **independentă** a unei variabile — utilă pentru date care nu ar trebui partajate între thread-uri (de exemplu, un format sau un context specific fiecărui thread):

```

1 private static final ThreadLocal<Integer>
2     numarTranzactiiProcesate =
3     ThreadLocal.withInitial(() -> 0);
4
5 public void proceseazaTranzactie() {
6     numarTranzactiiProcesate.set(numarTranzactiiProcesate.get() + 1);
7 }

```

Listing 13.9: `ThreadLocal`, cu o copie independentă per thread

```

6 // fiecare thread isi vede PROPRIA valoare, independenta
7 // de celelalte thread-uri
8 }

```

#### Notă

Într-un `ExecutorService` cu thread-uri reutilizate (Subsecțiunea 13.3.2), un `ThreadLocal` necurățat poate cauza scurgeri de memorie sau, mai subtil, poate „scurge” valori vechi către o sarcină ulterioară executată pe același thread reutilizat — apălați explicit `remove()` la finalul sarcinii, dacă folosiți `ThreadLocal` într-un pool de thread-uri.

### 13.4.2 Scoped values

`ScopedValue` (stabilizat recent) oferă o alternativă mai sigură la `ThreadLocal`, pentru cazul frecvent în care valoarea trebuie să fie disponibilă doar pe durata unui apel bine delimitat, fără riscul de a „uita” să o curățați:

Listing 13.10: `ScopedValue`, legat explicit doar pentru durata unui bloc

```

1 private static final ScopedValue<String> UTILIZATOR_CURENT =
   ScopedValue.newInstance();
2
3 public void proceseazaCaUtilizator(String utilizator) {
4     ScopedValue.where(UTILIZATOR_CURENT, utilizator).run(()
5     -> {
6         proceseazaTranzactie(); // in interiorul acestui
7         // bloc, UTILIZATOR_CURENT.get() e disponibil
8     });
9     // aici, in afara blocului, legatura a disparut automat
10    // - fara remove() manual
11 }

```

#### Notă

Spre deosebire de `ThreadLocal`, un `ScopedValue` este **imutabil** pe durata legăturii sale și se „dezleagă” automat la ieșirea din blocul `run(...)` — nu există echivalent al lui `set()` ulterior, nici riscul de a uita un `remove()`. Această restricție, aparent mai puțin flexibilă, elimină exact clasa de erori (scurgeri, valori vechi „uite”) discutată la `ThreadLocal` mai sus.

### 13.4.3 Framework-ul fork-join (context)

Framework-ul *fork-join* este specializat pentru probleme rezolvabile prin *divide et impera*: o sarcină mare este împărțită (*fork*) recursiv în sub-sarcini mai mici, executate concurrent, ale căror rezultate sunt apoi combinate (*join*):

Listing 13.11: O sarcină fork-join, pentru însumarea recursivă a unui array

```

1 public class SumaTask extends RecursiveTask<Double> {
2     private final double[] valori;
3     private final int inceput, sfarsit;
4
5     // ... constructor ...
6
7     @Override
8     protected Double compute() {
9         if (sfarsit - inceput <= 1000) {
10            // suficient de mic - calculeaza direct

```

```

11         double suma = 0;
12         for (int i = inceput; i < sfarsit; i++)
13             suma += valori[i];
14         return suma;
15     }
16     int mijloc = (inceput + sfarsit) / 2;
17     SumaTask stanga = new SumaTask(valori, inceput,
18     mijloc);
19     SumaTask dreapta = new SumaTask(valori, mijloc,
20     sfarsit);
21     stanga.fork(); // executa asincron jumatatea stanga
22     return dreapta.compute() + stanga.join();
23     // calculeaza dreapta direct, apoi asteapta stanga
24 }

```

### Notă

Framework-ul fork-join folosește intern o tehnică numită *work-stealing*: thread-urile fără sarcini de făcut „fură” sub-sarcini de la thread-uri ocupate, echilibrând automat încărcarea. Este util pentru probleme cu adevărat CPU-bound, ușor divizibile recursiv (sortare, căutare, procesare de array-uri mari) — mult mai puțin relevant pentru sarcini I/O-bound, unde virtual threads (Secțiunea 13.2) sunt, azi, alegerea mai simplă și la fel de eficientă.

### Rezumat

- ▶ Un Thread pornește cu `start()` (niciodată `run()` direct); parcurge stările NEW, RUNNABLE, BLOCKED/WAITING, TERMINATED; `interrupt()` semnalează cooperant o cerere de oprire.
- ▶ Virtual threads (Java 21+) sunt gestionate de JVM, mult mai ieftine decât platform threads, ideale pentru sarcini I/O-bound cu concurență masivă.
- ▶ `Callable<V>` (spre deosebire de `Runnable`) returnează o valoare, reprezentată asincron printr-un `Future<V>`; `ExecutorService` administrează un pool de thread-uri reutilizate — necesită `shutdown()` explicit.
- ▶ `invokeAll` așteaptă toate sarcinile dintr-un grup; `invokeAny` returnează prima reușită, anulând restul.
- ▶ `ThreadLocal` oferă câte o copie independentă per thread, cu riscul de scurgeri în pool-uri reutilizate; `ScopedValue` rezolvă același caz de utilizare mai sigur, cu legare/dezlegare automată pe durata unui bloc.
- ▶ Fork-join divizează recursiv o sarcină mare, combinând rezultatele parțiale — potrivit pentru probleme CPU-bound, ușor divizibile.

### Exerciții

1. Scrieți un program care pornește trei thread-uri, fiecare afișând numele său și numărând până la 5, cu o pauză de 500ms între numere (`Thread.sleep`), apoi așteaptă terminarea tuturor cu `join()`.
2. Folosind un `ExecutorService` cu 3 thread-uri, procesați concurrent o listă de 10 conturi `ContBancar`, calculând suma totală a soldurilor prin `invokeAll` pe sub-liste.
3. Creați 10 000 de virtual threads, fiecare simulând o operațiune

I/O scurtă (`Thread.sleep(100)`), și măsurați timpul total de execuție; comparați, dacă doriți, cu o încercare echivalentă folosind `platform threads` dintr-un pool fix mic.

4. Scrieți o mică demonstrație `ThreadLocal`, în care mai multe thread-uri incrementează, fiecare, propriul contor independent, și afișați valorile finale distincte ale fiecărui thread.
5. Explicați, în cuvinte proprii, de ce `ScopedValue` este considerat mai sigur decât `ThreadLocal` pentru transmiterea unui context (de exemplu, identitatea utilizatorului curent) printr-un lanț de apeluri de metode.

# Concurență (II): sincronizare.

## Colecții concurente.

## CompletableFuture

# 14

### Introducere

Capitolul anterior a arătat cum se creează și se coordonează thread-uri, dar a evitat cu grijă o întrebare esențială: ce se întâmplă atunci când mai multe thread-uri accesează **aceleași** date? Exercițiul cu `ExecutorService` din Subsecțiunea 13.3.3 a funcționat corect fiindcă fiecare thread lucra pe propriul sub-set de conturi, fără suprapunere. Acest capitol tratează exact cazul opus: date partajate, accesate concurent — sursa clasică de erori subtile în programarea multi-thread.

## 14.1 Condiții de cursă; `synchronized`; `deadlock-uri`

### 14.1.1 Condiții de cursă

Reluăm `ContBancar`. `depune`, de data aceasta apelată **concurent**, din mai multe thread-uri, pe același obiect:

```
1 ContBancar cont = new ContBancar("Ana Popescu", 0.0);
2
3 ExecutorService executor = Executors.newFixedThreadPool(8);
4 for (int i = 0; i < 1000; i++) {
5     executor.execute(() -> cont.depune(1.0));
6 }
7 executor.shutdown();
8 executor.awaitTermination(10, TimeUnit.SECONDS);
9
10 System.out.println(cont.getSold()); // AȘTEPTAT: 1000.0 -
11                                     // REAL: adesea MAI PUTIN!
```

**Listing 14.1:** O condiție de cursă: rezultatul final este, aproape sigur, greșit

### Notă

Instrucțiunea `sold = sold + suma`; din `depune` pare o singură operație, dar la nivelul procesorului sunt de fapt trei pași separați: citește `sold`, adună `suma`, scrie rezultatul înapoi. Dacă două thread-uri execută acești pași **intercalat** (thread A citește `sold=5`, thread B citește tot `sold=5` înainte ca A să fi scris rezultatul, ambele scriu 6 în loc de 7 așteptat), o actualizare se pierde silențios. Acest fenomen — rezultatul final depinde de ordinea imprevizibilă de intercalare a thread-urilor — se numește *race condition*.

### 14.1.2 `synchronized`; conceptul de monitor

Cuvântul cheie `synchronized` garantează că cel mult **un singur** thread execută, la un moment dat, codul protejat pentru un obiect anume:

```
1 public synchronized void depune(double suma) {
2     sold = sold + suma; // acum ATOMIC, din perspectiva
3                         // altor thread-uri
4 }
```

**Listing 14.2:** `ContBancar.depune`, corectată cu `synchronized`

**Definiție — Monitor**

Fiecare obiect Java are asociat, implicit, un *monitor* (numit uneori *lock intrinsec*). O metodă (sau bloc) `synchronized` **achiziționează** monitorul obiectului înainte de a executa, și îl **eliberează** la ieșire (normal sau prin excepție) — orice alt thread care încearcă să intre într-un bloc sincronizat pe **același** obiect așteaptă până la eliberare.

**Listing 14.3:** Formă echivalentă: bloc `synchronized` explicit, pe `this`

```
1 public void depune(double suma) {
2     synchronized (this) {
3         // echivalent cu metoda synchronized de mai sus
4         sold = sold + suma;
5     }
6 }
```

**Notă**

Forma de bloc oferă control mai fin — puteți sincroniza doar porțiunea de cod care chiar are nevoie de protecție, nu întreaga metodă, și puteți sincroniza pe un **alt** obiect decât `this` (util atunci când mai multe metode ar trebui să partajeze același *lock*, sau, dimpotrivă, *lock*-uri diferite pentru părți independente ale stării unui obiect). Rescriind exemplul din Subsecțiunea 14.1.1 cu `depune` sincronizată, rezultatul final este garantat 1000.0, indiferent de ordinea de execuție a thread-urilor.

### 14.1.3 Deadlock-uri

Sincronizarea rezolvă condițiile de cursă, dar introduce un risc nou: dacă două thread-uri așteaptă, fiecare, un *lock* deținut de celălalt, niciunul nu mai poate continua — un *deadlock*:

**Listing 14.4:** Un deadlock clasic: transfer în direcții opuse, între aceleași două conturi

```
1 public void transferaCatre(ContBancar destinatie, double suma
2     ) {
3     synchronized (this) {
4         synchronized (destinatie) {
5             this.retrage(suma);
6             destinatie.depune(suma);
7         }
8     }
9 }
10 // Thread 1: contA.transferaCatre(contB, 100); - blocheaza
11 // contA, apoi asteapta contB
12 // Thread 2: contB.transferaCatre(contA, 50); - blocheaza
13 // contB, apoi asteapta contA
14 // -> DEADLOCK, daca cele doua apeluri se intretes
15 // nefericit
```

**Notă**

Fiecare thread deține un *lock* și așteaptă celălalt *lock*, deținut de partener — niciunul nu cedează, ambele așteaptă la infinit. Cea mai simplă strategie de evitare este o **ordine consecventă** de achiziție a *lock*-urilor, aceeași pentru toate thread-urile (de exemplu, întotdeauna contul cu identificatorul mai mic primul), eliminând posibilitatea unui ciclu de așteptare. O alternativă, mai flexibilă, este folosirea unui `Lock` explicit cu `tryLock` și `timeout`, discutată în secțiunea următoare.

## 14.2 Lock/Condition; volatile; tipuri atomice

### 14.2.1 Lock explicit

Pachetul `java.util.concurrent.locks` oferă o alternativă mai flexibilă la `synchronized`, cu control explicit asupra achiziției și eliberării:

```

1 private final Lock lacat = new ReentrantLock();
2
3 public void depune(double suma) {
4     lacat.lock();
5     try {
6         sold = sold + suma;
7     } finally {
8         lacat.unlock(); // OBLIGATORIU in finally - altfel
9                         // lacatul ramane blocat la exceptie
10    }
11 }
```

**Listing 14.5:** `ReentrantLock`, echivalentul explicit al `synchronized`

#### Notă

Spre deosebire de `synchronized`, care eliberează automat *lock*-ul la ieșirea din bloc (chiar și printr-o excepție), un `Lock` explicit necesită eliberare manuală — de aici necesitatea strictă a blocului `finally` (Subsecțiunea 9.2.1), altfel un *lock* neeliberat ar bloca la infinit orice alt thread care îl așteaptă. În schimb, `Lock` oferă `tryLock(timp, unitate)` — o încercare de achiziție cu timeout, care evită complet scenariul de deadlock din Subsecțiunea 14.1.3, renunțând elegant dacă *lock*-ul nu devine disponibil la timp.

### 14.2.2 Câmpuri volatile

Un câmp `volatile` garantează vizibilitate — o scriere făcută de un thread este imediat vizibilă pentru toate celelalte — dar **nu** garantează atomicitate pentru operații compuse precum `sold = sold + suma`:

```

1 private volatile boolean opritCerut = false;
2
3 public void ruleazaPanaOprit() {
4     while (!opritCerut) {
5         // ... lucru repetitiv ...
6     }
7 }
8
9 public void cereOprire() {
10    opritCerut = true; // vizibil IMEDIAT pentru thread-ul
11                      // care verifica bucla
12 }
```

**Listing 14.6:** `volatile`, potrivit pentru un simplu semnal de oprire, nu pentru `sold`

#### Notă

Fără `volatile`, un thread ar putea, teoretic, să nu observe niciodată modificarea lui `opritCerut` făcută de alt thread, din cauza optimizărilor de cache ale procesorului. `volatile` rezolvă exact această problemă de vizibilitate, dar **nu** este suficient pentru `sold` din Subsecțiunea 14.1.1 — operația citește-adună-scrie rămâne neatomică chiar dacă fiecare citire/scriere individuală este vizibilă imediat. Pentru contoare simple, incrementate atomic, tipurile atomice de mai jos sunt

soluția potrivită.

### 14.2.3 Tipuri atomice

Pachetul `java.util.concurrent.atomic` oferă tipuri care garantează operații compuse atomice, fără sincronizare explicită, folosind instrucțiuni speciale ale procesorului (*compare-and-swap*):

**Listing 14.7:** `AtomicLong`, pentru un contor thread-safe fără `synchronized`

```
1 private final AtomicLong numarTranzactii = new AtomicLong(0);
2
3 public void inregistreazaTranzactie() {
4     numarTranzactii.incrementAndGet(); // ATOMIC - fara nicio
      sincronizare explicita
5 }
```

#### Notă

Pentru contoare simple și alte operații compuse elementare (incrementare, adunare, comparare-și-setare), tipurile atomice sunt de regulă mai rapide decât `synchronized` sau `Lock` explicit, fiindcă evită complet costul de a bloca/debloca un thread — mecanismul *compare-and-swap* reîncearcă operația la nivel de procesor, fără a suspenda niciun thread. Pentru sold, care e de tip `double`, ar fi nevoie de `DoubleAdder` sau de o combinație atomică pe biți — în practică, pentru logica financiară din `ContBancar`, `synchronized` rămâne mai simplu și suficient de eficient.

### 14.2.4 Inițializare on-demand; publicare sigură

Două tipare, frecvente în cod concurent real, merită menționate pe scurt. *Inițializarea on-demand*, pentru o valoare costisitor de creat, calculată o singură dată, la prima cerere:

**Listing 14.8:** Inițializare on-demand, sigură la concurență, prin clasa imbricată statică (Subsecțiunea 7.4.1)

```
1 public class ConfigurareBanca {
2     private static class Detinator { // clasa imbricata
3         // statica, incarcata LENES
4         static final ConfigurareBanca INSTANTA = new
      ConfigurareBanca();
5     }
6
7     public static ConfigurareBanca obtine() {
8         return Detinator.INSTANTA; // JVM-ul garanteaza
9         // incarcarea thread-safe a claselor
10    }
11 }
```

#### Notă

Acest tipar exploatează o garanție a JVM-ului: o clasă este încărcată și inițializată exact o dată, thread-safe implicit, la prima ei utilizare — clasa imbricată `Detinator` nu este încărcată (deci `INSTANTA` nu este creată) decât la primul apel al `obtine()`, iar JVM-ul garantează corectitudinea concurență a acestei încărcări, fără sincronizare explicită scrisă de programator. *Publicarea sigură* — garanția că un obiect complet construit este vizibil în întregime altor thread-uri, nu doar parțial construit — este asigurată automat pentru câmpuri `final` (Subsecțiunea 4.4.3), încă un motiv practic pentru a prefera imutabilitatea în cod

concurrent.

## 14.3 Colecții thread-safe

Am menționat deja, în Subsecțiunea 12.3.4, că nicio colecție obișnuită (ArrayList, HashMap) nu este sigură implicit pentru acces concurrent. Pachetul `java.util.concurrent` oferă alternative gândite explicit pentru acest scop, adesea mai eficiente decât un simplu `Collections.synchronizedX(...)`.

### 14.3.1 BlockingQueue

O `BlockingQueue` extinde `Queue` (Subsecțiunea 11.4.1) cu operații care așteaptă, în loc să eșueze, atunci când coada e plină sau goală — fundația clasică a tiparului *producător-consumator*:

```
1 BlockingQueue<String> tranzactiiDeProcesat = new
   ArrayBlockingQueue<>(100);
2
3 // Thread producator
4 Runnable producator = () -> {
5     while (true) {
6         tranzactiiDeProcesat.put("tranzactie noua");
7         // AȘTEAPTA dacă coada e plină
8     }
9 };
10
11 // Thread consumator
12 Runnable consumator = () -> {
13     while (true) {
14         String tranzactie = tranzactiiDeProcesat.take();
15         // AȘTEAPTA dacă coada e goală
16         System.out.println("Procesez: " + tranzactie);
17     }
18 };
```

Listing 14.9: Producător-consumator, cu `BlockingQueue`

#### Notă

`put()` și `take()` blochează automat thread-ul apelant atunci când e necesar (coadă plină, respectiv goală), fără a necesita nicio sincronizare scrisă manual de programator — `BlockingQueue` gestionează intern toată coordonarea producător-consumator, un tipar altfel notoriu de dificil de implementat corect de la zero, cu `wait/notify` manuale.

### 14.3.2 ConcurrentHashMap

`ConcurrentHashMap` este o alternativă thread-safe la `HashMap` (Subsecțiunea 12.1.1), proiectată pentru performanță ridicată sub concurență, folosind sincronizare la granulație fină internă, nu un singur *lock* global:

```
1 ConcurrentHashMap<String, ContBancar> bancaConcurenta = new
   ConcurrentHashMap<>();
2
3 bancaConcurenta.put("Ana Popescu", new ContBancar("Ana
   Popescu", 500.0));
4
```

Listing 14.10: `ConcurrentHashMap`, cu operații atomice compuse

```
5 bancaConcurrenta.computeIfAbsent("Ion Ionescu", titular -> new
    ContBancar(titular, 0.0));
```

#### Notă

Față de `Collections.synchronizedMap(new HashMap<>())` (Subsecțiunea 12.3.4), pe un `Map` sincronizat „din exterior”, o secvență `if (!map.containsKey(k)) map.put(k, v);` rămâne o condiție de cursă — fiecare apel individual e sincronizat, dar **combinația** celor două nu este atomică. Pe `ConcurrentHashMap`, metode precum `computeIfAbsent` (Subsecțiunea 12.1.2) sunt garantate atomice **ca întreg**, nu doar operațiile individuale — exact ce lipsea din varianta sincronizată manual.

### 14.3.3 Colecții copy-on-write

O `CopyOnWriteArrayList` creează, la **fiecare** operație de scriere, o copie complet nouă a array-ului intern — costisitor la scriere, dar extrem de eficient și sigur la citire concurentă:

**Listing 14.11:** `CopyOnWriteArrayList`, sigură la iterare concurentă cu modificări

```
1 List<ContBancar> observatoriActivi = new CopyOnWriteArrayList
    <>();
2
3 // Un thread poate itera in siguranta...
4 for (ContBancar cont : observatoriActivi) {
5     System.out.println(cont);
6 }
7
8 // ...in timp ce alt thread modifica lista, FARA
9 // ConcurrentModificationException
10 observatoriActivi.add(new ContBancar("Client nou", 0.0));
```

#### Notă

Spre deosebire de o `ArrayList` obișnuită (unde modificarea în timpul iterării aruncă `ConcurrentModificationException`, Subsecțiunea 11.1.2), un iterator `CopyOnWriteArrayList` lucrează pe o *instantanee* (*snapshot*) a array-ului, luată la momentul creării iteratorului — complet imună la modificări concurente, dar acest lucru înseamnă și că iteratorul **nu** va vedea elementele adăugate după ce a fost creat. Potrivit pentru colecții citite frecvent, dar modificate rar (de exemplu, o listă de observatori/listeners înregistrați).

## 14.4 *CompletableFuture*; procese

### 14.4.1 *CompletableFuture* de bază

`CompletableFuture<T>` extinde `Future<T>` (Subsecțiunea 13.3.1) cu un stil **neblocant**: în loc să apelezi `get()` și să așteptai, atașai callback-uri care se execută automat când rezultatul devine disponibil:

**Listing 14.12:** `CompletableFuture`, cu procesare neblocantă a rezultatului

```
1 CompletableFuture<Double> viitorSold = CompletableFuture.
    supplyAsync(() -> {
2     return calculeazaSoldTotal(conturi);
3     // executat asincron, pe un thread din pool-ul implicit
4 });
5
```

```

6 viitorSold.thenAccept(sold -> {
7     System.out.println("Soldul total este: " + sold);
8     // NU blocheaza thread-ul apelant
9 });

```

#### Notă

Spre deosebire de `Future.get()` (Subsecțiunea 13.3.1), care blochează thread-ul apelant până la finalizare, `thenAccept` (și restul metodelor `then...`) **înregistrează** o acțiune care rulează automat, ulterior, când rezultatul e gata — codul care urmează `thenAccept` continuă imediat, fără să aștepte. Acest stil, numit programare *reactivă* sau *orientată pe callback-uri*, este esențial pentru aplicații care nu își pot permite să blocheze thread-uri în așteptare (de exemplu, servere care deserveșc mii de cereri concurente).

### 14.4.2 Compunerea de future-uri

Mai multe operații asincrone pot fi **înlănțuite** sau **combinate**, construind pipeline-uri complexe fără niciun `get()` blocant intermediar:

```

1 CompletableFuture<Double> sold = CompletableFuture.
   supplyAsync(() -> calculeazaSoldTotal(conturi));
2
3 CompletableFuture<String> soldFormatat = sold.thenApply(s ->
   String.format("%.2f lei", s));
4
5 CompletableFuture<Double> dobanda = CompletableFuture.
   supplyAsync(() -> calculeazaDobandaTotala(conturi));
6
7 CompletableFuture<Double> soldPlusDobanda = sold.thenCombine(
   dobanda, (s, d) -> s + d);

```

Listing 14.13: Înlănțuirea (`thenApply`) și combinarea (`thenCombine`) de future-uri

Tabela 14.1: Metodele de compunere cele mai frecvente ale `CompletableFuture`.

| Metodă                                       | Rol                                                                    |
|----------------------------------------------|------------------------------------------------------------------------|
| <code>thenApply(funcție)</code>              | transformă rezultatul, returnând un nou <code>CompletableFuture</code> |
| <code>thenAccept(consumer)</code>            | consumă rezultatul, fără a returna o nouă valoare                      |
| <code>thenCompose(funcție)</code>            | înlănțuie un <b>alt</b> calcul asincron, dependent de primul           |
| <code>thenCombine(altFuture, funcție)</code> | combină rezultatele a <b>două</b> calcule independente                 |

#### Notă

`thenApply` este analogul asincron al lui `Function` (Subsecțiunea 7.2.3) — transformă rezultatul, fără să introducă o nouă operație asincronă independentă. `thenCompose` este necesar atunci când transformarea ea **însăși** returnează un `CompletableFuture` (de exemplu, un al doilea apel asincron care depinde de rezultatul primului); folosirea greșită a lui `thenApply` în acest caz ar produce un `CompletableFuture<CompletableFuture<T>>` imbricat incomod, exact tiparul de *flatMap* deja întâlnit conceptual la `Optional` și pe care îl vom regăsi, cu alt nume, la Stream-uri, în Capitolul 15.

### 14.4.3 Procese externe (ProcessBuilder)

Deși nu ține strict de concurență în interiorul JVM-ului, *ProcessBuilder* este menționat aici fiindcă lansează un proces **extern**, care rulează, la rândul lui, concurrent cu programul Java:

**Listing 14.14:** Lansarea unui proces extern, cu *ProcessBuilder*

```
1 ProcessBuilder pb = new ProcessBuilder("ls", "-la");
2 pb.redirectErrorStream(true); // combina stdout si stderr
3
4 Process proces = pb.start();
5 int codRezultat = proces.waitFor(); // asteapta terminarea
6                                     // procesului extern
```

#### Notă

`waitFor()` blochează thread-ul curent până când procesul extern se termină, similar cu `Thread.join()` (Subsecțiunea 13.2.1) sau cu `Future.get()` — este posibilă și o formă asincronă, `proces.onExit()`, care returnează un `CompletableFuture<Process>`, integrându-se natural cu stilul de compunere discutat mai sus.

#### Rezumat

- ▶ O *condiție de cursă* apare când rezultatul depinde de ordinea imprevizibilă de întretesere a thread-urilor pe date partajate; `synchronized` (metodă sau bloc) achiziționează monitorul unui obiect, garantând acces exclusiv.
- ▶ Un *deadlock* apare când thread-uri așteaptă circular *lock*-uri deținute reciproc; o ordine consecventă de achiziție sau `Lock.tryLock` cu timeout previn acest risc.
- ▶ volatile garantează doar vizibilitate, nu atomicitate pentru operații compuse; tipurile atomice (`AtomicLong` etc.) oferă operații compuse atomice fără sincronizare explicită.
- ▶ `BlockingQueue`, `ConcurrentHashMap` și colecțiile *copy-on-write* sunt alternative thread-safe specializate, adesea mai eficiente decât o colecție obișnuită înfășurată în `Collections.synchronizedX`.
- ▶ `CompletableFuture` oferă un stil neblokant, bazat pe callback-uri (`thenApply`, `thenAccept`, `thenCompose`, `thenCombine`), pentru compunerea calculelor asincrone.
- ▶ `ProcessBuilder` lansează procese externe, care rulează concurrent cu programul Java.

#### Exerciții

1. Reproduceți condiția de cursă din Subsecțiunea 14.1.1, apoi corectați-o folosind `synchronized`, apoi din nou folosind `AtomicLong` (pentru un contor simplu, nu pentru sold), și comparați rezultatele.
2. Scrieți un exemplu minimal de deadlock, cu două obiecte și două thread-uri care le blochează în ordine opusă, apoi corectați-l printr-o ordine consecventă de achiziție a *lock*-urilor.
3. Implementați un tipar producător-consumator simplu, cu un thread care generează numere și le pune într-o `BlockingQueue`, și un alt thread care le extrage și le afișează.
4. Scrieți un mic pipeline cu `CompletableFuture`: calculați asincron soldul total al unei liste de conturi, apoi (prin `thenApply`)

formatați rezultatul ca șir de caractere, apoi (prin `thenAccept`) afișați-l.

5. Explicați, în cuvinte proprii, de ce `computeIfAbsent` pe un `ConcurrentHashMap` este sigur la concurență, în timp ce o secvență echivalentă `containsKey + put`, chiar pe un `Map` sincronizat, nu este.



**Streams, intrare/ieșire, rețea și baze de  
date**



## Introducere

Am procesat colecții, pe tot parcursul capitolelor 11–12, prin bucle explicite: `for-each`, condiții, acumulate. API-ul *Stream*, introdus în Java 8, oferă un stil alternativ, *declarativ* [2]: descriem ce transformare dorim, nu **cum** anume se parcurge colecția pas cu pas. Rezultatul este adesea cod mai scurt, mai ușor de citit ca o singură propoziție logică, și — așa cum vom vedea în Capitolul 16 — paralelizabil aproape gratuit.

## 15.1 De la iterare la stream-uri

### 15.1.1 Motivație: stil imperativ versus declarativ

Reluăm o sarcină simplă — suma soldurilor conturilor cu sold peste 500 — scrisă întâi în stilul deja familiar:

```
1 double total = 0;
2 for (ContBancar cont : conturi) {
3     if (cont.getSold() > 500.0) {
4         total += cont.getSold();
5     }
6 }
```

```
1 double total = conturi.stream()
2     .filter(cont -> cont.getSold() > 500.0)
3     .mapToDouble(ContBancar::getSold)
4     .sum();
```

**Listing 15.1:** Stil imperativ: bucla explicită descrie pas cu pas CUM se calculează

**Listing 15.2:** Aceeași sarcină, cu un stream — descrie CE se dorește, nu cum se parcurge

## Notă

Cele două versiuni calculează exact același rezultat, dar diferă fundamental ca stil: prima specifică explicit variabila acumulator, condiția de test și actualizarea ei, pas cu pas; a doua descrie transformarea ca o *conductă* (*pipeline*) de operații — filtrare, apoi extragere a soldului, apoi însumare — fiecare exprimată independent, fără a gestiona manual starea intermediară. Vom detalia fiecare dintre aceste operații (`filter`, `mapToDouble`, `sum`) pe parcursul acestui capitol.

### 15.1.2 Crearea stream-urilor

Cea mai frecventă sursă a unui stream este o colecție existentă, prin metoda `stream()` moștenită din interfața `Collection` (Subsecțiunea 11.1.1):

```
1 List<ContBancar> conturi = new ArrayList<>();
2 Stream<ContBancar> streamConturi = conturi.stream();
```

**Listing 15.3:** Crearea unui stream dintr-o colecție existentă

Alte surse frecvente de stream-uri:

```
1 Stream<String> dinValori = Stream.of("Ana", "Ion", "Maria");
2 // direct din valori literale
3
```

**Listing 15.4:** Alte moduri de a crea un stream

```

4 | int[] valori = { 10, 20, 30 };
5 | IntStream dinArray = Arrays.stream(valori);
6 |     // dintr-un array (aici, de tipuri primitive)
7 |
8 | Stream<Integer> infinit = Stream.iterate(1, n -> n * 2);
9 |     // stream INFINIT - necesita limit()!

```

#### Notă

Stream.iterate generează un stream teoretic infinit, aplicând repetat funcția dată — **trebuie** combinat cu limit(n) (Subsecțiunea 15.2.4) înainte de orice operație care ar încerca să parcurgă tot stream-ul, altfel programul nu se mai termină. Un stream, odată **consumat** printr-o operație terminală (ca sum() din exemplul anterior), **nu poate fi refolosit** — o a doua încercare de a-l parcurge aruncă IllegalStateException; dacă aveți nevoie să parcurgeți aceleași date de mai multe ori, creați un nou stream din colecția sursă de fiecare dată.

## 15.2 filter, map, flatMap

### 15.2.1 filter

filter păstrează doar elementele care satisfac un Predicate (Subsecțiunea 7.2.3):

Listing 15.5: filter, cu un Predicate exprimat ca lambda

```

1 | List<ContBancar> conturiMari = conturi.stream()
2 |     .filter(cont -> cont.getSold() > 1000.0)
3 |     .collect(Collectors.toList());
4 |     // vezi Cap. 16 pentru detalii despre collect()

```

### 15.2.2 map

map transformă fiecare element, aplicând un Function (Subsecțiunea 7.2.3), producând un stream de alt tip:

Listing 15.6: map, cu o referință la metodă, pentru extragerea titularilor

```

1 | List<String> titulari = conturi.stream()
2 |     .map(ContBancar::getTitular) // referinta la metoda,
3 |                                     // din Sec. 7.3.1
4 |     .collect(Collectors.toList());

```

#### Notă

filter păstrează tipul stream-ului (Stream<ContBancar> rămâne Stream<ContBancar>), doar reduce numărul de elemente; map schimbă tipul (Stream<ContBancar> devine Stream<String> în exemplul de mai sus), păstrând numărul de elemente neschimbat — fiecare element de intrare produce exact un element de ieșire.

### 15.2.3 flatMap

flatMap rezolvă un caz pe care map singur nu îl poate trata elegant: atunci când fiecare element se transformă, la rândul lui, într-un **stream** de elemente (nu într-o singură valoare), care trebuie „aplatizat” într-un singur stream continuu:

```

1 List<ContBancar.Extras> toateTranzactiile = conturi.stream()
2   .flatMap(
3     cont -> cont.getIstoricTranzactii().stream()
4   ) // Stream<Stream<Extras>> - aplatizat
5   .collect(Collectors.toList());
6 // rezultat: Stream<Extras>, o singura lista continua

```

**Listing 15.7:** flatMap, pentru aplatizarea istoricului de tranzacții al tuturor conturilor

#### Notă

Fără flatMap, un map echivalent ar produce un Stream<List<Extras>> — o listă de liste, una per cont, incomodă de procesat mai departe (ar trebui o a doua parcurgere pentru a le combina). flatMap combină, într-un singur pas, transformarea și aplatizarea, exact tiparul *flatMap* deja întâlnit conceptual la CompletableFuture (Subsecțiunea 14.4.2) și pe care îl vom regăsi identic la Optional, în Secțiunea 15.4.

### 15.2.4 Alte transformări

```

1 List<String> primiiTreiTitulariUnici = conturi.stream()
2   .map(ContBancar::getTitular)
3   .distinct() // elimina duplicatele (foloseste equals)
4   .sorted() // ordine naturala (Comparable, Sec. 11.3.2)
5   .skip(0) // sare peste primele 0 elemente
6   .limit(3) // pastreaza doar primele 3 ramase
7   .collect(Collectors.toList());

```

**Listing 15.8:** distinct, sorted, limit, skip — combinate într-un singur pipeline

#### Notă

Ordinea operațiilor într-un pipeline contează pentru rezultat (sorted().limit(3) nu e totuna cu limit(3).sorted()) și poate conta semnificativ pentru **performanță** — un filter plasat devreme reduce numărul de elemente procesate de operațiile ulterioare, mai costisitor de aplicat abia la final. peek(actiune), o operație rar necesară în cod de producție, permite „inspectarea” elementelor pe măsură ce trec prin pipeline, utilă strict pentru depanare (Subsecțiunea 9.4.5).

## 15.3 Reduceri simple

### 15.3.1 forEach, count, min/max

forEach este o operație *terminală* (încheie pipeline-ul, nu returnează un nou stream) care aplică o acțiune fiecărui element, pentru efectul ei secundar:

```

1 conturi.stream()
2   .filter(cont -> cont.getSold() > 500.0)
3   .forEach(System.out::println);
4 // efect secundar - afisare, nu transformare
5
6 long numarConturiMari = conturi.stream()
7   .filter(cont -> cont.getSold() > 500.0)
8   .count();
9
10 Optional<ContBancar> contMaxim = conturi.stream()

```

**Listing 15.9:** forEach, count și max

```
11 | .max(Comparator.comparingDouble(ContBancar::getSold));
```

#### Notă

Observați că `max` (la fel ca `min`) returnează un `Optional<ContBancar>`, nu direct un `ContBancar` — dacă stream-ul e gol, nu există un maxim, iar `Optional` exprimă explicit această posibilitate, în locul unui `null` tăcut. Vom detalia riguros `Optional` în Secțiunea 15.4, imediat după această secțiune.

### 15.3.2 reduce

`reduce` este forma cea mai generală de agregare — combină toate elementele într-un singur rezultat, printr-o funcție binară aplicată succesiv:

**Listing 15.10:** `reduce`, cu valoare identitate explicită

```
1 | double totalReduce = conturi.stream()
2 |   .mapToDouble(ContBancar::getSold)
3 |   .reduce(0.0, Double::sum); // 0.0 = valoarea de start;
4 |   // Double::sum = cum se combina doua valori
```

**Listing 15.11:** `reduce`, fără identitate — rezultatul e un `Optional`

```
1 | Optional<ContBancar> celMaiBogat = conturi.stream()
2 |   .reduce(
3 |     (c1, c2) -> c1.getSold() >= c2.getSold() ? c1 : c2
4 |   );
```

#### Notă

Forma cu **identitate** explicită (primul exemplu) garantează un rezultat chiar și pentru un stream gol (returnează identitatea, aici `0.0`), motiv pentru care returnează direct `double`, nu `Optional<Double>`. Forma **fără identitate** (al doilea exemplu) nu are nicio valoare de rezervă pentru un stream gol, deci returnează `Optional<T>` — exact aceeași distincție întâlnită deja la `min/max`. O a treia formă, cu trei argumente (identitate, acumulator, combinator), este necesară pentru stream-uri paralele, subiect pe care îl reluăm în Capitolul 16.

## 15.4 Tipul `Optional`

### 15.4.1 Obținerea și consumarea valorilor

#### Definiție — `Optional`

`Optional<T>` este un container care fie conține o valoare de tip `T`, fie este gol — o alternativă explicită, tipizată, la returnarea unui `null` tăcut din Subsecțiunea 4.3.2 pentru a semnala „nicio valoare”.

**Listing 15.12:** Crearea și interogarea unui `Optional`

```
1 | Optional<ContBancar> gasit = conturi.stream()
2 |   .filter(cont -> cont.getTitular().equals("Ana Popescu"))
3 |   .findFirst(); // Optional - poate sa nu existe niciun
4 |                 // cont cu acest titular
5 |
6 | if (gasit.isPresent()) {
7 |   System.out.println(gasit.get().getSold());
8 | }
```

```

1 double sold = gasit.map(ContBancar::getSold).orElse(0.0);
2 // valoare implicita, daca lipseste
3
4 gasit.ifPresent(cont -> System.out.println("Gasit " + cont));
5 // actiune, doar daca exista
6
7 ContBancar cont = gasit.orElseThrow(() -> new
  NoSuchElementException("Cont inexistent"));

```

**Listing 15.13:** Consumarea sigură, în stil funcțional, fără `isPresent/get` explicit

#### Notă

`orElse(valoareImplicita)` evaluează întotdeauna argumentul, chiar dacă `Optional` conține deja o valoare — pentru o valoare implicită costisitor de calculat, preferați `orElseGet(supplier)`, care o calculează doar dacă e cu adevărat necesară, un `Supplier` (Subsecțiunea 7.2.3) evaluat leneș.

### 15.4.2 Pipelining cu Optional

`Optional` oferă propriile `map/filter`, permițând transformări înlănțuite fără a despacheta valoarea prematur:

```

1 String rezultat = conturi.stream()
2   .filter(cont -> cont.getTitular().equals("Ana Popescu"))
3   .findFirst()
4   .filter(cont -> cont.getSold() > 100.0)
5   // Optional ramane gol daca filtrul esueaza
6   .map(ContBancar::getTitular)
7   .map(String::toUpperCase)
8   .orElse("CONT NEGASIT SAU SOLD INSUFICIENT");

```

**Listing 15.14:** Transformări înlănțuite pe `Optional`, fără verificări explicite intermediare

#### Notă

Fiecare `map/filter` din lanț se aplică doar dacă `Optional`-ul curent conține o valoare — un `Optional` gol rămâne gol prin tot restul lanțului, fără nicio verificare explicită la fiecare pas. Acest stil este direct analog pipeline-ului de `Stream` discutat în Secțiunea 15.2: `Optional` se comportă, conceptual, ca un stream cu cel mult un element.

### 15.4.3 Anti-pattern-uri

#### Notă

Câteva utilizări greșite, frecvente la prima întâlnire cu `Optional`:

- ▶ `optional.get()` fără verificare prealabilă — reintroduce exact riscul pe care `Optional` încearcă să-l elimine (`NoSuchElementException` în loc de `NullPointerException`, dar la fel de nesigur).
- ▶ `if (optional.isPresent()) { optional.get()... }` — funcțional corect, dar mai verbos și mai puțin idiomatic decât `map/ifPresent/orElse`, care exprimă direct intenția.
- ▶ `Optional` ca tip de câmp sau parametru de metodă — `Optional` a fost proiectat pentru tipuri de retur, semnalând apelantului o posibilă absență a rezultatului; folosit ca tip de câmp,

adaugă complexitate (necesită despachetare la fiecare acces) fără beneficiu real față de o simplă verificare null, iar folosit ca parametru, mută responsabilitatea de a gestiona absența către interiorul metodei, într-un mod neclar pentru apelant.

- **Optional.of(valoare)** cu o valoare care ar putea fi null — aruncă imediat `NullPointerException`; pentru o valoare potențial null, folosiți `Optional.ofNullable(valoare)`.

#### 15.4.4 flatMap cu Optional

La fel ca la Stream (Subsecțiunea 15.2.3), un map aplicat unei funcții care returnează ea însăși un Optional produce o imbricare incomodă, de tipul `Optional<Optional<T>>`:

Listing 15.15: Fără flatMap: imbricare incomodă de Optional

```
1 public Optional<String> gasesteAdresaTitular(ContBancar cont)
2     {
3         // returneaza Optional<String> - adresa poate lipsi
4         // din baza de date
5     }
6 Optional<Optional<String>> imbricat = gasit.map(this::
7     gasesteAdresaTitular);
8     // incomod!
```

Listing 15.16: Cu flatMap: aplatizare automată, un singur nivel de Optional

```
1 Optional<String> adresa = gasit.flatMap(
2     this::gasesteAdresaTitular
3 ); // Optional<String>, curat
```

#### Notă

Regula practică, identică celei de la Stream (Subsecțiunea 15.2.3) și de la `CompletableFuture` (Subsecțiunea 14.4.2): folosiți `map` atunci când funcția de transformare returnează o valoare simplă; folosiți `flatMap` atunci când funcția de transformare returnează, ea însăși, un `Optional` (sau `Stream`, sau `CompletableFuture`) — același tipar *monadic*, sub trei nume diferite, regăsit consecvent în bibliotecă.

#### Rezumat

- Stream-urile oferă un stil declarativ de procesare a colecțiilor: `filter` reduce, `map` transformă, `flatMap` aplatizează transformări care produc, ele însele, stream-uri.
- Un stream se consumă o singură dată; operațiile terminale (`forEach`, `count`, `reduce`, `collect`) declanșează efectiv procesarea, cele intermediare (`filter`, `map`) doar descriu pipeline-ul.
- `reduce` este forma generală de agregare; cu identitate explicită garantează un rezultat direct, fără identitate returnează `Optional`.
- `Optional<T>` exprimă explicit posibila absență a unei valori, mai ales ca tip de retur; `map/filter/orElse` permit consumare sigură, fără verificări `isPresent/get` explicite.
- `flatMap` rezolvă, identic la `Stream`, `Optional` și `CompletableFuture`, problema imbricării atunci când o transformare produce, ea însăși, un container de același tip.

**Exerciții**

1. Rescrieți, folosind un stream, o metodă care primește o listă de `ContBancar` și returnează lista titularilor (fără duplicate), ordonată alfabetic.
2. Folosind `flatMap`, scrieți o metodă care primește o listă de conturi și returnează suma tuturor sumelor din istoricul de tranzacții (`Extras`) al tuturor conturilor.
3. Scrieți o metodă `Optional<ContBancar> gasesteDupaTitular(List<ContBancar> conturi, String titular)`, folosind `stream().filter().findFirst()`.
4. Folosind `reduce` (fără `mapToDouble/sum`), calculați soldul total al unei liste de conturi, pornind de la o identitate `0.0`.
5. Scrieți un mic lanț de transformări pe un `Optional<ContBancar>` (obținut printr-o căutare), care extrage titularul, îl transformă cu majuscule, și oferă o valoare implicită dacă niciun cont nu a fost găsit — fără niciun `if/isEmpty` explicit.



# Streams (II): Collectors. Gatherers. Stream-uri paralele

# 16

## Introducere

Am folosit deja `Collectors.toList()` de câteva ori în capitolul anterior, fără să insistăm asupra lui. `Collectors` este, de fapt, un instrument mult mai bogat — capabil să colecteze un stream direct într-un `Map`, să grupeze sau să partiționeze elementele, sau să calculeze simultan mai multe agregări. Capitolul se încheie cu *Gatherers* (o extensie recentă pentru transformări intermediare personalizate), stream-uri pe tipuri primitive, și paralelizarea stream-urilor.

## 16.1 Collectors: colectare în colecții și Map

### 16.1.1 collect(), toList/toSet

`collect` este operația terminală care „adună” elementele unui stream într-o structură de date concretă, ghidată de un `Collector` — cel mai adesea, unul dintre cei predefiniți în clasa utilitară `Collectors`:

```
1 List<String> titulari = conturi.stream()
2   .map(ContBancar::getTitular)
3   .collect(Collectors.toList());
4
5 Set<String> titulariUnici = conturi.stream()
6   .map(ContBancar::getTitular)
7   .collect(Collectors.toSet());
8
9 List<String> imutabila = conturi.stream()
10  .map(ContBancar::getTitular)
11  .collect(Collectors.toUnmodifiableList());
12 // rezultat imutabil, ca List.of din Sec. 12.3.1
```

**Listing 16.1:** Colectare de bază: listă, mulțime, listă imutabilă

## Notă

Începând cu Java 16, metodele scurte `stream.toList()` (fără `Collectors`.) oferă un echivalent direct pentru cazul cel mai comun — o listă imutabilă — fără a mai necesita apelul explicit `collect(Collectors.toList())`. Rămâne totuși util să cunoașteți forma completă cu `Collectors`, necesară pentru toate celelalte variante din acest capitol.

### 16.1.2 Colectare într-un Map

`Collectors.toMap` construiește direct un `Map`, dintr-un stream, specificând cum se extrage cheia și cum se extrage valoarea din fiecare element:

```
1 Map<String, Double> soldPeTitular = conturi.stream()
2   .collect(Collectors.toMap(
3       ContBancar::getTitular,
4       ContBancar::getSold
5   ));
```

**Listing 16.2:** Colectare directă într-un Map, din stream

**Listing 16.3:** toMap cu funcție de combinare, pentru chei duplicate

```

1 Map<String, Double> soldTotalPeTitular = conturi.stream()
2   .collect(Collectors.toMap(
3       ContBancar::getTitular,
4       ContBancar::getSold,
5       Double::sum // ce se intampla daca doi titulari au
6                   // NUME identic - se ADUNA soldurile
7   ));

```

**Notă**

Fără o funcție de combinare explicită (primul exemplu), toMap aruncă `IllegalStateException` dacă două elemente produc **aceeași** cheie — comportament intenționat de siguranță, care previne o pierdere silențioasă de date. Al treilea argument (al doilea exemplu) rezolvă exact această situație, la fel ca `Map.merge` deja folosit direct în Subsecțiunea 12.1.2: aici, toMap aplică intern acea logică de combinare automat, pentru fiecare coliziune de chei întâlnită în timpul colectării.

## 16.2 Grupare, partiționare, downstream collectors

### 16.2.1 groupingBy

`groupingBy` grupează elementele unui stream după o cheie derivată, produsă printr-o funcție de clasificare, rezultatul fiind de tipul `Map<Cheie, List<Element>>`:

**Listing 16.4:** Gruparea tuturor tranzacțiilor, din toate conturile, după tip

```

1 Map<String, List<ContBancar.Extras>> tranzactiiPeTip =
2   conturi.stream()
3   .flatMap(cont -> cont.getIstoricTranzactii().stream())
4   // aplatizare, din Sec. 15.2.3
5   .collect(Collectors.groupingBy(ContBancar.Extras::tip));
6 List<ContBancar.Extras> toateDepunerile = tranzactiiPeTip.get(
7   "depunere");

```

**Notă**

Rezultatul unui `groupingBy` simplu, ca mai sus, este întotdeauna un `Map` către `List` — fiecare grup conține **toate** elementele originale care au produs acea cheie, nepierzând nicio informație. Vom vedea în Subsecțiunea 16.2.3 cum acest comportament implicit (colectare în listă) poate fi înlocuit cu orice altă agregare, printr-un *downstream collector*.

### 16.2.2 partitioningBy

`partitioningBy` este un caz special de grupare, după o condiție booleană — rezultatul are **întotdeauna** exact două chei, `true` și `false`, chiar dacă unul dintre grupuri e gol:

**Listing 16.5:** Partiționarea conturilor, după un prag de sold

```

1 Map<Boolean, List<ContBancar>> partitionate = conturi.stream()
2   .collect(Collectors.partitioningBy(
3       cont -> cont.getSold() > 1000.0
4   ));

```

```

5
6 List<ContBancar> conturiMari = partitionate.get(true);
7 List<ContBancar> conturiMici = partitionate.get(false);

```

#### Notă

Spre deosebire de `groupBy` (care poate produce oricâte chei, în funcție de datele reale), `partitioningBy` garantează structural exact două grupuri — util atunci când condiția de clasificare este, prin natura ei, binară, și codul care urmează are nevoie sigur de ambele ramuri (spre deosebire de `groupBy`, unde o cheie absentă din date pur și simplu nu apare în Map-ul rezultat).

### 16.2.3 Downstream collectors

Al doilea argument al lui `groupBy` (opțional) înlocuiește colectarea implicită în listă cu orice altă agregare — un *downstream collector*:

```

1 Map<String, Long> numarTranzactiiPeTip = conturi.stream()
2   .flatMap(cont -> cont.getIstoricTranzactii().stream())
3   .collect(Collectors.groupingBy(
4     ContBancar.Extras::tip,
5     Collectors.counting()
6   ));
7
8 Map<String, Double> totalPeTip = conturi.stream()
9   .flatMap(cont -> cont.getIstoricTranzactii().stream())
10  .collect(Collectors.groupingBy(
11    ContBancar.Extras::tip,
12    Collectors.summingDouble(ContBancar.Extras::suma)
13  ));
14
15 String toateNumele = conturi.stream()
16   .map(ContBancar::getTitular)
17   .collect(Collectors.joining(", ", "[", "]"));
18   // separator, prefix, sufix

```

**Listing 16.6:** `groupBy` cu downstream collectors: numărare, sumare, concatenare

Tabela 16.1: Downstream collectors frecvent combinate cu `groupBy`.

| Downstream collector                                | Rol                                                     |
|-----------------------------------------------------|---------------------------------------------------------|
| <code>counting()</code>                             | numără elementele fiecărui grup                         |
| <code>summingDouble(funcție)/summingInt(...)</code> | însumează o valoare extrasă din fiecare element         |
| <code>averagingDouble(funcție)</code>               | calculează media aceleiași valori                       |
| <code>joining(sep, prefix, sufix)</code>            | concatenează șiruri, cu separator/prefix/sufix opțional |
| <code>mapping(funcție, collector)</code>            | transformă elementele, apoi aplică alt collector        |

#### Notă

Downstream collectors se pot **combina** recursiv — de exemplu, `groupBy(cheie, mapping(funcție, joining(", ")))` extrage o valoare din fiecare element al fiecărui grup, apoi le concatenează, într-un singur pas. Această compunere este exact ce face `Collectors` un instrument atât de flexibil, evitând nevoia de a scrie bucle manuale pentru agregări complexe pe grupuri.

### 16.2.4 Collectors proprii (context)

Pentru agregări care nu se potrivesc niciunei combinații predefinite, `Collector.of(...)` permite definirea unui collector complet personalizat, specificând un `Supplier` (creează containerul inițial), un `accumulator` (adaugă un element), un `combiner` (combină două containere parțiale, necesar pentru paralelizare) și, opțional, un `finisher` (transformă containerul final în rezultat).

#### Notă

Definirea unui collector propriu este rar necesară în practică — combinația de `groupingBy/partitioningBy` cu downstream collectors adecvate (inclusiv `mapping`, care poate „glisa” aproape orice transformare suplimentară) acoperă marea majoritate a nevoilor reale. Menționăm posibilitatea aici mai ales pentru completitudine, dacă o veți întâlni în cod sau documentație de bibliotecă.

## 16.3 Gatherers

### 16.3.1 Motivație și gatherers predefiniți

`filter`, `map` și `flatMap` (Secțiunea 15.2) au o limitare comună: fiecare procesează elementele **independent**, unul câte unul, fără memorie a elementelor anterioare. *Gatherers*, o extensie recentă a API-ului Stream, permite operații intermediare **cu stare**, precum ferestre glisante sau totaluri cumulative:

**Listing 16.7:** `Gatherers.windowFixed`, pentru gruparea în loturi de dimensiune fixă

```
1 List<List<ContBancar.Extras>> loturi = tranzactii.stream()
2   .gather(Gatherers.windowFixed(10)) // grupeaza in loturi
3                                     // de cate 10 tranzactii
4   .collect(Collectors.toList());
```

**Listing 16.8:** `Gatherers.scan`, pentru un sold cumulativ (rulant), pe măsura tranzacțiilor

```
1 List<Double> soldRulant = tranzactii.stream()
2   .gather(Gatherers.scan(
3     () -> 0.0, // valoare initiala
4     (soldAnterior, tranzactie)
5       -> soldAnterior + tranzactie.suma()
6   ))
7   .collect(Collectors.toList());
```

#### Notă

`Gatherers.scan` este conceptual apropiat de `reduce` (Subsecțiunea 15.3.2), dar cu o diferență esențială: `reduce` este o operație **terminală**, care produce un singur rezultat final; `gather(Gatherers.scan(...))` este o operație **intermediară**, care produce un **nou stream**, cu câte un rezultat cumulativ după fiecare element — exact soldul rulant al contului, tranzacție cu tranzacție, util pentru afișarea unui extras de cont complet, nu doar a soldului final.

### 16.3.2 Implementare proprie (context)

Pentru transformări **cu stare** care nu se potrivesc gatherer-elor predefinite, `Gatherer.of(...)` permite definirea uneia proprii, specificând starea inițială și modul în care fiecare element o actualizează — structural

analog collector-ilor proprii din Subsecțiunea 16.2.4, dar pentru operații intermediare, nu terminale.

#### Notă

La fel ca la collectors proprii, definirea unui gatherer propriu este necesară rar — gatherer-ele predefinite (`windowFixed`, `windowSliding`, `scan`, `fold`, `mapConcurrent`) acoperă majoritatea tiparelor comune de procesare cu stare. Fiind o adăugare relativ recentă la API, recunoașteți-o mai ales pentru a înțelege de ce există — completează exact golul dintre operațiile fără stare (`map/filter`) și cele terminale (`reduce/collect`).

## 16.4 Stream-uri pe tipuri primitive; stream-uri paralele

### 16.4.1 `IntStream`, `DoubleStream`, `LongStream`

Am folosit deja `mapToDouble` în Subsecțiunea 15.1.1, fără explicație detaliată. Un `Stream<Double>` obișnuit ar suferi de costul autoboxing-ului (Subsecțiunea 6.4.2) pentru fiecare element — `DoubleStream`, `IntStream` și `LongStream` evită complet acest cost, lucrând direct cu valori primitive:

```
1 DoubleStream solduri = conturi.stream()
2   .mapToDouble(ContBancar::getSold);
3   // Stream<ContBancar> -> DoubleStream, fara autoboxing
4
5 double suma = solduri.sum();
6
7 IntStream secventa = IntStream.range(1, 11); // 1, 2, ..., 10
8 OptionalDouble medie = secventa.average();
9   // OptionalDouble, nu Optional<Double>
```

**Listing 16.9:** Conversia la un stream de tip primitiv, și operații specifice acestuia

#### Notă

Observați `OptionalDouble` (nu `Optional<Double>`) ca tip returnat de `average()` — consecvent cu evitarea autoboxing-ului, streams-urile primitive au propriile variante specializate ale lui `Optional` (`OptionalInt`, `OptionalLong`, `OptionalDouble`), cu aceeași interfață (`orElse`, `isPresent`) ca varianta generică din Secțiunea 15.4.

### 16.4.2 Stream-uri paralele

Orice stream poate fi executat **paralel**, distribuind procesarea pe mai multe nuclee, printr-un apel `parallel()` sau `collection.parallelStream()` direct:

```
1 double totalParalel = conturi.parallelStream()
2   .mapToDouble(ContBancar::getSold)
3   .sum();
```

**Listing 16.10:** Un stream paralel — aceeași sintaxă, execuție potențial pe mai multe nuclee

**Notă**

Intern, un stream paralel folosește exact framework-ul fork-join din Subsecțiunea 13.4.3 — `ForkJoinPool.commonPool()`, partajat implicit de toate stream-urile paralele ale aplicației. Paralelizarea are sens doar pentru volume de date suficient de mari (costul de coordonare între thread-uri poate depăși ușor beneficiul, pentru colecții mici) și pentru operații **fără efecte secundare** — o lambda pasată la `map/filter` care modifică o stare partajată, mutabilă (exact condiția de cursă din Subsecțiunea 14.1.1), produce rezultate incorecte, imprevizibile, sub execuție paralelă.

**Notă**

Regula practică: măsurați efectiv performanța înainte de a paraleliza un stream de dragul paralelizării — pentru multe cazuri reale, un stream secvențial este la fel de rapid sau mai rapid, din cauza costurilor de coordonare. Paralelizați explicit doar acolo unde se arată un beneficiu real, pe volume de date suficient de mari, cu operații fără stare partajată mutabilă.

**Rezumat**

- ▶ `collect(Collectors.toList()/toSet())` colectează un stream într-o colecție; `toMap` colectează direct într-un `Map`, necesitând o funcție de combinare pentru chei duplicate.
- ▶ `groupBy` grupează după o cheie derivată (oricâte grupuri rezultă din date); `partitioningBy` produce exact două grupuri, după o condiție booleană.
- ▶ Downstream collectors (`counting`, `summingDouble`, `joining`, `mapping`) înlocuiesc colectarea implicită în listă a lui `groupBy` cu orice altă agregare, inclusiv combinate recursiv.
- ▶ `Gatherers` adaugă operații intermediare **cu stare** (ferestre, sume cumulative), completând golul dintre operațiile fără stare (`map/filter`) și cele terminale (`reduce/collect`).
- ▶ `IntStream/DoubleStream/LongStream` evită costul autoboxing-ului pentru date primitive, cu propriile variante `OptionalInt/OptionalDouble/OptionalLong`.
- ▶ Stream-urile paralele (`parallel()/parallelStream()`) folosesc intern fork-join — utile doar pentru volume mari de date și operații fără efecte secundare pe stare partajată mutabilă.

**Exerciții**

1. Folosind `Collectors.toMap` cu funcție de combinare, construiți un `Map<String, Double>` cu soldul total per titular, pornind de la o listă de conturi care poate conține mai multe conturi pentru același titular.
2. Grupați toate tranzacțiile (Extras) ale tuturor conturilor după tip, apoi, folosind un downstream collector, calculați suma totală pentru fiecare tip într-un singur pipeline.
3. Partiționați o listă de conturi în „cu sold pozitiv” și „cu sold zero sau negativ”, apoi afișați numărul de conturi din fiecare partiție.
4. Folosind `Gatherers.scan`, calculați și afișați soldul rulant al unui cont, pornind de la istoricul complet al tranzacțiilor sale.
5. Comparați timpul de execuție al unei însumări de solduri pe o

listă mare de conturi (generată sintetic, de exemplu 10 milioane de elemente), folosind `stream()` versus `parallelStream()`, și discutați rezultatul obținut.



## Introducere

### Notă

O clarificare terminologică necesară chiar de la început: *stream-urile* din acest capitol (`InputStream`, `OutputStream`, `Reader`, `Writer`) **nu** au nicio legătură cu `java.util.stream.Stream` din Capitolul 15–Capitolul 16 — doar numele coincide istoric. Stream-urile I/O modelează un **flux de date** (octeți sau caractere) către/dinspre o sursă externă (fișier, rețea, memorie); stream-urile din Capitolul 15 modelează o **conductă de procesare** pentru colecții de date deja aflate în memorie.

Am folosit deja `Scanner` pentru citire de la tastatură, din Subsecțiunea 3.3.1. Acest capitol tratează I/O riguros: cum se citesc și scriu octeți bruți, cum se gestionează corect textul (inclusiv codificările de caractere), și cum se manipulează fișiere și directoare prin API-ul modern NIO.2.

## 17.1 Stream-uri de octeți

### 17.1.1 `InputStream` și `OutputStream`

`InputStream` și `OutputStream` sunt clasele abstracte de bază pentru citirea, respectiv scrierea, de **octeți bruți** — cea mai elementară formă de I/O, comună tuturor surselor de date (fișiere, rețea, memorie):

```
1 try (OutputStream out = new FileOutputStream("date.bin")) {
2     out.write(65); // scrie un singur octet (aici, codul
3                   // ASCII al literei 'A')
4     out.write(new byte[] { 1, 2, 3 }); // scrie un array de
5                                       // octeti
6 }
7
8 try (InputStream in = new FileInputStream("date.bin")) {
9     int octet = in.read(); // citește UN octet, sau -1 la
10                          // sfârșitul fișierului
11     while (octet != -1) {
12         System.out.println(octet);
13         octet = in.read();
14     }
15 }
```

**Listing 17.1:** Scrierea și citirea de octeți, folosind implementările pentru fișiere

### Notă

Ambele clase implementează `AutoCloseable`, motiv pentru care apar natural în `try-with-resources` (Subsecțiunea 9.3.1) — un fișier deschis, dar neînchis, rămâne blocat pentru alte procese și consumă resurse ale sistemului de operare. `read()` returnează `-1` pentru a semnala sfârșitul fluxului, o convenție folosită consecvent în toată ierarhia `InputStream`.

### 17.1.2 Combinarea filtrelor

Biblioteca Java oferă o multitudine de clase `InputStream/OutputStream` specializate, gândite pentru a fi **combinate**, nu alese individual — fiecare adaugă o singură capabilitate, prin înfășurarea unui alt stream:

Listing 17.2: Combinarea filtrelor: buffering, peste un stream de fișier

```
1 try (InputStream fisierBrut = new FileInputStream("date.in");
2     // adauga bufferare
3     InputStream cuBuffer = new BufferedInputStream(
4     fisierBrut)) {
5
6     int octet = cuBuffer.read(); // citeste, in spate, in
7                                 // blocuri mari, nu octet
8                                 // cu octet
9     // ...
10 }
```

#### Notă

Acest tipar — un constructor care primește un alt stream, ca argument, și îi „adaugă” o capabilitate — este o aplicare directă a compoziției peste moștenire (Secțiunea 6.6, prin analogie): în loc de o clasă separată pentru fiecare combinație posibilă (fișier bufferat, fișier bufferat și comprimat, rețea bufferată etc.), fiecare capabilitate e o clasă independentă, ce poate fi combinată la nevoie. `BufferedInputStream` reduce dramatic numărul de operații de citire fizică (de pe disc), citind date în blocuri mari și servindu-le din memorie — aproape întotdeauna recomandat pentru fișiere, cu excepția citirilor deja bufferate la alt nivel.

## 17.2 Input/output de text; codificări de caractere

### 17.2.1 Reader și Writer

Pentru text (spre deosebire de date binare arbitrare), `Reader` și `Writer` oferă echivalentul orientat pe caractere al lui `InputStream/OutputStream`:

Listing 17.3: Scrierea și citirea de text, cu `BufferedReader/BufferedWriter`

```
1 try (Writer scriitor = new BufferedWriter(new FileWriter("
2     conturi.txt"))) {
3     scriitor.write("Ana Popescu, 500.0");
4     scriitor.newLine();
5     scriitor.write("Ion Ionescu, 1200.0");
6 }
7
8 try (BufferedReader cititor = new BufferedReader(new
9     FileReader("conturi.txt"))) {
10    String linie;
11    while ((linie = cititor.readLine()) != null) {
12        System.out.println(linie);
13    }
14 }
```

#### Notă

`BufferedReader.readLine()` returnează `null` la sfârșitul fișierului (nu `-1`, convenția specifică octeților din Subsecțiunea 17.1.1) —

o sursă frecventă de confuzie între cele două ierarhii paralele, byte-oriented și character-oriented. Tiparul `while ((linie = cititor.readLine()) != null)` de mai sus, cu atribuire **în interiorul** condiției, este idiomul standard pentru această buclă de citire, deși poate părea neobișnuit la prima vedere.

`InputStreamReader/OutputStreamWriter` fac **puntea** între cele două ierarhii, convertind explicit octeți în caractere (și invers), pe baza unei codificări specificate:

```
1 try (Reader cititor = new InputStreamReader(new
2     FileInputStream("date.txt"), StandardCharsets.UTF_8)) {
3     // citește caractere, decodificate explicit din UTF-8
4 }
```

Listing 17.4: Puntea explicită dintre octeți și caractere, cu codificare specificată

## 17.2.2 Codificări de caractere

O *codificare* (*charset*) definește regula de conversie între caractere Unicode (Subsecțiunea 2.2.2) și octeții bruți dintr-un fișier. Aceeași secvență de octeți poate produce text complet diferit, în funcție de codificarea presupusă la citire:

```
1 // FARA codificare explicita - folosește codificarea
2 // IMPLICITA a platformei, RISCANT!
3 Writer scriitorNesigur = new FileWriter("date.txt");
4
5 // CU codificare explicita - comportament identic,
6 // indiferent de platforma
7 Writer scriitorSigur = new FileWriter("date.txt",
8     StandardCharsets.UTF_8);
```

Listing 17.5: Specificarea explicită a codificării, la citire și scriere

### Notă

Codificarea implicită a platformei **diferă** între sisteme de operare și configurații regionale — cod care funcționează corect pe mașina de dezvoltare poate produce text corupt (caractere greșite, cunoscute informal ca *mojibake*) atunci când rulează pe un server cu altă codificare implicită. Specificați **întotdeauna** explicit `StandardCharsets.UTF_8` (sau altă codificare, dacă aveți un motiv specific) pentru orice I/O de text — niciodată nu vă bazați pe codificarea implicită, nedeterminată la citirea codului sursă.

## 17.3 Date binare; fișiere cu acces aleator; arhive ZIP

### 17.3.1 DataInput/DataOutput

`DataOutputStream/DataInputStream`, alte două filtre din „zoo”-ul discutat în Subsecțiunea 17.1.2, permit scrierea/citirea directă a tipurilor primitive, într-un format binar portabil:

```
1 try (DataOutputStream out = new DataOutputStream(
2     new BufferedOutputStream(new FileOutputStream("cont.
3     dat")))) {
4     out.writeUTF("Ana Popescu");// sir de caractere, cu
5     // lungime codificata automat
```

Listing 17.6: Scrierea datelor unui cont, în format binar structurat

```

5      out.writeDouble(500.0);          // 8 octeti, format binar
6                                          // standard IEEE 754
7    }
8
9    try (DataInputStream in = new DataInputStream(
10         new BufferedInputStream(new FileInputStream("cont.dat"
11         ")))) {
12        String titular = in.readUTF(); // TREBUIE citit în
13        aceeași ordine ca la scriere!
14        double sold = in.readDouble();
15    }

```

#### Notă

Ordinea de citire trebuie să corespundă **exact** ordinii de scriere — formatul binar nu conține niciun marcator care să indice ce tip urmează, spre deosebire de text, unde structura e vizibilă. `writeUTF/readUTF` folosesc un format specific Java (lungime precedând textul codificat), **nu** identic cu UTF-8 standard pentru șiruri foarte lungi — suficient de robust pentru comunicare între programe Java, dar nepotrivit pentru interoperare cu alte limbaje.

### 17.3.2 Fișiere cu acces aleator

Toate stream-urile discutate până acum sunt **secvențiale** — citesc/scriu strict de la un capăt către celălalt. `RandomAccessFile` permite *poziționare* explicită, oriunde în fișier:

Listing 17.7: `RandomAccessFile`, cu acces direct la o poziție arbitrară

```

1    try (RandomAccessFile fisier = new RandomAccessFile("conturi.
2        dat", "rw")) {
3        long dimInregistrare = 100; // fiecare cont ocupa exact
4                                          // 100 de octeti
5
6        fisier.seek(3 * dimInregistrare); // sare direct la a
7                                          // 4-a inregistrare
8        String titular = fisier.readUTF();
9        double sold = fisier.readDouble();
10
11        fisier.seek(3 * dimInregistrare); // revine, pentru a
12                                          // suprascrie aceeași
13                                          // inregistrare
14        fisier.writeUTF(titular);
15        fisier.writeDouble(sold + 100.0); // actualizeaza
16                                          // soldul, FARA a
17                                          // rescrie tot fisierul
18    }

```

#### Notă

Acces aleator util necesită, de regulă, înregistrări de **dimensiune fixă** (ca în exemplul de mai sus, 100 de octeți per cont) — altfel, poziția exactă a înregistrării  $N$  nu poate fi calculată direct, ci ar necesita parcurgerea secvențială a celor dinaintea ei, anulând avantajul accesului aleator. Modul "rw" (citire și scriere) contrastează cu stream-urile obișnuite, care sunt implicit fie doar pentru citire, fie doar pentru scriere.

### 17.3.3 Arhive ZIP

ZipOutputStream/ZipInputStream tratează o arhivă ZIP ca o secvență de *intrări* (*entries*), fiecare cu propriul flux de date:

```

1 try (ZipOutputStream zip = new ZipOutputStream(new
2   FileOutputStream("extrase.zip"))) {
3   for (ContBancar cont : conturi) {
4     zip.putNextEntry(new ZipEntry(cont.getTitular() + ".
5     txt"));
6     zip.write(("Sold: " + cont.getSold()).getBytes(
7     StandardCharsets.UTF_8));
8     zip.closeEntry();
9   }
10 }

```

**Listing 17.8:** Crearea unei arhive ZIP, cu mai multe intrări

#### Notă

Fiecare pereche putNextEntry/closeEntry delimitează conținutul unui singur fișier din arhivă — scrierile dintre ele (zip.write(...)) merg către intrarea curent deschisă. Menționăm ZIP-ul aici mai ales pentru completitudine; pentru citirea/scrierea conținutului unui JAR (care este, la urma urmei, tot o arhivă ZIP, Secțiunea 5.2), API-ul NIO.2 din secțiunea următoare oferă adesea o alternativă mai convenabilă, tratând arhiva ca un sistem de fișiere navigabil.

## 17.4 Lucrul cu fișiere (NIO.2)

### 17.4.1 Path și operații de bază

API-ul modern pentru fișiere (NIO.2, din pachetul java.nio.file) reprezintă o cale prin Path, mai flexibil decât vechea clasă File:

```

1 Path cale = Path.of("date", "conturi", "extras.txt");
2
3 System.out.println(cale.getFileName()); // "extras.txt"
4 System.out.println(cale.getParent());   // "date/conturi"
5 System.out.println(cale.toAbsolutePath()); // calea completa,
6 // de la radacina sistemului de fisiere
7
8 // echivalent
9 Path combinata = Path.of("date").resolve("conturi").resolve("
10  extras.txt");

```

**Listing 17.9:** Crearea și manipularea unui Path

### 17.4.2 Citire/scriere directă de text

Pentru fișiere de dimensiune rezonabilă, clasa utilitară Files oferă metode care evită complet gestionarea manuală a stream-urilor din Secțiunea 17.2:

```

1 // tot fisierul, ca un singur String:
2 String continut = Files.readString(Path.of("conturi.txt"));
3 // fisierul, linie cu linie:
4 List<String> linii = Files.readAllLines(Path.of("conturi.txt"
5 ));

```

**Listing 17.10:** Citire și scriere directă, fără Reader/Writer explicite

```
6 Files.writeString(Path.of("rezultat.txt"), "Ana Popescu,
500.0");
```

#### Notă

Aceste metode presupun **implicit** codificarea UTF-8 (spre deosebire de constructorii `FileReader/FileWriter` folosiți în Subsecțiunea 17.2.1, unde codificarea implicită a platformei era un risc real) — comportament sigur și portabil, fără parametri suplimentari necesari. Pentru fișiere mari, unde încărcarea completă în memorie ar fi ineficientă, `Files.newBufferedReader/newBufferedWriter` oferă acces streaming, linie cu linie, la fel ca în Subsecțiunea 17.2.1, dar pornind de la un `Path`.

### 17.4.3 Creare, copiere, mutare, ștergere

Listing 17.11: Operații de bază asupra fișierelor și directoarelor, prin `Files`

```
1 Files.createDirectories(Path.of("date", "arhiva"));
2 // creeaza si directoarele intermediare lipsa
3 Files.createFile(Path.of("date", "gol.txt"));
4
5 Files.copy(Path.of("conturi.txt"), Path.of("date", "arhiva",
6 "conturi_copie.txt"));
7
8 Files.move(Path.of("rezultat.txt"), Path.of("date", "
9 rezultat_final.txt"));
10
11 Files.deleteIfExists(Path.of("date", "gol.txt"));
12 // NU arunca exceptie daca fisierul nu exista
```

#### Notă

`Files.delete` aruncă `NoSuchFileException` dacă fișierul nu există; `Files.deleteIfExists` returnează pur și simplu `false` în acel caz, fără excepție — alegeți varianta potrivită situației: dacă absența fișierului e o eroare reală de logică, prima formă documentează mai clar acea așteptare.

### 17.4.4 Informații despre fișiere; vizitarea directoarelor

Listing 17.12: Interogarea proprietăților unui fișier, și parcurgerea unui director

```
1 boolean exista = Files.exists(cale);
2 boolean director = Files.isDirectory(cale);
3 long dimensiune = Files.size(cale); // in octeti
4
5 try (Stream<Path> continutDirector = Files.list(Path.of("date
6 "))) {
7     // NU recursiv
8     continutDirector.forEach(System.out::println);
9 }
10 try (Stream<Path> toateFisierele = Files.walk(Path.of("date")
11 )) {
12     // RECURSIV, tot subarborele
13     toateFisierele
14         .filter(p -> p.toString().endsWith(".txt"))
15         .forEach(System.out::println);
16 }
```

**Notă**

Atât `Files.list`, cât și `Files.walk`, returnează un `Stream<Path>` (Capitolul 15) — de aici posibilitatea de a combina direct `filter/map/collect` cu parcurgerea sistemului de fișiere. Ambele metode **trebuie** folosite în `try-with-resources`: stream-ul rezultat menține deschis un handle către sistemul de fișiere, eliberat abia la închidere, spre deosebire de stream-urile obișnuite din colecții, deja aflate complet în memorie.

Pentru filtrare directă după un tipar de nume de fișier (*glob pattern*), fără a construi manual un `Stream` filtrat:

```
1 try (DirectoryStream<Path> fisierText = Files.  
    newDirectoryStream(Path.of("date"), "*.txt")) {  
2     for (Path p : fisierText) {  
3         System.out.println(p);  
4     }  
5 }
```

**Listing 17.13:** `DirectoryStream` cu filtru glob, pentru fișierele dintr-un singur director

**Notă**

Tiparele glob (`*.txt`, `cont-???.dat`, `**/*.java` pentru recursivitate) sunt o notație mai simplă decât expresiile regulate complete (Secțiunea 18.3, pe care o vom studia riguros în capitolul următor), suficientă pentru majoritatea filtrărilor de nume de fișiere întâlnite în practică.

**Rezumat**

- ▶ `InputStream/OutputStream` operează pe octeți; `Reader/Writer` operează pe caractere — ambele ierarhii sunt gândite ca filtre combinabile (`BufferedInputStream`, `InputStreamReader` etc.), nu clase monolitice.
- ▶ Specificați întotdeauna explicit codificarea pentru I/O de text — codificarea implicită a platformei variază și poate produce text corupt.
- ▶ `DataInput/DataOutput` scriu/citesc tipuri primitive în format binar portabil; `RandomAccessFile` permite poziționare explicită, utilă pentru înregistrări de dimensiune fixă.
- ▶ NIO.2 (`Path`, `Files`) oferă un API modern, cu metode directe pentru citire/scriere de text (`readString`, `writeString`) și operații pe fișiere (`copy`, `move`, `delete`).
- ▶ `Files.list/walk` returnează un `Stream<Path>`, combinabil cu operațiile din Cap. 15–16; ambele necesită `try-with-resources`.

**Exerciții**

1. Scrieți o metodă care salvează o listă de `ContBancar` într-un fișier text, o linie per cont (titular și sold separate prin virgulă), folosind `Files.writeString` (construind mai întâi întregul conținut ca `String`).
2. Scrieți metoda complementară, care citește fișierul de mai sus cu `Files.readAllLines` și reconstruiește lista de conturi.
3. Folosind `DataOutputStream/DataInputStream`, salvați și recitiți datele unui singur `ContBancar` în format binar.
4. Scrieți un mic program care parcurge recursiv un director (`Files.walk`) și afișează toate fișierele cu extensia `.java`, îm-

preună cu dimensiunea fiecăruia.

5. Explicați, în cuvinte proprii, de ce `RandomAccessFile` necesită înregistrări de dimensiune fixă pentru a fi cu adevărat util, și ce s-ar întâmpla dacă ați încerca să folosiți acces aleator pe înregistrări de dimensiune variabilă.

# Input/Output (II): serializare. Expresii regulate. Fișiere mapate în memorie

# 18

## Introducere

Înceie subiectul I/O cu trei instrumente specializate: fișierele mapate în memorie (pentru performanță la fișiere foarte mari), serializarea (pentru salvarea directă a obiectelor Java, fără a scrie manual conversia lor la text/octeți), și expresiile regulate (pentru potrivire și extragere de tipare din text, un instrument mult mai puternic decât metodele `String` folosite până acum).

## 18.1 Fișiere mapate în memorie; blocarea fișierelor

### 18.1.1 Fișiere mapate în memorie

Un fișier *mapat în memorie* este adus direct în spațiul de adrese al procesului, prin sistemul de operare — accesul la conținutul lui devine echivalent cu accesul la un array obișnuit din memorie, fără apeluri explicite de citire/scriere:

```
1 try (RandomAccessFile fisier = new RandomAccessFile("conturi.  
   dat", "rw");  
2     FileChannel canal = fisier.getChannel()) {  
3  
4     MappedByteBuffer buffer = canal.map(FileChannel.MapMode.  
       READ_WRITE, 0, fisier.length());  
5  
6     double sold = buffer.getDouble(8); // citește direct din  
       // memoria mapată, fără I/O explicit  
7     buffer.putDouble(8, sold + 100.0); // scrie direct -  
       // modificările ajung automat pe disc  
8  
9 }  
10 }
```

Listing 18.1: Maparea unui fișier în memorie, prin `FileChannel`

## Notă

Beneficiul principal apare la fișiere **foarte mari** sau la acces **aleator intensiv** (Subsecțiunea 17.3.2): sistemul de operare gestionează automat ce porțiuni ale fișierului sunt efectiv aduse în memoria fizică, evitând suprapunerea costului de sistem al fiecărui apel individual read/write. Pentru fișiere mici sau acces predominant secvențial, diferența de performanță față de stream-urile obișnuite (Secțiunea 17.1) este de regulă neglijabilă — alegeți maparea în memorie doar când profilarea arată efectiv un beneficiu.

### 18.1.2 Blocarea fișierelor

Spre deosebire de lock-urile din Secțiunea 14.2 (care coordonează thread-uri în interiorul aceleiași aplicații), un `FileLock` coordonează accesul la un fișier între procese separate ale sistemului de operare:

**Listing 18.2:** Blocarea unui fișier, pentru acces exclusiv între procese

```

1 try (RandomAccessFile fisier = new RandomAccessFile("conturi.
  dat", "rw"));
2     FileChannel canal = fisier.getChannel();
3     FileLock lacat = canal.lock()) {
4
5     // sectiune critica - niciun alt proces nu poate scrie
6     // concurrent in acest fisier
7 }

```

**Notă**

`lock()` blochează (așteaptă) până la obținerea *lock*-ului; `tryLock()` încearcă o singură dată, returnând `null` imediat dacă fișierul e deja blocat de alt proces — paralelă directă cu `Lock.tryLock()` din Subsecțiunea 14.2.1, dar la nivelul sistemului de fișiere, nu al thread-urilor JVM. Un `FileLock` nu protejează împotriva altor thread-uri din **aceeași** aplicație — pentru acel scop, mecanismele din Secțiunea 14.1–Secțiunea 14.2 rămân necesare.

## 18.2 Serializare

### 18.2.1 Salvare și încărcare de obiecte serializabile

*Serializarea* convertește automat un obiect complet (împreună cu toate câmpurile sale) într-o secvență de octeți, salvabilă și reconstituibilă ulterior, fără a scrie manual logica de conversie folosită în Subsecțiunea 17.3.1:

**Listing 18.3:** Marcarea unei clase ca serializabilă

```

1 public class ContBancar implements Serializable {
2     private static final long serialVersionUID = 1L;
3     // vezi Sec. 18.2.4
4
5     private String titular;
6     private double sold;
7     // ...
8 }

```

**Listing 18.4:** Salvarea și încărcarea unei liste de obiecte, prin serializare

```

1 try (ObjectOutputStream out = new ObjectOutputStream(new
  FileOutputStream("conturi.ser"))) {
2     out.writeObject(conturi); // salveaza intreaga lista,
3     // RECURSIV, cu toate obiectele continute
4 }
5
6 try (ObjectInputStream in = new ObjectInputStream(new
  FileInputStream("conturi.ser"))) {
7
8     @SuppressWarnings("unchecked")
9     List<ContBancar> conturiRecuperate =
10         (List<ContBancar>) in.readObject(); // necesita cast
11 }

```

**Notă**

`Serializable` este o *interfață marcator* — nu declară nicio metodă (spre deosebire de orice altă interfață întâlnită până acum, Secțiunea 7.1), ci doar semnalează JVM-ului că instanțele clasei pot fi serializate.

Serializarea este **recursivă**: dacă `ContBancar` conține o listă de `Extras` (Subsecțiunea 7.4.4), acea listă și fiecare `Extras` din ea trebuie, la rândul lor, să fie serializabile — altfel serializarea eșuează cu `NotSerializableException`, la execuție, nu la compilare.

### 18.2.2 Câmpuri transient

Un câmp transient este exclus explicit din serializare — util pentru câmpuri care nu ar avea sens salvate (conexiuni, thread-uri) sau care pur și simplu nu sunt serializabile:

```
1 public class ContBancar implements Serializable {
2     private static final long serialVersionUID = 1L;
3     private static final transient Logger LOG = Logger.
4         getLogger(ContBancar.class.getName());
5
6     private String titular;
7     private double sold;
8     private transient String parolaTemporaraSesiune;
9     // NU trebuie salvata niciodata
10 }
```

Listing 18.5: Un câmp transient, exclus din serializare

#### Notă

`Logger` (Subsecțiunea 9.4.2) nu este serializabil — fără `transient`, prezența sa ca simplu câmp de instanță ar face întreaga clasă nefuncțională la serializare (deși, fiind și static, ar fi oricum exclus automat — câmpurile static nu fac parte din starea unei instanțe și nu sunt niciodată serializate, indiferent de `transient`). La deserializare, un câmp transient primește valoarea implicită a tipului său (`null` pentru obiecte, `0/false` pentru primitive) — codul care folosește obiectul reconstituit trebuie să anticipeze această reinițializare, eventual explicit, în `readObject` personalizat (Subsecțiunea 18.2.3).

### 18.2.3 `readObject/writeObject` personalizate

Pentru control fin asupra procesului de serializare, o clasă poate defini metode private speciale, apelate automat de mecanismul de serializare:

```
1 private void writeObject(ObjectOutputStream out) throws
2     IOException {
3     out.defaultWriteObject(); // serializeaza normal toate
4     // ...                      // campurile ne-transient
5 }
6
7 private void readObject(ObjectInputStream in) throws
8     IOException, ClassNotFoundException {
9     in.defaultReadObject(); // deserializeaza normal toate
10    // ...                      // campurile ne-transient
11    if (sold < 0) {
12        throw new InvalidObjectException("Sold negativ dupa
13        deserializare - date corupte");
14    }
15 }
```

Listing 18.6: Personalizarea serializării, pentru validare la reconstituire

**Notă**

Aceste metode sunt private, dar apelate automat de `ObjectOutputStream/ObjectInputStream` prin reflecție (Capitolul 8) — un mecanism special, cunoscut doar de infrastructura de serializare, nu un apel obișnuit de metodă. `readObject` personalizat este locul potrivit pentru validarea invariantilor unui obiect reconstituit dintr-o sursă externă, potențial nesigură (vezi Subsecțiunea 18.2.4).

**Notă**

Două metode înrudite, mai rar necesare: `writeReplace` permite **substituirea** obiectului serializat cu altul (util, de exemplu, pentru a serializa o versiune simplificată); `readResolve` permite **înlocuirea** obiectului deserializat, util pentru a păstra invariantul unui singur-obiect (*singleton*, Subsecțiunea 14.2.4) — fără `readResolve`, deserializarea ar putea crea o a doua instanță a unei clase gândite să aibă una singură.

### 18.2.4 Versionare și securitate

Câmpul `serialVersionUID`, deja introdus în Subsecțiunea 18.2.1, identifică versiunea formatului serializat al unei clase:

**Notă**

Fără `serialVersionUID` declarat explicit, JVM-ul îl calculează automat, pe baza structurii complete a clasei — **orice** modificare a clasei (chiar adăugarea unei metode noi, fără legătură cu datele) schimbă acest identificator calculat, invalidând orice date serializate anterior. Declararea explicită a lui `serialVersionUID` (o valoare arbitrară, de obicei 1L inițial, incrementată manual la schimbări incompatibile) oferă control asupra a ce anume constituie o schimbare „incompatibilă” cu datele deja salvate.

**Notă**

Un avertisment important de securitate: deserializarea unor date provenite dintr-o sursă **nesigură** (de exemplu, primite prin rețea, de la un utilizator neautentificat) este un vector de atac cunoscut — procesul de deserializare poate fi manipulat pentru a construi obiecte arbitrare sau a declanșa cod neintenționat, înainte ca aplicația să apuce să valideze datele rezultate. Nu deserializați niciodată date dintr-o sursă în care nu aveți încredere completă; pentru schimb de date cu surse externe, formate text precum JSON (procesate prin biblioteci dedicate, nu prin serializarea Java nativă) sunt, azi, alegerea implicit mai sigură.

## 18.3 Expresii regulate: sintaxă de bază

### 18.3.1 Sintaxă și testarea potrivirilor

O *expresie regulată* descrie un **tipar** de text, mult mai expresiv decât metodele `String` folosite până acum (`contains`, `startsWith`, din Subsecțiunea 3.1.2):

```

1 Pattern tiparNume = Pattern.compile("[A-Z][a-z]+ [A-Z][a-z]+"
  );
2 // Prenume Nume
3 Matcher potrivitor = tiparNume.matcher("Ana Popescu");
4
5 boolean valid = potrivitor.matches(); // true - se potrivește
6                                     // EXACT tiparul, integral

1 boolean valid = "Ana Popescu".matches("[A-Z][a-z]+ [A-Z][a-z]
  ]+");
2 // echivalent, mai concis

```

Listing 18.7: Testarea unei potriviri, cu Pattern și Matcher

Listing 18.8: Forma scurtă, pentru o singură verificare

Tabela 18.1: Simbolurile de bază ale sintaxei expresiilor regulate în Java.

| Simbol          | Semnificație                                         |
|-----------------|------------------------------------------------------|
| .               | orice caracter (cu excepția, implicit, a liniei noi) |
| *, +, ?         | zero-sau-mai-multe, unu-sau-mai-multe, zero-sau-unu  |
| [A-Z], [a-z0-9] | o clasă de caractere, dintr-un interval dat          |
| \d, \w, \s      | cifră, caracter de cuvânt, spațiu alb                |
| ^, \$           | începutul, respectiv sfârșitul textului (sau liniei) |

**Notă**

Pattern.compile(...) este relativ costisitor — pentru un tipar folosit repetat (de exemplu, într-o buclă sau apelat frecvent), compilați-l o singură dată, într-un câmp static final, în loc să apelați forma scurtă String.matches(...) repetat, care recompilează tiparul la fiecare apel. matches() cere o potrivire pe întregul text; find(), folosit mai jos, caută o potrivire oriunde în text, nu neapărat de la început.

**18.3.2 Grupuri de captură**

Parantezele într-o expresie regulată delimitează *grupuri de captură* — porțiuni ale potrivirii, extrase individual:

```

1 Pattern tiparLinie = Pattern.compile("(\\w+ \\w+), (\\d+\\.\\d+)");
2 // (titular), (sold)
3 Matcher potrivitor = tiparLinie.matcher("Ana Popescu, 500.0");
4
5 if (potrivitor.find()) {
6     String titular = potrivitor.group(1);
7     // primul grup - "Ana Popescu"
8     double sold = Double.parseDouble(potrivitor.group(2));
9     // al doilea grup - "500.0"
10 }

```

Listing 18.9: Extragerea datelor dintr-un text formatat, cu grupuri de captură

**Notă**

group(0) (sau group(), fără argument) returnează întreaga potrivire; group(1), group(2) etc. returnează conținutul fiecărui grup de captură, în ordinea parantezelor deschise. Această tehnică oferă o alternativă mai robustă la analiza manuală a unui fișier text precum conturi.txt, salvat în Subsecțiunea 17.4.2, tolerantă la variații mi-

nore de format (spații suplimentare, de exemplu) pe care un simplu `split(",")` le-ar trata greșit.

## 18.4 Expresii regulate: aplicații

### 18.4.1 `split`

Am folosit deja `split` pe un separator simplu, în Subsecțiunea 3.1.2 — argumentul este, de fapt, întotdeauna o expresie regulată, nu un simplu text literal:

**Listing 18.10:** `split`, cu un tipar mai flexibil decât un singur caracter literal

```
1 String linie = "Ana Popescu, 500.0, depunere";
2 // spatii inconsistente dupa virgula
3
4 String[] campuri = linie.split(",\\s*");
5 // virgula, urmata de ZERO sau mai multe spatii
6 // campuri = "Ana Popescu", "500.0", "depunere"
```

#### Notă

`split(",")` simplu ar fi lăsat spațiile inconsistente în fiecare câmp rezultat (" 500.0", cu spațiu la început) — tiparul `",\\s*"` le elimină automat, direct în pasul de separare. Un al doilea argument opțional, `limit`, controlează numărul maxim de câmpuri rezultate, util atunci când restul textului (după un număr fix de separatoare) trebuie păstrat neseplat.

### 18.4.2 `replace`

`replaceAll/replaceFirst` înlocuiesc potrivirile unui tipar, putând referi grupurile capturate în textul de înlocuire:

**Listing 18.11:** Înlocuire simplă, și înlocuire cu referire la grupuri capturate

```
1 String curatat = "Ana Popescu".replaceAll("\\s+", " ");
2 // normalizeaza spatiile multiple
3
4 String mascat = "1234567890".replaceAll("(\\d{4})\\d+(\\d{2})",
5 // "1234****90"
6 // "$1****$2");
```

#### Notă

În șirul de înlocuire, `$1`, `$2` etc. referă conținutul grupurilor de captură din tiparul de căutare — exemplul de mai sus păstrează primele 4 și ultimele 2 cifre ale unui identificator, mascând restul, un tipar frecvent pentru afișarea parțială a datelor sensibile (numere de cont, carduri) în interfețe sau jurnale de logging (Subsecțiunea 9.4.2).

### 18.4.3 `Flags`

Câteva opțiuni modifică global comportamentul unei potriviri:

**Listing 18.12:** Flag-uri comune: potrivire insensibilă la majuscule, mod multi-linie

```
1 Pattern insensibil = Pattern.compile("ana popescu", Pattern.
2 CASE_INSENSITIVE);
3
4 boolean potrivestePeAna = insensibil.matcher("ANA POPESCU").
5 matches();
```

```

3 // true
4
5 Pattern multiLinie = Pattern.compile("^Sold:", Pattern.
6     MULTILINE);
7 // ^se potrivește la FIECARE linie

```

### Notă

Fără `Pattern.MULTILINE`, ancorele `^/$` din Tabela 18.1 se potrivesc doar la începutul/sfârșitul **întregului** text, nu al fiecărei linii dintr-un text pe mai multe linii — o sursă frecventă de confuzie atunci când se procesează fișiere text linie cu linie folosind expresii regulate cu ancore.

### Rezumat

- ▶ Fișierele mapate în memorie (`MappedByteBuffer`) tratează un fișier ca memorie obișnuită, util pentru fișiere mari cu acces aleator intensiv; `FileLock` coordonează accesul la un fișier între procese, nu între thread-uri.
- ▶ `Serializable` (interfață marcator) permite salvarea/reconstituirea automată a unui obiect; `transient` exclude câmpuri din serializare; `readObject/writeObject` personalizate oferă control fin.
- ▶ `serialVersionUID` controlează compatibilitatea între versiuni ale unei clase serializate; deserializarea datelor dintr-o sursă nesigură este un risc real de securitate.
- ▶ `Pattern/Matcher` testează potriviri de expresii regulate; grupurile de captură (paranteze) extrag porțiuni ale potrivirii.
- ▶ `split` și `replaceAll/replaceFirst` acceptă expresii regulate complete, nu doar text literal; flag-uri precum `CASE_INSENSITIVE/MULTILINE` modifică global comportamentul potrivirii.

### Exerciții

1. Scrieți o expresie regulată care validează un format simplu de sold monetar (cifre, opțional cu punct zecimal și exact două zecimale), și testați-o pe câteva exemple valide/invalid.
2. Folosind grupuri de captură, extrageți titularul și soldul dintr-o listă de linii text de forma "Titular: Ana Popescu, Sold: 500.0", tolerantă la spații suplimentare.
3. Scrieți o metodă care maschează parțial un nume de titular (păstrează prima literă, înlocuiește restul primului nume cu \*), folosind `replaceAll` cu grupuri.
4. Rescrieți exercițiul de parsare a fișierului `conturi.txt` din Cap. 17 folosind expresii regulate cu grupuri de captură, în loc de `split(",")` simplu.
5. Serializați o listă de `ContBancar` într-un fișier, apoi modificați clasa (adăugați un câmp nou) și încercați să deserializați fișierul vechi – observați și explicați comportamentul obținut, cu și fără `serialVersionUID` declarat explicit.



## Introducere

Am salvat datele conturilor, până acum, ca text simplu (Capitolul 17), format binar (Capitolul 18) și obiecte serializate (Secțiunea 18.2). *XML* (*Extensible Markup Language*) oferă un format text **structurat ierarhic** și **auto-descriptiv**, larg folosit pentru schimbul de date între sisteme diferite [2] — spre deosebire de serializarea Java din capitolul anterior, un document XML poate fi citit și de programe scrise în cu totul alte limbaje.

## 19.1 Structura unui document XML; parsare DOM

### 19.1.1 Structura unui document XML

Reprezentăm banca noastră ca document XML, cu conturi și istoricul lor de tranzacții:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <banca>
3   <cont titular="Ana Popescu" sold="500.0">
4     <tranzactie tip="depunere" suma="150.0"/>
5     <tranzactie tip="retragere" suma="50.0"/>
6   </cont>
7   <cont titular="Ion Ionescu" sold="1200.0">
8     <tranzactie tip="depunere" suma="1200.0"/>
9   </cont>
10 </banca>

```

**Listing 19.1:** Un document XML, pentru banca noastră

## Notă

Un document XML **bine format** (*well-formed*) respectă câteva reguli stricte: un singur element rădăcină (<banca> de mai sus), fiecare element deschis trebuie închis (sau auto-inchis, ca <tranzactie .../>), elementele se imbrică fără suprapunere, iar valorile atributelor sunt întotdeauna între ghilimele. Spre deosebire de HTML, unde browserele tolerează adesea abateri de la aceste reguli, un parser XML **respinge** complet un document care nu le respectă.

### 19.1.2 Parsare DOM

Parsarea *DOM* (*Document Object Model*) încarcă întregul document într-un arbore de obiecte în memorie, navigabil integral:

```

1 DocumentBuilderFactory factory = DocumentBuilderFactory.
   newInstance();
2 DocumentBuilder builder = factory.newDocumentBuilder();
3 Document document = builder.parse(new File("banca.xml"));
4
5 document.getDocumentElement().normalize();
6 // consolideaza nodurile de text adiacente

```

**Listing 19.2:** Parsarea documentului XML, cu DOM

**Listing 19.3:** Parcurgerea arborelui DOM, extrăgând date în obiecte ContBancar

```

7
8 NodeList conturiXml = document.getElementsByTagName("cont");
1 List<ContBancar> conturi = new ArrayList<>();
2
3 for (int i = 0; i < conturiXml.getLength(); i++) {
4     Element contXml = (Element) conturiXml.item(i);
5
6     String titular = contXml.getAttribute("titular");
7     double sold = Double.parseDouble(contXml.getAttribute("
8     sold"));
9
10    conturi.add(new ContBancar(titular, sold));
11 }
```

#### Notă

NodeList nu implementează Iterable (Subsecțiunea 11.1.2) — spre deosebire de colecțiile din Capitolul 11, necesită o buclă indexată clasică (for cu contor), nu for-each direct. Cast-ul explicit (**Element**) conturiXml.item(i) este necesar fiindcă NodeList lucrează, la nivelul cel mai general, cu Node (care poate fi element, text, comentariu etc.) — Element este interfața specifică pentru noduri care reprezintă elemente XML propriu-zise, cu atribute și copii.

#### Notă

DOM este potrivit pentru documente de dimensiune **moderată**, unde întregul document încapă confortabil în memorie și aveți nevoie de acces aleator, repetat, la structura sa (căutări, modificări). Pentru documente foarte mari, unde încărcarea completă ar fi ineficientă, parserele de tip *streaming* din Subsecțiunea 19.3.3 sunt alegerea potrivită.

## 19.2 Validare (DTD, XML Schema)

### 19.2.1 DTD

Un *DTD* (*Document Type Definition*) declară structura permisă a unui document — ce elemente și atribute sunt valide, și cum se pot imbrica:

**Listing 19.4:** Un DTD simplu, pentru documentul banca.xml

```

1 <!DOCTYPE banca [
2     <!ELEMENT banca (cont*)>
3     <!ELEMENT cont (tranzactie*)>
4     <!ATTLIST cont titular CDATA #REQUIRED>
5     <!ATTLIST cont sold CDATA #REQUIRED>
6     <!ELEMENT tranzactie EMPTY>
7     <!ATTLIST tranzactie tip CDATA #REQUIRED>
8     <!ATTLIST tranzactie suma CDATA #REQUIRED>
9 ]>
```

#### Notă

DTD este tehnologia de validare **istorică** a XML-ului — sintaxă proprie, distinctă de XML însuși, fără suport pentru tipuri de date (orice atribut e tratat ca text simplu, CDATA) și fără suport pentru namespace-uri (Subsecțiunea 19.3.2). Rămâne întâlnită în documente mai vechi, dar rar aleasă pentru proiecte noi.

## 19.2.2 XML Schema

*XML Schema* (fișiere .xsd) este succesorul modern al DTD — scris el însuși în XML, cu suport pentru tipuri de date și namespace-uri:

```

1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
2   <xs:element name="banca">
3     <xs:complexType>
4       <xs:sequence>
5         <xs:element name="cont" maxOccurs="unbounded"
6       >
7         <xs:complexType>
8           <xs:sequence>
9             <xs:element name="tranzactie"
10            maxOccurs="unbounded" minOccurs="0">
11              <xs:complexType>
12                <xs:attribute name="tip"
13                type="xs:string" use="required"/>
14                <xs:attribute name="suma"
15                type="xs:double" use="required"/>
16              </xs:complexType>
17            </xs:element>
18          </xs:sequence>
19          <xs:attribute name="titular" type="xs:
20            string" use="required"/>
21          <xs:attribute name="sold" type="xs:
22            double" use="required"/>
23        </xs:complexType>
24      </xs:element>
25    </xs:sequence>
26  </xs:complexType>
27 </xs:element>
28 </xs:schema>

```

**Listing 19.5:** O schemă XML, echivalentă cu DTD-ul de mai sus, dar cu tipuri de date

```

1 SchemaFactory schemaFactory = SchemaFactory.newInstance(
2   XMLConstants.W3C_XML_SCHEMA_NS_URI);
3 Schema schema = schemaFactory.newSchema(new File("banca.xsd"));
4 Validator validator = schema.newValidator();
5 validator.validate(new StreamSource(new File("banca.xml")));
6 // arunca SAXException daca invalid

```

**Listing 19.6:** Validarea unui document XML față de o schemă

### Notă

Observați `type="xs:double"` pentru atributul `sold` — spre deosebire de DTD, unde totul e text simplu, XML Schema declară explicit **tipuri de date**, permițând validarea nu doar a structurii, ci și a conținutului (un `sold` care nu e un număr valid eșuează validarea). Deși schema de mai sus pare considerabil mai verboasă decât DTD-ul echivalent, ea oferă garanții mult mai puternice, verificate automat, înainte ca datele să ajungă în codul aplicației — similar principiului validării timpurii discutat deja la excepții, în Subsecțiunea 9.1.4.

## 19.3 XPath; namespace-uri; parsere streaming

### 19.3.1 XPath

*XPath* este un limbaj de interogare pentru documente XML, mult mai direct decât navigarea manuală a arborelui DOM din Subsecțiunea 19.1.2:

**Listing 19.7:** O interogare XPath, pentru conturile cu sold peste 1000

```

1 XPathFactory xpathFactory = XPathFactory.newInstance();
2 XPath xpath = xpathFactory.newXPath();
3
4 XPathExpression expresie = xpath.compile("/banca/cont[@sold >
   1000]/@titular");
5 NodeList rezultate = (NodeList) expresie.evaluate(document,
   XPathConstants.NODESET);
6
7 for (int i = 0; i < rezultate.getLength(); i++) {
8     System.out.println(rezultate.item(i).getNodeValue());
9     // titularii conturilor mari
10 }
```

#### Notă

Expresia `/banca/cont[@sold > 1000]/@titular` se citește: pornind de la rădăcină, elementele `cont` ale căror atribut `sold` depășește 1000, apoi atributul `titular` al fiecăruia. O singură expresie XPath înlocuiește aici bucla `for` explicită din Subsecțiunea 19.1.2, combinată cu o condiție — similar, ca stil declarativ, cu `filter/map` de la stream-uri (Secțiunea 15.2), dar exprimat ca text, evaluat la execuție, nu verificat la compilare.

### 19.3.2 Namespace-uri

Un *namespace* XML previne coliziunile de nume de elemente atunci când documente din surse diferite sunt combinate — declarat printr-un atribut special `xmlns`:

**Listing 19.8:** Un document XML cu namespace explicit

```

1 <banca:document xmlns:banca="http://exemplu.ro/banca/v1">
2     <banca:cont titular="Ana Popescu" sold="500.0"/>
3 </banca:document>
```

**Listing 19.9:** Parsare „namespace-aware”, necesară pentru documente cu namespace-uri

```

1 DocumentBuilderFactory factory = DocumentBuilderFactory.
   newInstance();
2 factory.setNamespaceAware(true); // OBLIGATORIU pentru
   // documente cu namespace-uri
3
4
5 DocumentBuilder builder = factory.newDocumentBuilder();
6 Document document = builder.parse(new File("banca_namespace.
   xml"));
7
8 NodeList conturi = document.getElementsByTagNameNS("http://
   exemplu.ro/banca/v1", "cont");
```

#### Notă

Fără `setNamespaceAware(true)`, parserul tratează prefixul `banca`: ca parte literală a numelui elementului ("`banca:cont`"), nu ca un namespace separat — `getElementsByTagName("cont")` obișnuit nu ar găsi elementele, fiind necesară varianta `...NS`, cu URI-ul namespace-

ului explicit. Namespace-urile devin relevante mai ales la combinarea documentelor din surse externe (de exemplu, un fișier XML care încorporează fragmente SVG sau MathML, fiecare cu propriul namespace) — pentru documente proprii, simple, ca `banca.xml` din Subsecțiunea 19.1.1, namespace-urile sunt adesea inutile.

### 19.3.3 Parsere streaming: SAX și StAX

Spre deosebire de DOM (Subsecțiunea 19.1.2), care încarcă tot documentul în memorie, parserele *streaming* procesează documentul secvențial, fără a-l reține integral — esențiale pentru documente foarte mari.

```

1 SAXParserFactory factory = SAXParserFactory.newInstance();
2 SAXParser parser = factory.newSAXParser();
3
4 parser.parse(new File("banca.xml"), new DefaultHandler() {
5     @Override
6     public void startElement(String uri, String localName,
7         String qName, Attributes attrs) {
8         if (qName.equals("cont")) {
9             System.out.println("Cont: " + attrs.getValue("titular"));
10        }
11    });

```

Listing 19.10: SAX: parsare bazată pe evenimente (*push*)

```

1 XMLInputFactory factory = XMLInputFactory.newInstance();
2 XMLStreamReader cititor = factory.createXMLStreamReader(new
3     FileReader("banca.xml"));
4
5 while (cititor.hasNext()) {
6     int eveniment = cititor.next();
7     if (eveniment == XMLStreamConstants.START_ELEMENT &&
8         cititor.getLocalName().equals("cont")) {
9         System.out.println("Cont: " + cititor.
10             getAttributeValue(null, "titular"));
11     }
12 }

```

Listing 19.11: StAX: parsare bazată pe cursor (*pull*)

#### Notă

SAX este *push*: parserul „împinge” evenimente către callback-urile voastre (`startElement` etc.), pe măsură ce le întâlnește — codul vostru nu controlează ritmul parcurgerii. StAX este *pull*: codul vostru cere explicit `next()` evenimentul următor, oferind control mai fin (de exemplu, oprirea parcurgerii de îndată ce datele necesare au fost găsite, fără a citi restul documentului). Ambele evită costul de memorie al DOM, la prețul unui cod mai puțin direct decât navigarea unui arbore complet construit.

## 19.4 Generarea documentelor XML; transformări XSL

### 19.4.1 Generarea documentelor XML

Simetric parsării din Subsecțiunea 19.1.2, un document XML poate fi construit programatic, pornind de la un Document gol:

**Listing 19.12:** Construirea unui document XML, din obiecte ContBancar

```
1 DocumentBuilder builder = DocumentBuilderFactory.newInstance
  ().newDocumentBuilder();
2 Document document = builder.newDocument();
3
4 Element radacina = document.createElement("banca");
5 document.appendChild(radacina);
6
7 for (ContBancar cont : conturi) {
8     Element contXml = document.createElement("cont");
9     contXml.setAttribute("titular", cont.getTitular());
10    contXml.setAttribute("sold", String.valueOf(cont.getSold
11    ()));
12    radacina.appendChild(contXml);
13 }
```

**Listing 19.13:** Scrierea documentului construit, într-un fișier

```
1 Transformer transformer = TransformerFactory.newInstance().
  newTransformer();
2 transformer.setOutputProperty(OutputKeys.INDENT, "yes");
3 // formatare lizibila, cu indentare
4
5 transformer.transform(new DOMSource(document), new
  StreamResult(new File("banca_noua.xml")));
```

#### Notă

Transformer este, aparent surprinzător, clasa folosită atât pentru scrierea unui document XML într-un fișier, cât și pentru transformările XSL discutate mai jos — scrierea simplă este, conceptual, o „transformare identică” (fără nicio modificare de conținut), aplicată direct sursei DOM construite.

### 19.4.2 Transformări XSL (context)

XSLT (*Extensible Stylesheet Language Transformations*) descrie, tot în XML, reguli de transformare a unui document XML în alt format — alt XML, HTML, sau text simplu:

**Listing 19.14:** Aplicarea unei transformări XSLT, pentru a genera un raport HTML

```
1 Transformer transformer = TransformerFactory.newInstance()
2   .newTransformer(new StreamSource(new File("raport.xsl")))
3   ;
4 transformer.transform(
5     new StreamSource(new File("banca.xml")),
6     new StreamResult(new File("raport.html"))
7 );
```

#### Notă

Foaia de stil XSLT (raport.xsl) conține propriile reguli, scrise tot în XML, care specifică cum se transformă fiecare tip de element sursă —

un limbaj declarativ complet separat, dincolo de scopul acestui curs introductiv. Menționăm XSLT aici mai ales pentru completitudine: recunoașteți conceptul (transformare de format, descrisă declarativ, aplicată printr-un Transformer Java) dacă îl întâlniți într-un sistem existent, fără a fi nevoie să scrieți propriile foi de stil XSLT pentru scopurile acestui curs.

### Rezumat

- ▶ Un document XML bine format are un singur element rădăcină, taguri închise corect și atribute între ghilimele; DOM încarcă întregul document ca arbore navigabil în memorie.
- ▶ DTD și XML Schema validează structura (și, pentru Schema, tipurile de date) unui document față de o definiție separată; XML Schema este alegerea modernă recomandată.
- ▶ XPath oferă interogări declarative asupra unui document XML, alternativă concisă la navigarea manuală a arborelui DOM.
- ▶ Namespace-urile (xmlns) previn coliziuni de nume la combinarea documentelor din surse diferite; necesită parsare „namespace-aware” explicită.
- ▶ SAX (push, bazat pe evenimente) și StAX (pull, bazat pe cursor) sunt parsare streaming, potrivite pentru documente mari, unde încărcarea completă în memorie (ca la DOM) ar fi inefficientă.
- ▶ Un document XML poate fi construit programatic (DOM) și scris cu Transformer; XSLT descrie transformări declarative XML→alt format.

### Exerciții

1. Scrieți un document XML banca.xml cu cel puțin trei conturi, fiecare cu câteva tranzacții, apoi parsați-l cu DOM și reconstruiți lista de obiecte ContBancar corespunzătoare.
2. Scrieți o expresie XPath care extrage titularii tuturor conturilor cu **cel puțin două** tranzacții, folosind funcția XPath count().
3. Scrieți o schemă XML (.xsd) simplificată pentru documentul banca.xml, care validează că atributul tip al unei tranzacții este fie "depunere", fie "retragere" (indicație: căutați xs:enumeration).
4. Rescrieți exercițiul 1, folosind de această dată un parser SAX în loc de DOM, și comparați complexitatea codului rezultat.
5. Generați programatic un document XML banca.xml dintr-o listă de obiecte ContBancar existentă, folosind DocumentBuilder și Transformer, apoi verificați manual conținutul fișierului rezultat.



### Introducere

Am menționat deja rețeaua, în trecut, ca exemplu tipic de sarcină I/O-bound potrivită pentru virtual threads (Subsecțiunea 13.2.1). Acest capitol tratează efectiv comunicarea în rețea: sockets TCP de nivel jos, apoi API-ul HTTP modern, folosit azi de marea majoritate a aplicațiilor care comunică pe Internet.

## 20.1 Conectarea la un server

### 20.1.1 Socket: conectare de bază

Un Socket reprezintă o conexiune TCP către un server, identificat prin adresă și port:

```
1 try (Socket socket = new Socket("localhost", 5000)) {
2     Writer catreServer = new OutputStreamWriter(socket.
3         getOutputStream(), StandardCharsets.UTF_8);
4     BufferedReader dinServer = new BufferedReader(
5         new InputStreamReader(socket.getInputStream(),
6             StandardCharsets.UTF_8));
7
8     catreServer.write("SOLD Ana Popescu\n");
9     catreServer.flush(); // OBLIGATORIU - altfel textul
10                        // poate ramane in buffer, netrimis
11
12     String raspuns = dinServer.readLine();
13     System.out.println("Server: " + raspuns);
14 }
```

Listing 20.1: Conectarea la un server, ca client, folosind stream-uri I/O deja cunoscute

### Notă

Odată stabilită conexiunea, comunicarea prin socket.getInputStream()/getOutputStream() folosește exact aceleași clase din Capitolul 17 (Reader, Writer, BufferedReader) — un socket este, din perspectiva codului care îl folosește, doar o altă sursă/destinație de octeți, la fel ca un fișier. flush() este esențial pentru comunicarea în rețea: fără el, datele scrise pot rămâne în buffer-ul intern, netransmise, până la umplerea completă a acestuia sau închiderea conexiunii.

### 20.1.2 Adrese de Internet; timeouts

```
1 InetAddress adresa = InetAddress.getByName("site.ro");
2 System.out.println(adresa.getHostAddress());
3 // adresa IP rezolvata
4
5 Socket socket = new Socket();
6 socket.connect(new InetSocketAddress("site.ro", 80), 5000);
7 // timeout la CONECTARE: 5s
8 socket.setSoTimeout(10000);
9 // timeout la CITIRE ulterioara: 10s
```

Listing 20.2: Rezolvarea unei adrese, și configurarea unui timeout de citire

**Notă**

Fără un timeout explicit, o operație de citire pe un socket poate bloca thread-ul apelant la **infinit**, dacă serverul nu răspunde niciodată (conexiune întreruptă, server blocat, rețea căzută) — `setSoTimeout` garantează că citirea eșuează, cu `SocketTimeoutException`, după intervalul specificat, în loc să blocheze nedefinit. Tratați întotdeauna acest caz explicit (Subsecțiunea 9.2.1), mai ales în cod care gestionează conexiuni către servere externe, potențial nesigure.

## 20.2 Implementarea serverelor

### 20.2.1 ServerSocket: server de bază

Un `ServerSocket` ascultă pe un port dat, acceptând conexiuni de la clienți:

Listing 20.3: Un server minimal, care deserveste un singur client, apoi se oprește

```

1 try (ServerSocket serverSocket = new ServerSocket(5000)) {
2     System.out.println("Server pornit, astept conexiuni...");
3
4     try (Socket clientSocket = serverSocket.accept()) {
5         // BLOCHEAZA pana la o conexiune noua
6         BufferedReader dinClient = new BufferedReader(
7             new InputStreamReader(clientSocket.getInputStream()
8             ), StandardCharsets.UTF_8));
9         Writer catreClient = new OutputStreamWriter(
10            clientSocket.getOutputStream(),
11            StandardCharsets.UTF_8
12        );
13
14         String cerere = dinClient.readLine();
15         // de exemplu, "SOLD Ana Popescu"
16         String raspuns = proceseazaCerere(cerere);
17
18         catreClient.write(raspuns + "\n");
19         catreClient.flush();
20     }
21 }

```

**Notă**

`accept()` blochează thread-ul apelant până când un client se conectează, returnând apoi un `Socket` separat pentru comunicarea cu acel client specific — `ServerSocket` însuși este folosit doar pentru acceptarea de conexiuni noi, nu pentru schimbul efectiv de date.

### 20.2.2 Deservirea mai multor clienți

Serverul de mai sus deserveste un singur client, apoi se termină — nepotrivit pentru orice utilizare reală. Combinăm `ServerSocket` cu instrumentele de concurență din Capitolul 13, pentru a deservi mai mulți clienți simultan:

Listing 20.4: Un server care deserveste fiecare client pe propriul virtual thread

```

1 try (ServerSocket serverSocket = new ServerSocket(5000)) {
2     while (true) {
3         Socket clientSocket = serverSocket.accept();
4         // asteapta URMATORUL client

```

```

5
6      Thread.ofVirtual().start(() -> {
7          // fiecare client, pe propriul virtual thread
8          try (clientSocket) {
9              deservesteClient(clientSocket);
10         } catch (IOException e) {
11             LOG.log(Level.WARNING, "Eroare la deservirea
unui client", e);
12         }
13     });
14 }
15 }

```

#### Notă

Acest tipar — o buclă accept() pe thread-ul principal, care lansează câte un thread (astăzi, ideal, un *virtual thread*, Secțiunea 13.2) pentru fiecare client acceptat — este arhitectura clasică a unui server care deservește concurrent mulți clienți. Virtual threads sunt alegerea naturală aici: fiecare conexiune de client petrece cea mai mare parte a timpului **așteptând** date (I/O-bound, exact cazul de utilizare motivat în Subsecțiunea 13.2.1), iar costul redus per virtual thread permite mii de conexiuni simultane fără epuizarea resurselor sistemului, spre deosebire de câte un platform thread per client.

### 20.2.3 Comunicare securizată (SSL/TLS)

Pentru comunicare criptată, SSLSocket/SSLServerSocket (din pachetul `javax.net.ssl`) înlocuiesc direct Socket/ServerSocket obișnuite, cu aceeași interfață de bază:

```

1  SSLSocketFactory factory = (SSLSocketFactory)
   SSLSocketFactory.getDefault();
2  try (SSLSocket socketSecurizat = (SSLSocket) factory.
   createSocket("exemplu.ro", 443)) {
3      socketSecurizat.startHandshake();
4      // negociaza criptarea inainte de comunicare
5      // restul codului identic cu Socket obișnuit -
6      // getInputStream()/getOutputStream()
7  }

```

Listing 20.5: Un socket client securizat, prin SSL/TLS

#### Notă

Din perspectiva codului aplicație, un SSLSocket se folosește identic cu un Socket obișnuit — toată complexitatea criptării (schimb de chei, verificare de certificate) este gestionată intern de clasă. Pentru comunicare HTTP securizată (`https://`), API-ul `HttpClient` modern, discutat în Secțiunea 20.3, gestionează SSL/TLS complet automat, fără a mai fi nevoie să lucrați direct cu SSLSocket.

## 20.3 Obținerea de date web; HTTP Client modern

### 20.3.1 URL, URI și URLConnection (context istoric)

O distincție utilă: URI este pur **sintactic** (un identificator, conform unei sintaxe standard), în timp ce URL implică, în plus, un mecanism cunoscut de **localizare** a resursei (protocol, server, cale):

Listing 20.6: URLConnection, API-ul istoric pentru citirea unei resurse web

```
1 URL url = new URI("https://exemplu.ro/api/curs").toURL();
2 URLConnection conexiune = url.openConnection();
3
4 try (BufferedReader cititor = new BufferedReader(
5     new InputStreamReader(conexiune.getInputStream(),
6     StandardCharsets.UTF_8))) {
7     String continut = cititor.readLine();
8 }
```

#### Notă

URLConnection este API-ul istoric pentru acest scop — funcțional, dar verbos și limitat (fără suport nativ pentru HTTP/2, fără procesare asincronă convenabilă). Începând cu Java 11, HttpClient din secțiunea următoare este alegerea recomandată pentru orice comunicare HTTP nouă; URLConnection rămâne relevant mai ales pentru protocoale non-HTTP mai vechi sau cod existent.

### 20.3.2 HttpClient

HttpClient (din `java.net.http`, Java 11+) oferă un API modern, fluent, pentru cereri HTTP:

Listing 20.7: O cerere HTTP simplă, sincronă, cu HttpClient

```
1 HttpClient client = HttpClient.newHttpClient();
2
3 HttpRequest cerere = HttpRequest.newBuilder()
4     .uri(URI.create("https://exemplu.ro/api/curs-valutar"))
5     .header("Accept", "application/json")
6     .GET()
7     .build();
8
9 HttpResponse<String> raspuns = client.send(cerere,
10     HttpResponse.BodyHandlers.ofString());
11
12 System.out.println(raspuns.statusCode());
13 // 200, 404, 500 etc.
14 System.out.println(raspuns.body());
15 // continutul raspunsului, ca String
```

#### Notă

`HttpRequest.newBuilder()` folosește un stil *fluent* (fiecare metodă returnează `this`, permițând înlănțuirea apelurilor) — un tipar de proiectare des întâlnit în API-uri moderne Java, aici aplicat construcției unei cereri HTTP complete într-o singură expresie. `BodyHandlers.ofString()` indică cum se interpretează corpul răspunsului — alte variante includ `ofByteArray()` (date binare) sau `ofFile(path)` (salvare directă pe disc, utilă pentru descărcări mari).

### 20.3.3 Procesare asincronă

HttpClient oferă și o variantă asincronă, care returnează direct un `CompletableFuture` (Subsecțiunea 14.4.1), în locul blocării thread-ului apelant:

```
1 CompletableFuture<String> cursValutar = client
2   .sendAsync(cerere, HttpResponse.BodyHandlers.ofString())
3   .thenApply(HttpResponse::body); // extrage doar corpul
4   // raspunsului, din intreg HttpResponse
5
6 cursValutar.thenAccept(json -> System.out.println("Curs
   primit: " + json));
```

**Listing 20.8:** Cerere HTTP asincronă, compusă cu `CompletableFuture`

#### Notă

`sendAsync` combină direct tot ce am învățat despre `CompletableFuture` în Subsecțiunea 14.4.2: `thenApply` transformă răspunsul complet într-un simplu `String` (corpul), iar lanțul poate continua cu orice altă compunere (`thenCompose` pentru o a doua cerere dependentă, `thenCombine` pentru a agrega mai multe cereri paralele independente). Această integrare directă este exact motivul pentru care am introdus `CompletableFuture` riguros înainte de a ajunge la comunicarea în rețea — aplicațiile reale care fac multe cereri HTTP concurente se bazează pe acest stil neblocant.

## 20.4 Simple HTTP Server; trimiterea de e-mail

### 20.4.1 Simple HTTP Server (context)

JDK-ul include, pentru scopuri de testare și prototipare rapidă, un server HTTP minimal, `com.sun.net.httpserver.HttpServer` — simetric API-ului client din secțiunea anterioară:

```
1 HttpServer server = HttpServer.create(new InetSocketAddress
   (8080), 0);
2
3 server.createContext("/sold", exchange -> {
4   String raspuns = "Sold: " + gasesteSoldTitular("Ana
   Popescu");
5   exchange.sendResponseHeaders(200, raspuns.length());
6   try (OutputStream os = exchange.getResponseBody()) {
7     os.write(raspuns.getBytes(StandardCharsets.UTF_8));
8   }
9 });
10
11 server.start();
```

**Listing 20.9:** Un server HTTP minimal, care expune soldul unui cont

#### Notă

Acest server minimal este util pentru prototipare rapidă, teste locale sau exemple didactice — nu este gândit pentru producție (fără gestionare avansată a concurenței la scară largă, fără multe facilități de securitate prezente în servere HTTP dedicate). Pentru aplicații web reale, cadre de lucru precum Spring Boot sau servere de aplicații dedicate oferă infrastructura completă necesară, dincolo de scopul acestui curs introductiv.

## 20.4.2 Trimiterea de e-mail (context)

Trimiterea de e-mail din Java **nu** face parte din JDK însuși — necesită o bibliotecă externă, cel mai adesea Jakarta Mail (fosta JavaMail), adăugată explicit ca dependență a proiectului:

**Listing 20.10:** Structura tipică a trimiterii unui e-mail, cu Jakarta Mail (schematic)

```

1 Properties proprietati = new Properties();
2 proprietati.put("mail.smtp.host", "smtp.exemplu.ro");
3 proprietati.put("mail.smtp.auth", "true");
4
5 Session sesiune = Session.getInstance(proprietati);
6
7 Message mesaj = new MimeMessage(sesiune);
8 mesaj.setFrom(new InternetAddress("banca@exemplu.ro"));
9 mesaj.setRecipient(Message.RecipientType.TO, new
    InternetAddress("ana.popescu@exemplu.ro"));
10 mesaj.setSubject("Extras de cont");
11 mesaj.setText("Soldul dumneavoastra actual este: 500.0 lei");
12
13 Transport.send(mesaj);

```

### Notă

Codul de mai sus este **schematic** — necesită biblioteca Jakarta Mail adăugată explicit ca dependență (via Maven/Gradle, instrumente pe care le vom atinge doar tangențial în acest curs), configurare de server SMTP reală, și, de regulă, autentificare. Menționăm structura generală pentru completitudine și pentru a recunoaște tiparul (Session, Message, Transport) dacă îl întâlniți într-un proiect real.

### Rezumat

- ▶ Socket stabilește o conexiune TCP către un server; `ServerSocket.accept()` acceptă conexiuni de la clienți, câte un Socket separat per client.
- ▶ Un server real deservește clienți concurent, tipic printr-un thread (ideal, virtual thread) per conexiune acceptată; `setSoTimeout` previne blocarea nedefinită la citire.
- ▶ `SSLSocket/SSLServerSocket` adaugă criptare TLS, cu aceeași interfață ca socket-urile obișnuite.
- ▶ `HttpClient` (Java 11+) este API-ul modern recomandat pentru cereri HTTP, cu suport nativ sincron și asincron (`sendAsync`, integrat direct cu `CompletableFuture`).
- ▶ JDK-ul include un server HTTP minimal, potrivit pentru prototipare, nu pentru producție; trimiterea de e-mail necesită o bibliotecă externă (Jakarta Mail).

### Exerciții

1. Scrieți o pereche client-server minimală cu `Socket/ServerSocket`: clientul trimite un nume de titular, serverul răspunde cu soldul contului respectiv (dintr-o listă predefinită, în memoria serverului).
2. Extindeți serverul de mai sus să deservească mai mulți clienți simultan, folosind virtual threads, și testați-l cu cel puțin trei clienți conectați concurent.
3. Folosind `HttpClient`, scrieți o cerere GET către un API public la alegere (de exemplu, un API gratuit de curs valutar), și afișați

corpul răspunsului.

4. Rescrieți cererea de mai sus în stil asincron, cu `sendAsync` și `thenApply`, afișând rezultatul printr-un `thenAccept`.
5. Folosind `HttpServer` (JDK), expuneți un mic API care returnează, la cererea `GET /conturi`, o listă simplă de conturi în format text.



### Introducere

Am salvat datele conturilor, până acum, ca text (Capitolul 17), format binar (Capitolul 18) și XML (Capitolul 19) — toate potrivite pentru volume moderate de date, citite/scrise integral. O bază de date relațională rezolvă o problemă diferită: interogări eficiente peste volume mari de date, concurență controlată între mai mulți utilizatori, și garanții tranzacționale. *JDBC* (*Java Database Connectivity*) este API-ul standard prin care Java comunică cu orice bază de date relațională [2].

## 21.1 Designul JDBC; SQL — recapitulare

### 21.1.1 Designul JDBC

JDBC este proiectat exact pe principiul interfețelor discutat în Secțiunea 7.1: codul aplicație programează împotriva unui set de **interfețe** standard (*Connection*, *Statement*, *ResultSet*), în timp ce fiecare producător de bază de date furnizează propriul *driver* — o implementare concretă a acestor interfețe, specifică motorului de bază de date respectiv:

```
1 Connection conexiune = DriverManager.getConnection(url,  
    utilizator, parola);  
2 // 'conexiune' este o INTERFATA - implementarea reala vine  
3 // din driver-ul JDBC incarcat
```

**Listing 21.1:** Codul aplicație rămâne neschimbat, indiferent de baza de date folosită

### Notă

Această separare este motivul pentru care, teoretic, aplicația poate trece de la o bază de date la alta (de exemplu, de la H2 la PostgreSQL) schimbând doar URL-ul de conectare și driver-ul JAR adăugat la proiect, fără a rescrie codul care execută interogări SQL — exact beneficiul programării împotriva interfeței, nu a implementării, discutat prima dată în Subsecțiunea 11.1.1. În practică, diferențele subtile de dialect SQL între motoare de baze de date limitează adesea această portabilitate completă, dar structura API-ului rămâne identică.

### Notă

Driverule JDBC moderne sunt aproape universal *Type IV* — scrise integral în Java, comunicând direct cu baza de date prin propriul protocol de rețea, fără cod nativ intermediar. Clasificarea istorică în patru tipuri (I–IV) este menționată în documentația mai veche, dar rar relevantă pentru dezvoltarea de azi.

### 21.1.2 SQL — recapitulare

Presupunem o cunoaștere de bază a SQL; recapitulăm rapid instrucțiunile pe care le vom folosi, pentru un tabel `conturi`, corespunzător clasei `ContBancar`:

**Listing 21.2:** Definirea tabelului, și instrucțiunile SQL de bază

```

1 CREATE TABLE conturi (
2     id INT PRIMARY KEY AUTO_INCREMENT,
3     titular VARCHAR(100) NOT NULL,
4     sold DOUBLE NOT NULL
5 );
6
7 INSERT INTO conturi (titular, sold) VALUES ('Ana Popescu',
8     500.0);
9 SELECT * FROM conturi WHERE sold > 1000;
10 UPDATE conturi SET sold = sold + 150.0 WHERE titular = 'Ana
    Popescu';
11 DELETE FROM conturi WHERE sold = 0;

```

**Notă**

Coloana `id`, cu `AUTO_INCREMENT`, este o *cheie primară* generată automat de baza de date la fiecare `INSERT` — vom relua acest mecanism, din perspectiva codului Java, în Subsecțiunea 21.4.1. Restul acestui capitol presupune familiaritate cu sintaxa SQL de bază ilustrată mai sus; pentru o recapitulare completă a SQL, consultați materialul dedicat bazelor de date din programa facultății.

## 21.2 Configurare și conectare

### 21.2.1 URL-uri de bază de date; JAR-uri de driver

Un URL JDBC identifică baza de date țintă, cu un format specific driver-ului folosit:

**Listing 21.3:** Exemple de URL-uri JDBC, pentru diverse baze de date

```

1 jdbc:h2:./date/banca // H2, fisier local
2 jdbc:postgresql://localhost:5432/banca // PostgreSQL,
3 // server local
4 jdbc:mysql://localhost:3306/banca?useSSL=false // MySQL

```

**Notă**

Prefixul `jdbc:` este comun tuturor; segmentul următor (`h2`, `postgresql`, `mysql`) identifică driver-ul specific, care trebuie să fie prezent pe class path-ul aplicației (Subsecțiunea 5.1.3) — de obicei ca dependență de proiect (adăugată prin Maven/Gradle), rar ca JAR gestionat manual. Drivererele JDBC moderne se „înregistrează” automat la încărcare (prin `ServiceLoader`, menționat deja în Subsecțiunea 8.4.3), fără a mai fi nevoie de un apel explicit `Class.forName(...)`, necesar în cod Java mai vechi.

### 21.2.2 Conectare

**Listing 21.4:** Deschiderea unei conexiuni, cu `try-with-resources`

```

1 String url = "jdbc:h2:./date/banca";
2
3 try (Connection conexiune = DriverManager.getConnection(url,
4     "utilizator", "parola")) {
5     System.out.println("Conectat: " + !conexiune.isClosed());
6 } catch (SQLException e) {
7     System.out.println("Eroare la conectare: " + e.getMessage());
8 }

```

```
7 | }
```

#### Notă

Connection implementează AutoCloseable (la fel ca Statement și ResultSet, pe care le vom folosi în secțiunile următoare) — try-with-resources (Subsecțiunea 9.3.1) este forma idiomatică de gestionare a acestor resurse, garantând închiderea conexiunii chiar dacă apare o excepție în mijlocul unei operații. SQLException este o excepție verificată (Subsecțiunea 9.1.2) — aproape orice metodă JDBC o poate arunca, reflectând faptul că o bază de date este o resursă externă, predispusă la eșecuri independente de corectitudinea codului (conexiune întreruptă, permisiuni insuficiente, sintaxă SQL invalidă).

## 21.3 Statement-uri; PreparedStatement; excepții SQL

### 21.3.1 Statement: execuție de bază

```
1 | try (Statement stmt = conexiune.createStatement();
2 |     ResultSet rezultat = stmt.executeQuery("SELECT titular,
3 |     sold FROM conturi WHERE sold > 1000")) {
4 |
5 |     while (rezultat.next()) { // avanseaza la urmatorul rand
6 |         // -> FALSE cand nu mai sunt
7 |         // randuri
8 |         String titular = rezultat.getString("titular");
9 |         double sold = rezultat.getDouble("sold");
10 |        System.out.println(titular + ": " + sold);
11 |    }
12 | }
```

Listing 21.5: Execuția unei interogări, și parcurgerea rezultatului

#### Notă

executeQuery este pentru SELECT (returnează un ResultSet); executeUpdate este pentru INSERT/UPDATE/DELETE (returnează numărul de rânduri afectate, un int); execute generic tratează cazul în care tipul instrucțiunii nu e cunoscut dinainte. ResultSet.next() trebuie apelat înainte de prima citire — cursorul începe poziționat înaintea primului rând, nu pe el.

### 21.3.2 PreparedStatement

Construirea unei interogări prin concatenare directă de text este periculoasă — vulnerabilă la *injecție SQL*, dacă vreo valoare provine din intrare externă:

```
1 | String titularNesigur = "Ana"; DROP TABLE conturi; --";
2 | // input malitios, ipotetic
3 | String sqlNesigur = "SELECT * FROM conturi WHERE titular = '"
4 |     + titularNesigur + "'";
5 | // executarea acestei interogari ar putea sterge
6 | // intregul tabel!
```

Listing 21.6: PERICULOS: concatenare directă – vulnerabil la injecție SQL

```
1 | String sql = "SELECT * FROM conturi WHERE titular = ?";
2 |
```

Listing 21.7: SIGUR: PreparedStatement, cu parametri legați explicit

```

3 try (PreparedStatement pstmt = conexiune.prepareStatement(sql
4     )) {
5     pstmt.setString(1, titularNesigur);
6     // legat ca DATA, niciodata interpretat ca SQL
7
8     try (ResultSet rezultat = pstmt.executeQuery()) {
9         while (rezultat.next()) {
10             System.out.println(rezultat.getString("titular"))
11         }
12     }
13 }

```

#### Notă

PreparedStatement separă strict **structura** interogării (fixată la `prepareStatement`) de **valorile** ei (legate ulterior, prin `setString/setDouble/setInt` etc.) — driver-ul JDBC transmite valorile ca date pure către baza de date, niciodată interpretate ca parte din sintaxa SQL, eliminând complet riscul de injecție ilustrat mai sus. Folosiți **întotdeauna** PreparedStatement pentru orice interogare care include valori din afara codului sursă — fără excepție, indiferent cât de „sigură” pare sursa datelor.

Listing 21.8: PreparedStatement pentru actualizare, cu mai mulți parametri

```

1 String sqlUpdate = "UPDATE conturi SET sold = ? WHERE titular
2     = ?";
3
4 try (PreparedStatement pstmt = conexiune.prepareStatement(
5     sqlUpdate)) {
6     pstmt.setDouble(1, 650.0);
7     pstmt.setString(2, "Ana Popescu");
8
9     int randuriAfectate = pstmt.executeUpdate();
10 }

```

### 21.3.3 Excepții SQL

Am întâlnit deja `SQLException` în Subsecțiunea 21.2.2; detaliem acum informația suplimentară pe care o oferă:

Listing 21.9: Informații specifice `SQLException`, dincolo de mesajul obișnuit

```

1 try {
2     // ... operatie JDBC ...
3 } catch (SQLException e) {
4     System.out.println("Cod eroare: " + e.getErrorCode());
5     // specific driver-ului/bazei de date
6     System.out.println("Stare SQL: " + e.getSQLState());
7     // cod standard, portabil
8
9     SQLException urmatoarea = e.getNextException();
10    // exceptii SQL INLANTUITE, distinct de getCause()
11    while (urmatoarea != null) {
12        System.out.println("Cauzata si de: " + urmatoarea.
13            getMessage());
14        urmatoarea = urmatoarea.getNextException();
15    }
16 }

```

**Notă**

`getNextException()` este un mecanism de înlănțuire **separat** de `getCause()` din Subsecțiunea 9.2.3 — specific `SQLException`, folosit atunci când o singură operație JDBC generează **mai multe** erori simultan (de exemplu, la un batch update cu mai multe instrucțiuni eșuate, Subsecțiunea 21.5.3). `getSQLState()` returnează un cod standardizat (definit de standardul SQL, nu de un anumit driver), mai portabil între baze de date decât `getErrorCode()`, care e specific fiecărui motor de bază de date.

## 21.4 Chei autogenerate; metadata; result set-uri avansate

### 21.4.1 Chei autogenerate

Revenim la coloana `id` `AUTO_INCREMENT` din Subsecțiunea 21.1.2 — valoarea generată de baza de date la `INSERT` poate fi recuperată imediat, fără o interogare separată:

```

1 String sql = "INSERT INTO conturi (titular, sold) VALUES (?,
2     ?)";
3 try (PreparedStatement pstmt = conexiune.prepareStatement(sql
4     , Statement.RETURN_GENERATED_KEYS)) {
5     pstmt.setString(1, "Maria Popa");
6     pstmt.setDouble(2, 300.0);
7     pstmt.executeUpdate();
8
9     try (ResultSet chei = pstmt.getGeneratedKeys()) {
10        if (chei.next()) {
11            long idNou = chei.getLong(1); // id-ul generat
12                                           // automat de baza
13                                           // de date
14        }
15    }
16 }
```

**Listing 21.10:** Recuperarea cheii autogenerate, imediat după inserare

### 21.4.2 Metadata

`ResultSetMetaData` descrie **structura** unui rezultat — util pentru cod care procesează interogări cu structură necunoscută dinainte (de exemplu, un instrument generic de vizualizare a tabelor):

```

1 try (ResultSet rezultat = stmt.executeQuery("SELECT * FROM
2     conturi")) {
3     ResultSetMetaData metadata = rezultat.getMetaData();
4
5     for (int i = 1; i <= metadata.getColumnCount(); i++) {
6         System.out.println(metadata.getColumnName(i) + ": " +
7             metadata.getColumnTypeName(i));
8     }
9 }
```

**Listing 21.11:** Inspectarea structurii unui `ResultSet`, prin metadata

**Notă**

Coloanele unui `ResultSet` (ca și parametrii unui `PreparedStatement`) sunt indexate **de la 1**, nu de la 0 — o convenție moștenită din SQL însuși, diferită de indexarea obișnuită a array-urilor și listelor Java (Secțiunea 2.6, Subsecțiunea 3.1.2), o sursă frecventă de erori off-by-one pentru cineva obișnuit cu indexarea de la 0.

### 21.4.3 Result set-uri scrollabile și actualizabile; row sets

Un `ResultSet` obișnuit permite doar parcurgere **înainte**, o singură dată. Un `ResultSet` *scrollabil* permite navigare liberă (`previous()`, `first()`, `absolute(n)`); unul *actualizabil* permite modificarea directă a rândurilor, propagată automat în baza de date:

**Listing 21.12:** Un `ResultSet` scrollabil și actualizabil

```

1 try (Statement stmt = conexiune.createStatement(
2     ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.
3     CONCUR_UPDATABLE);
4     ResultSet rezultat = stmt.executeQuery("SELECT * FROM
5     conturi")) {
6
7     rezultat.last(); // sare direct la ULTIMUL rand
8     rezultat.updateDouble("sold", 999.0); // modifica valoarea
9     // rea in memorie
10    rezultat.updateRow(); // propaga modificarea in baza de
11    // date
12 }
```

**Notă**

Un `RowSet` (de exemplu, `CachedRowSet`) extinde conceptul de `ResultSet` cu funcționare *deconectată* — datele sunt încărcate integral în memorie, permițând operarea asupra lor **după** închiderea conexiunii la baza de date, utilă pentru transmiterea rezultatelor unei interogări către alt strat al aplicației, fără a menține o conexiune activă mai mult decât e necesar.

## 21.5 Tranzacții; batch updates; gestionarea conexiunilor

### 21.5.1 Tranzacții

Revenim la scenariul de transfer între conturi, discutat deja la nivel de thread-uri în Subsecțiunea 14.1.3 — de această dată, coordonat de baza de date printr-o *tranzacție*:

**Listing 21.13:** O tranzacție JDBC: ambele actualizări reușesc, sau niciuna

```

1 conexiune.setAutoCommit(false); // dezactiveaza commit-ul
2                                     // automat dupa fiecare
3                                     // instructiune
4
5 try {
6     try (PreparedStatement retrage = conexiune.
7         prepareStatement(
8             "UPDATE conturi SET sold = sold - ? WHERE titular
9             = ?")) {
10        retrage.setDouble(1, 100.0);
11        retrage.setString(2, "Ana Popescu");
12    }
```

```

10     retrage.executeUpdate();
11 }
12
13 try (PreparedStatement depune = conexiune.
14     prepareStatement(
15         "UPDATE conturi SET sold = sold + ? WHERE titular
16         = ?")) {
17     depune.setDouble(1, 100.0);
18     depune.setString(2, "Ion Ionescu");
19     depune.executeUpdate();
20 }
21
22 conexiune.commit(); // ambele actualizari devin
23                     // PERMANENTE, impreuna
24 } catch (SQLException e) {
25     conexiune.rollback(); // ORICE eroare - anuleaza AMBELE
26                           // actualizari, revenind la starea
27                           // initiala
28     throw e;
29 } finally {
30     conexiune.setAutoCommit(true); // revenire la comporta-
31                                     // mentul implicit
32 }

```

### Notă

Spre deosebire de coordonarea prin `synchronized/Lock` din Secțiunea 14.1–Secțiunea 14.2 (care previne accesul concurrent în interiorul aceluiași proces JVM), o tranzacție JDBC garantează *atomicitate* la nivelul bazei de date: ori ambele actualizări (retragere și depunere) reușesc împreună, ori — dacă oricare eșuează — niciuna nu rămâne aplicată, indiferent câți alți clienți accesează simultan aceeași bază de date. Implicit, `Connection` rulează în mod `auto-commit` (fiecare instrucțiune e propria ei tranzacție, confirmată imediat) — dezactivarea explicită, ca mai sus, este necesară ori de câte ori mai multe instrucțiuni trebuie tratate ca o singură unitate atomică.

## 21.5.2 Savepoints

Un *savepoint* permite anularea parțială a unei tranzacții, revenind la un punct intermediar, fără a anula tot ce s-a făcut de la începutul ei:

```

1 conexiune.setAutoCommit(false);
2
3 // ... prima actualizare, care ramane valida ...
4
5 Savepoint punctIntermediar = conexiune.setSavepoint();
6
7 try {
8     // ... a doua actualizare, riscanta ...
9 } catch (SQLException e) {
10     conexiune.rollback(punctIntermediar); // anuleaza DOAR a
11                                           // doua actualizare
12 }
13
14 conexiune.commit(); // confirma prima actualizare (si a
15                     // doua, daca nu a esuat)

```

Listing 21.14: Un *savepoint*, pentru anulare parțială în interiorul unei tranzacții

**Notă**

Savepoints sunt utile în tranzacții complexe, cu mai mulți pași independenți, unde eșecul unui pas nu trebuie să invalideze automat tot ce s-a realizat înaintea lui — o granularitate mai fină decât alegerea binară commit/rollback complet, ilustrată în Subsecțiunea 21.5.1.

### 21.5.3 Batch updates

Executarea individuală a multor instrucțiuni similare (de exemplu, inserarea a 1000 de tranzacții) implică un cost de rețea per instrucțiune. Un *batch* grupează mai multe instrucțiuni, trimise și executate împreună:

Listing 21.15: Inserare în lot (*batch*), pentru multe rânduri similare

```

1 try (PreparedStatement pstmt = conexiune.prepareStatement(
2     "INSERT INTO tranzactii (cont_id, tip, suma) VALUES
3     (?, ?, ?)")) {
4
5     for (ContBancar.Extras tranzactie : toateTranzactiile) {
6         pstmt.setInt(1, idulContului(tranzactie));
7         pstmt.setString(2, tranzactie.tip());
8         pstmt.setDouble(3, tranzactie.suma());
9         pstmt.addBatch(); // adauga la lot, NU executa inca
10    }
11
12    int[] rezultate = pstmt.executeBatch(); // executa TOATE
13                                           // instructiunile din lot, dintr-o data
14 }

```

**Notă**

`executeBatch()` returnează un array cu numărul de rânduri afectate de **fiecare** instrucțiune din lot, în ordine — util pentru a verifica individual care instrucțiuni au reușit. Dacă o instrucțiune din lot eșuează, driver-ul poate arunca `BatchUpdateException` (o subclasă de `SQLException`, Subsecțiunea 21.3.3), ale cărei rezultate parțiale (pentru instrucțiunile executate cu succes înaintea eșecului) rămân disponibile prin `getUpdateCounts()`.

### 21.5.4 Gestionarea conexiunilor

Deschiderea unei conexiuni noi (`DriverManager.getConnection`, din Subsecțiunea 21.2.2) este relativ costisitoare — nepotrivită pentru a fi repetată la fiecare cerere, într-o aplicație web sau enterprise care deservește mulți utilizatori concurenți. Un *pool de conexiuni* menține un set de conexiuni deja deschise, reutilizate între cereri:

**Notă**

Interfața `DataSource` (spre deosebire de `DriverManager`) este abstracția standard folosită de aplicații reale pentru a obține conexiuni dintr-un pool gestionat — biblioteci precum HikariCP implementează `DataSource` peste un pool eficient, integrat de regulă automat de cadrele de lucru enterprise (Spring, Jakarta EE). Codul care execută interogări (`Statement`, `PreparedStatement`, tranzacții) rămâne identic, indiferent dacă `Connection` provine dintr-un `DriverManager` simplu (potrivit pentru exemple didactice, ca în acest capitol) sau dintr-un pool de producție — doar modul de **obținere** a conexiunii diferă.

## Rezumat

- ▶ JDBC separă interfețele standard (Connection, Statement, ResultSet) de implementările specifice fiecărui driver; Connection trebuie deschisă cu try-with-resources, la fel ca Statement/ResultSet.
- ▶ PreparedStatement este obligatoriu pentru orice interogare cu valori externe, prevenind injecția SQL prin separarea structurii interogării de valorile ei.
- ▶ SQLException oferă getSQLState()/getErrorCode() și un lanț propriu (getNextException()), distinct de getCause() obișnuit.
- ▶ Cheile autogenerate se recuperează cu RETURN\_GENERATED\_KEYS; ResultSetMetaData descrie structura unui rezultat necunoscut dinainte; result set-uri scrollabile/actualizabile permit navigare liberă și modificare directă.
- ▶ Tranzacțiile (setAutoCommit(false), commit/rollback) garantează atomicitate la nivelul bazei de date; savepoints permit anulare parțială; batch updates reduc costul de rețea pentru multe instrucțiuni similare.
- ▶ Aplicațiile reale folosesc pooluri de conexiuni (DataSource), nu DriverManager direct, pentru a evita costul repetat al deschiderii de conexiuni noi.

## Exerciții

1. Creați o bază de date locală (de exemplu, H2) cu tabelul `conturi` din Subsecțiunea 21.1.2, apoi scrieți cod Java care inserează cinci conturi și le afișează pe toate cu o interogare `SELECT`.
2. Implementați transferul între două conturi (retragere dintr-unul, depunere în altul) ca o singură tranzacție, cu `rollback` explicit dacă soldul sursă ar deveni negativ.
3. Scrieți o metodă care inserează, printr-un batch update, un număr mare (de exemplu, 500) de tranzacții sintetice, și măsurați diferența de timp față de inserarea individuală a fiecăreia.
4. Folosind `ResultSetMetaData`, scrieți o metodă generică ce afișează conținutul oricărui `ResultSet` primit ca parametru, indiferent de tabelul din care provine, formatat ca un tabel text simplu.
5. Explicați, în cuvinte proprii, de ce `PreparedStatement` previne injecția SQL chiar și atunci când valoarea legată conține caractere speciale SQL (ghilimele, punct și virgulă), spre deosebire de concatenarea directă de text.



# **Interfețe grafice și tehnologii complementare**



## Introducere

Toate programele din acest curs au comunicat cu utilizatorul prin consolă — linie de text, intrare de la tastatură. Acest capitol și următorul introduc programarea cu interfețe grafice (GUI), folosind *Swing*, bibliotecă standard Java pentru aplicații desktop cu ferestre, butoane și alte componente vizuale.

## 22.1 Istoricul toolkit-urilor UI; JFrame

### 22.1.1 Istoricul toolkit-urilor UI în Java

Java a avut, de-a lungul timpului, mai multe biblioteci pentru interfețe grafice: AWT (*Abstract Window Toolkit*, 1995), componente native ale sistemului de operare, dar limitate și inconsecvente vizual între platforme; *Swing* (1997), componente desenate integral de Java însuși (*lightweight*), oferind aspect consecvent și control total; *JavaFX* (introdusă ulterior), un toolkit modern, dar eliminată din JDK-ul standard începând cu Java 11 și devenită proiect separat.

## Notă

Ediția manualului folosită pentru acest curs (a 14-a) reflectă exact această evoluție: **nu** mai include JavaFX, revenind la AWT/Swing ca soluție standard, disponibilă direct în JDK, fără dependențe suplimentare. *Swing* rămâne complet funcțional și întreținut activ, deși nu a mai primit funcționalități majore noi de câțiva ani — suficient de stabil și de bine documentat pentru scopurile acestui curs introductiv.

### 22.1.2 JFrame; proprietăți de fereastră

*JFrame* reprezintă o fereastră de nivel superior — punctul de plecare al oricărei aplicații *Swing*:

```
1 JFrame fereastră = new JFrame("Extras de cont");
2     // titlul ferestrei
3
4 fereastră.setSize(400, 200);
5 fereastră.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6     // închide APLICATIA la X
7
8 JLabel etichetaSold = new JLabel("Sold: 500.0 lei");
9 fereastră.add(etichetaSold);
10
11 fereastră.setLocationRelativeTo(null);
12     // centreaza fereastră pe ecran
13 fereastră.setVisible(true);
14     // FARA acest apel, fereastră ramane invizibilă!
```

Listing 22.1: O fereastră minimală, afișând soldul unui cont

Listing 22.2: Structurarea unei aplicații Swing ca subclasă de JFrame

## Notă

setDefaultCloseOperation controlează ce se întâmplă la apăsarea butonului de închidere al ferestrei — EXIT\_ON\_CLOSE termină întreaga aplicație; DISPOSE\_ON\_CLOSE închide doar fereastra curentă (util pentru aplicații cu mai multe ferestre); implicit, HIDE\_ON\_CLOSE doar ascunde fereastra, fără a o distruge sau a termina aplicația — rareori comportamentul dorit. setVisible(true) este ultimul pas obligatoriu: o fereastră Swing este invizibilă implicit, chiar dacă restul configurării e completă.

```

1 public class FereastrCont extends JFrame {
2     public FereastrCont(ContBancar cont) {
3         super("Extras de cont - " + cont.getTitular());
4         setSize(400, 200);
5         setDefaultCloseOperation(EXIT_ON_CLOSE);
6
7         add(new JLabel("Sold: " + cont.getSold() + " lei"));
8
9         setLocationRelativeTo(null);
10        setVisible(true);
11    }
12 }
```

## Notă

Extinderea directă a lui JFrame (moștenire, Secțiunea 6.1) este un tipar comun în cod Swing existent, deși nu întotdeauna cea mai curată alegere de design — consumă singura moștenire de clasă disponibilă (Subsecțiunea 13.1.1, aceeași observație făcută despre extinderea Thread). O alternativă la fel de validă este **compoziția**: o clasă proprie care **conține** un JFrame ca membru, în loc să-l extindă, păstrând libertatea de a extinde altceva la nevoie.

## 22.2 Desenare 2D; culoare, fonturi, imagini

### 22.2.1 Desenare de bază

Pentru desenare personalizată (grafice, forme geometrice — revenind la FormaGeometrica din Subsecțiunea 6.6.1), suprascriem paintComponent pe o componentă:

Listing 22.3: Desenare 2D de bază, prin suprascrierea paintComponent

```

1 public class PanouGrafic extends JPanel {
2     @Override
3     protected void paintComponent(Graphics g) {
4         super.paintComponent(g);
5         // OBLIGATORIU - curata fundalul, printre altele
6
7         g.drawLine(10, 10, 100, 100);
8         g.drawRect(50, 50, 80, 40);
9         g.fillOval(150, 50, 60, 60);
10        g.drawString("Sold: 500.0 lei", 20, 150);
11    }
12 }
```

**Notă**

Apelul `super.paintComponent(g)` este esențial — fără el, componenta nu-și mai curăță fundalul între redesenări succesive, rezultând artefacte vizuale acumulate. `paintComponent` este apelat **automat** de Swing, ori de câte ori componenta trebuie redesenată (la redimensionarea ferestrei, la suprapunerea temporară cu altă fereastră etc.) — nu îl apelați niciodată direct; pentru a solicita o redesenare explicită (de exemplu, după o modificare de date), apelați `repaint()`.

**22.2.2 Culoare și fonturi**

```

1  @Override
2  protected void paintComponent(Graphics g) {
3      super.paintComponent(g);
4
5      g.setColor(Color.BLUE);
6      g.fillRect(20, 20, 100, 50);
7
8      g.setColor(Color.RED);
9      g.setFont(new Font("SansSerif", Font.BOLD, 16));
10     g.drawString("Sold: 500.0 lei", 20, 100);
11 }

```

**Listing 22.4:** Configurarea culorii și fontului, înaintea operațiilor de desenare

**Notă**

`setColor/setFont` modifică starea **curentă** a obiectului `Graphics` — toate operațiile de desenare ulterioare (până la următoarea schimbare) folosesc acea culoare/font, similar cu un „stilou” care rămâne configurat până e schimbat explicit. `Color` oferă constante predefinite (`Color.RED`, `Color.BLUE` etc.) și constructori pentru culori personalizate (`new Color(r, g, b)`, valori 0–255).

**22.2.3 Imagini**

```

1  Image logo = ImageIO.read(new File("logo_banca.png"));
2      // vezi Cap. 17 pentru I/O de fisiere
3
4  @Override
5  protected void paintComponent(Graphics g) {
6      super.paintComponent(g);
7      g.drawImage(logo, 10, 10, this); // 'this' - componenta
8                                      // observatoare a
9                                      // incarcarii
10 }

```

**Listing 22.5:** Încărcarea și desenarea unei imagini

**Notă**

Al patrulea parametru al lui `drawImage` (`ImageObserver`, aici pur și simplu `this`) este relevant mai ales pentru imagini încărcate asincron, unde componenta trebuie notificată pe măsură ce datele imaginii devin disponibile — pentru imagini încărcate complet, sincron, ca mai sus (prin `ImageIO.read`, care blochează până la finalizare), rolul său este minor, dar parametrul rămâne obligatoriu în semnătura metodei.

## 22.3 Gestionarea evenimentelor

### 22.3.1 Listeners; clase adapter

O aplicație grafică reacționează la acțiunile utilizatorului prin *listeners* — interfețe (Secțiunea 7.1) implementate de codul aplicație, înregistrate pe o componentă:

**Listing 22.6:** Un ActionListener, pentru un buton, exprimat ca lambda

```
1 JButton butonDepunere = new JButton("Depune 100 lei");
2
3 butonDepunere.addActionListener(e -> {
4     cont.depune(100.0);
5     etichetaSold.setText("Sold: " + cont.getSold() + " lei");
6 });
```

#### Notă

ActionListener are o singură metodă abstractă (`actionPerformed`), deci este o interfață funcțională (Subsecțiunea 7.2.2) — lambda-ul de mai sus înlocuiește elegant o clasă anonimă, exact tiparul deja familiar din Subsecțiunea 7.1.4. Actualizarea etichetei (`setText`) direct în interiorul listener-ului este sigură aici fiindcă listener-ul rulează, la rândul lui, pe Event Dispatch Thread — același thread care a creat componentele (Subsecțiunea 22.1.2).

Pentru interfețe cu **mai multe** metode abstracte (de exemplu, `WindowListener`, cu șapte metode), o *clasă adapter* oferă implementări implicite goale pentru toate, permițând suprascrierea doar a celor de interes:

**Listing 22.7:** `WindowAdapter`, pentru a suprascrie o singură metodă dintr-o interfață mare

```
1 fereastră.addWindowListener(new WindowAdapter() {
2     // extinde adapter-ul, NU implementeaza interfata direct
3     @Override
4     public void windowClosing(WindowEvent e) {
5         System.out.println("Fereastră se inchide...");
6     }
7 });
```

#### Notă

Fără `WindowAdapter`, implementarea directă a `WindowListener` ar obliga la scrierea tuturor celor șapte metode, chiar dacă majoritatea rămân goale — clasa adapter (o clasă abstractă, Subsecțiunea 6.6.1, cu toate metodele deja implementate trivial) elimină acest cod repetitiv, lăsând doar metodele relevante de suprascris.

### 22.3.2 Action

O `Action` încapsulează **simultan** comportamentul unui listener și meta-datele asociate (text, iconiță, stare activă/inactivă) — reutilizabilă pe mai multe componente deodată:

**Listing 22.8:** O `Action`, reutilizată pentru un buton și un element de meniu

```
1 Action actiuneDepunere = new AbstractAction("Depune 100 lei")
2 {
3     @Override
4     public void actionPerformed(ActionEvent e) {
5         cont.depune(100.0);
6     }
7 };
```

```

7
8 JButton buton = new JButton(actiuneDepunere);
9 // textul butonului vine din Action
10 JMenuItem itemMeniu = new JMenuItem(actiuneDepunere);
11 // ACEEASI actiune, alta componenta

```

#### Notă

Dacă mai târziu dezactivați acțiunea (prin apelul metodei `actiuneDepunere.setEnabled(false)`), **ambele** componente — butonul și elementul de meniu — se dezactivează automat împreună, fără cod suplimentar. Acest tipar este util ori de câte ori aceeași operație logică trebuie declanșată din mai multe locuri ale interfeței (buton, meniu, scurtătură de tastatură), păstrând starea și comportamentul sincronizate implicit.

### 22.3.3 Evenimente de mouse

```

1 panouGrafic.addMouseListener(new MouseAdapter() {
2     @Override
3     public void mouseClicked(MouseEvent e) {
4         System.out.println("Clic la (" + e.getX() + ", " + e.
5             getY() + ")");
6     }
7 });

```

Listing 22.9: Reacție la clic de mouse, pe un panou de desenare

#### Notă

`MouseAdapter` urmează același tipar ca `WindowAdapter` din Subsecțiunea 22.3.1 — `MouseListener` are cinci metode (`mouseClicked`, `mousePressed`, `mouseReleased`, `mouseEntered`, `mouseExited`), rareori toate necesare simultan. Coordonatele evenimentului (`e.getX()`, `e.getY()`) sunt relative la componenta pe care a fost înregistrat listener-ul, nu la ecran în întregime.

## 22.4 Swing, Model-View-Controller, layout managers

### 22.4.1 Swing și Model-View-Controller

Componentele Swing urmează, intern, tiparul *Model-View-Controller* (MVC): datele (*model*) sunt separate de reprezentarea lor vizuală (*view*), iar interacțiunea utilizatorului (*controller*) actualizează modelul, care notifică apoi vizualizarea să se redeseneze.

```

1 public class ModelSold {
2     private double sold;
3     private List<Runnable> ascultatori = new ArrayList<>();
4
5     public void seteazaSold(double nouSold) {
6         this.sold = nouSold;
7         ascultatori.forEach(Runnable::run);
8         // notifica toti ascultatorii de schimbare
9     }
10
11     public void adaugaAscultator(Runnable ascultator) {

```

Listing 22.10: Separarea datelor (model) de componenta vizuală (view)

```

12         ascultatori.add(ascultator);
13     }
14
15     public double getSold() {
16         return sold;
17     }
18 }

```

#### Notă

Această separare pare, la prima vedere, cod suplimentar inutil pentru o aplicație simplă — beneficiul devine clar când **mai multe** componente vizuale trebuie să reflecte aceeași dată (de exemplu, soldul afișat simultan într-o etichetă și într-un grafic): fiecare se înregistrează ca ascultător al aceluiași model, fără nicio legătură directă între ele, doar prin modelul partajat. Componentele Swing complexe (JTable, JTree, pe care le vom întâlni în Capitolul 23) expun explicit propriile modele (TableModel, TreeModel), urmând exact acest tipar.

## 22.4.2 Layout managers

Un *layout manager* decide poziția și dimensiunea componentelor dintr-un container, adaptându-se automat la redimensionare — spre deosebire de poziționarea manuală, pe coordonate fixe:

Listing 22.11: BorderLayout: cinci zone, nord/sud/est/vest/centru

```

1 fereastră.setLayout(new BorderLayout());
2 fereastră.add(new JLabel("Banca Exemplu"), BorderLayout.NORTH);
3 fereastră.add(panouGrafic, BorderLayout.CENTER);
4 // zona CENTER se extinde, umplind spațiul rămas
5 fereastră.add(new JButton("Iesire"), BorderLayout.SOUTH);

```

Listing 22.12: GridLayout: grilă uniformă de rânduri și coloane

```

1 JPanel panouButoane = new JPanel(new GridLayout(2, 2));
2 // 2 rânduri, 2 coloane, EGAL
3 panouButoane.add(new JButton("Depunere"));
4 panouButoane.add(new JButton("Retragere"));
5 panouButoane.add(new JButton("Istoric"));
6 panouButoane.add(new JButton("Inchide"));

```

Listing 22.13: GridBagLayout: control fin, pentru aranjamente complexe

```

1 JPanel panou = new JPanel(new GridBagLayout());
2 GridBagConstraints c = new GridBagConstraints();
3
4 c.gridx = 0; c.gridy = 0;
5 panou.add(new JLabel("Titular:"), c);
6
7 c.gridx = 1; c.gridy = 0;
8 c.fill = GridBagConstraints.HORIZONTAL;
9 // campul de text se extinde pe orizontala
10 panou.add(new JTextField(20), c);

```

#### Notă

BorderLayout și GridLayout acoperă majoritatea aranjamentelor simple, cu efort minim; GridBagLayout, deși semnificativ mai verbos (fiecare componentă necesită propriul obiect GridBagConstraints, configurat explicit), oferă control complet asupra poziției, dimensiunii relative și comportamentului la redimensionare al fiecărei componente — alegerea potrivită atunci când BorderLayout/GridLayout nu pot exprima aranjamentul dorit. Containerele Swing pot fi **imbri-**

cate (un JPanel cu propriul layout, adăugat ca o singură componentă într-un alt container), combinând layout-uri simple pentru a construi interfețe complexe, în locul unui singur GridBagLayout monolitic.

### Rezumat

- ▶ Java a trecut de la AWT (componente native) la Swing (componente *lightweight*, desenate de Java) la JavaFX și, în ediția curentă a manualului, înapoi la Swing.
- ▶ JFrame este fereastra de nivel superior; setVisible(true) trebuie apelat ultimul; operațiile Swing trebuie să ruleze pe Event Dispatch Thread.
- ▶ Desenarea personalizată se face prin suprascrierea paintComponent pe un JPanel, cu apel obligatoriu al super.paintComponent(g).
- ▶ Listeners (adesea interfețe funcționale, exprimabile ca lambda) reacționează la evenimente; clasele adapter elimină codul repetitiv pentru interfețe cu multe metode; Action unifică comportamentul și metadatele unei operații pe mai multe componente.
- ▶ Swing urmează tiparul Model-View-Controller; layout managers (BorderLayout, GridLayout, GridBagLayout) organizează automat componentele într-un container.

### Exerciții

1. Construiți o fereastră cu un JLabel afișând soldul unui cont și două butoane („Depune”, „Retrage”), care actualizează eticheta la apăsare.
2. Desenați, într-un JPanel personalizat, o reprezentare grafică simplă a soldurilor mai multor conturi, ca bare orizontale de lungimi proporționale.
3. Adăugați un MouseListener unui panou, care afișează coordonatele fiecărui clic direct pe panou (redesenat la fiecare clic, cu repaint()).
4. Construiți un formular simplu (titular, sold inițial) folosind GridBagLayout, cu etichete aliniate pe o coloană și câmpuri de text pe cealaltă.
5. Creați o Action comună pentru „Ieșire din aplicație”, folosită atât ca element de meniu, cât și ca buton, și verificați că dezactivarea acțiunii afectează ambele componente simultan.



## Introducere

Am construit, în capitolul anterior, o fereastră capabilă să deseneze și să reacționeze la evenimente. Acest capitol adaugă componentele Swing concrete pentru interacțiune structurată: câmpuri de text, componente de selecție, meniuri, și dialoguri — suficiente pentru a construi un formular complet de deschidere a unui cont bancar.

## 23.1 Input de text

### 23.1.1 JTextField, JPasswordField

```
1 JTextField campTitular = new JTextField(20);
2 // 20 - latimea preferata, in coloane de caractere
3 JPasswordField campPin = new JPasswordField(4);
4
5 JButton butonCreeaza = new JButton("Creeaza cont");
6 butonCreeaza.addActionListener(e -> {
7     String titular = campTitular.getText();
8     char[] pin = campPin.getPassword();
9     // NU getText() - vezi nota de mai jos
10
11     // ... foloseste titular si pin pentru a crea contul ...
12
13     Arrays.fill(pin, ' ');
14     // sterge PIN-ul din memorie, imediat dupa folosire
15 });
```

Listing 23.1: Un formular simplu, cu câmp de text și câmp de parolă

## Notă

`JPasswordField.getPassword()` returnează `char[]`, nu `String` — o distincție intenționată de securitate: un `String` este imutabil (Subsecțiunea 3.1.3) și rămâne în memorie, potențial vizibil în memory dumps, până la colectarea gunoiului, fără control explicit asupra momentului; un `char[]` poate fi **suprascris explicit** (`Arrays.fill`, Subsecțiunea 2.6.4) imediat după utilizare, minimizând fereastra de timp în care date sensibile rămân accesibile în memorie.

### 23.1.2 JTextArea; scroll panes

Pentru text pe mai multe linii (de exemplu, istoricul tranzacțiilor unui cont), `JTextArea` este componenta potrivită — dar, spre deosebire de `JTextField`, nu oferă scroll propriu:

```
1 JTextArea zonaIstoric = new JTextArea(10, 30);
2 // 10 randuri, 30 coloane, dimensiune preferata
3 zonaIstoric.setEditable(false);
4 // doar afisare, utilizatorul nu poate modifica istoricul
5
6 for (ContBancar.Extras tranzactie : cont.getIstoricTranzactii()) {
```

Listing 23.2: `JTextArea`, cu scroll explicit prin `JScrollPane`

```

7      zonaIstoric.append(tranzactie.tip() + ": " + tranzactie.
8      suma() + " lei\n");
9  }
10 JScrollPane panouCuScroll = new JScrollPane(zonaIstoric);
11     // adauga bare de scroll, dupa nevoie
12 fereastra.add(panouCuScroll);

```

#### Notă

JScrollPane „înfașoară” orice componentă (nu doar JTextArea) — același tipar de compoziție întâlnit deja la filtrele I/O din Subsecțiunea 17.1.2: o componentă adaugă o capabilitate (aici, scroll) peste alta, fără a modifica componenta originală. Barele de scroll apar automat doar atunci când conținutul depășește spațiul disponibil, nu permanent.

## 23.2 Componente de selecție

### 23.2.1 Checkbox-uri și radio buttons

Un JCheckBox reprezintă o alegere independentă, binară; mai multe JRadioButton grupate într-un ButtonGroup formează o selecție **exclusivă**, dintr-un set de opțiuni:

Listing 23.3: JCheckBox independent, versus JRadioButton grupate exclusiv

```

1 JCheckBox bifaContActiv = new JCheckBox("Cont activ", true);
2     // independent, poate fi bifat/nebifat
3
4 JRadioButton optiuneStandard = new JRadioButton("Standard",
5     true);
6     // selectat implicit
7 JRadioButton optiuneEconomii = new JRadioButton("Economii");
8 JRadioButton optiunePremium = new JRadioButton("Premium");
9
10 ButtonGroup grupTipCont = new ButtonGroup();
11     // garanteaza EXCLUSIVITATE - doar unul selectat
12 grupTipCont.add(optiuneStandard);
13 grupTipCont.add(optiuneEconomii);
14 grupTipCont.add(optiunePremium);

```

#### Notă

ButtonGroup **nu** este un container vizual (nu se adaugă direct la fereastră) — este un obiect de coordonare pur logică, care garantează că selectarea unui radio button din grup deselectează automat pe oricare altul din același grup. Componentele trebuie totuși adăugate individual la un container vizual (de exemplu, un JPanel), separat de înregistrarea în ButtonGroup.

### 23.2.2 JComboBox

Pentru o listă de opțiuni prea lungă pentru radio buttons individuale, JComboBox oferă o listă derulantă. Populăm una direct din valorile enumului TipCont (Subsecțiunea 6.6.2):

Listing 23.4: JComboBox, populat direct dintr-un tip enum

```

1 JComboBox<TipCont> selectorTipCont = new JComboBox<>(TipCont.
2     values());
3 selectorTipCont.setSelectedItem(TipCont.STANDARD);

```

```

3
4 selectorTipCont.addActionListener(e -> {
5     TipCont ales = (TipCont) selectorTipCont.getSelectedItem
6     ();
7     System.out.println("Tip selectat: " + ales + ", rata: " +
8     ales.getRataDobanda());
9 });

```

#### Notă

JComboBox<TipCont>(TipCont.values()) funcționează direct fiindcă JComboBox apelează implicit toString() pe fiecare element pentru afișare — iar reprezentarea implicită a unei constante enum este chiar numele ei (STANDARD, ECONOMII, PREMIUM), fără a fi nevoie de nicio conversie explicită. getSelectedItem() returnează Object (componenta e generică doar parțial, din motive istorice de compatibilitate), motiv pentru care cast-ul explicit la TipCont rămâne necesar.

### 23.2.3 Slidere

```

1 JSlider sliderDobanda = new JSlider(0, 100, 20);
2 // min 0, max 100, valoare initiala 20 (reprezinta 2.0%)
3 sliderDobanda.setMajorTickSpacing(25);
4 sliderDobanda.setPaintTicks(true);
5 sliderDobanda.setPaintLabels(true);
6
7 sliderDobanda.addChangeListener(e -> {
8     double rata = sliderDobanda.getValue() / 1000.0;
9     // 20 -> 0.02, ca in TipCont
10    System.out.println("Rata selectata: " + rata);
11 });

```

Listing 23.5: JSlider, pentru selectarea unei rate în interiorul unui interval

#### Notă

JSlider lucrează exclusiv cu valori int — pentru o rată zecimală (ca 0.02 din TipCont), este necesară o conversie explicită între scala întreagă a sliderului (aici, 0–100) și valoarea zecimală reală dorită, exact ca în exemplul de mai sus. ChangeListener (nu ActionListener) este interfața specifică pentru componente cu valori continue, notificată la fiecare modificare, inclusiv în timpul tragerii cursorului, nu doar la eliberare.

## 23.3 Meniuri, toolbar-uri, tooltips

### 23.3.1 Meniuri

```

1 JMenuBar bara = new JMenuBar();
2 JMenu meniuFisier = new JMenu("Fisier");
3
4 JMenuItem itemContNou = new JMenuItem("Cont nou");
5 itemContNou.setMnemonic('C'); // Alt+C, pe majoritatea
6 // platformelor
7 itemContNou.addActionListener(e -> deschideDialogContNou());
8
9 JMenuItem itemIesire = new JMenuItem("Iesire");
10 itemIesire.addActionListener(e -> System.exit(0));
11

```

Listing 23.6: O bară de meniu, cu un meniu și câteva elemente

```

12 | meniuFisier.add(itemContNou);
13 | meniuFisier.addSeparator(); // o linie despartitoare vizuala
14 | meniuFisier.add(itemIesire);
15 |
16 | bara.add(meniuFisier);
17 | fereastra.setJMenuBar(bara); // METODA SPECIALA - nu add()
18 |                               // obisnuit, ca la alte
19 |                               // componente

```

#### Notă

setJMenuBar este o metodă specifică lui JFrame (nu add obișnuit) — bara de meniu ocupă o poziție structurală fixă, deasupra restului conținutului ferestrei, gestionată separat de layout managerul containerului principal (Subsecțiunea 22.4.2). setMnemonic activează navigarea din tastatură (Alt+litera subliniată), o cerință de accesibilitate des trecută cu vederea în aplicații didactice, dar importantă în cod de producție.

### 23.3.2 Toolbar-uri

Un JToolBar oferă acces rapid, vizual, la operații frecvente — adesea aceleași Action (Subsecțiunea 22.3.2) deja folosite în meniu:

**Listing 23.7:** Un toolbar, reutilizând aceeași Action ca meniul și un buton obișnuit

```

1 | JToolBar bara2 = new JToolBar();
2 | bara2.add(actiuneDepunere); // ACEEASI Action din Sec. 22.3.2
3 |                               // - reutilizata a treia oara
4 | fereastra.add(bara2, BorderLayout.NORTH);

```

#### Notă

Această reutilizare — aceeași Action pe buton, meniu și toolbar — este exact beneficiul practic promis în Subsecțiunea 22.3.2: o singură sursă de adevăr pentru text, iconiță, stare activă/inactivă și comportament, sincronizată automat peste toate cele trei prezentări vizuale.

### 23.3.3 Tooltips

**Listing 23.8:** Un tooltip, afișat la staționarea cursorului deasupra componentei

```

1 | campTitular.setToolTipText("Introduceți numele complet al
   | titularului contului");

```

#### Notă

setToolTipText este disponibil pe orice componentă Swing (JComponent), fără configurare suplimentară — Swing gestionează automat afișarea temporizată, poziționarea și dispariția tooltip-ului. Folosiți tooltips pentru clarificări scurte, opționale, nu ca substitut al unei etichete vizibile permanent pentru informație esențială.

## 23.4 Dialoguri

### 23.4.1 JOptionPane

Pentru dialoguri simple — mesaje, confirmări, o singură valoare de intrare — JOptionPane oferă metode statice convenabile, fără a construi

manual o fereastră:

```

1 JOptionPane.showMessageDialog(fereastra, "Contul a fost creat
    cu succes.");
2
3 int raspuns = JOptionPane.showConfirmDialog(
4     fereastra, "Sigur doriti sa stergeti contul?", "
5     Confirmare", JOptionPane.YES_NO_OPTION);
6 if (raspuns == JOptionPane.YES_OPTION) {
7     // ... stergere ...
8 }
9 String suma = JOptionPane.showInputDialog(fereastra, "
10     Introduceti suma de depus:");
11 if (suma != null) {
12     // null daca utilizatorul apasa Anuleaza, nu OK
13     cont.depune(Double.parseDouble(suma));
14 }

```

Listing 23.9: Cele trei forme cele mai frecvente de JOptionPane

#### Notă

Primul parametru (fereastra) este componenta *părinte* — dialogul este centrat față de ea și rămâne **modal** implicit (blochează interacțiunea cu restul aplicației până la închidere). showInputDialog returnează null dacă utilizatorul anulează, spre deosebire de un șir gol dacă a apăsă OK fără să introducă nimic — distincție importantă de verificat explicit, similar cu Optional din Secțiunea 15.4, deși aici mecanismul e un simplu null, nu un container tipizat.

### 23.4.2 Dialoguri personalizate (JDialog)

Pentru un formular mai complex decât JOptionPane poate oferi — de exemplu, formularul de creare cont din Secțiunea 23.1–Secțiunea 23.2 — JDialog oferă o fereastră proprie, dependentă de fereastra principală:

```

1 public class DialogContNou extends JDialog {
2     public DialogContNou(JFrame parinte) {
3         super(parinte, "Cont nou", true);
4         // 'true' - MODAL, blocheaza fereastra parinte
5
6         setLayout(new GridBagLayout());
7         // ... adauga campTitular, campPin, selectorTipCont
8         // din sectiunile anterioare ...
9
10        JButton butonCreeaza = new JButton("Creeaza");
11        butonCreeaza.addActionListener(e -> {
12            // ... creeaza efectiv contul ...
13            dispose(); // inchide DOAR acest dialog, nu
14                       // toata aplicatia
15        });
16
17        pack(); // dimensioneaza fereastra automat, dupa
18               // continutul componentelor
19        setLocationRelativeTo(parinte);
20    }
21 }

```

Listing 23.10: Un JDialog personalizat, cu formularul complet de creare cont

**Notă**

Al treilea argument al constructorului `super(parinte, titlu, modal)` controlează modalitatea — `true` blochează complet interacțiunea cu fereastra părinte până la închiderea dialogului, comportamentul dorit pentru majoritatea formularelor care necesită date complete înainte de a continua. `dispose()` închide doar fereastra curentă (analog `DISPOSE_ON_CLOSE` din Subsecțiunea 22.1.2), spre deosebire de `System.exit(0)`, care ar termina întreaga aplicație.

### 23.4.3 Dialoguri de fișiere (JFileChooser)

Listing 23.11: Selectarea unui fișier, prin dialogul standard al sistemului de operare

```

1 JFileChooser selector = new JFileChooser();
2 int rezultat = selector.showOpenDialog(fereastra);
3
4 if (rezultat == JFileChooser.APPROVE_OPTION) {
5     File fisierAles = selector.getSelectedFile();
6     // ... foloseste fisierAles, de exemplu cu
7     // Files.readString (Sec. 17.4.2) ...
8 }

```

**Notă**

`JFileChooser` afișează dialogul **nativ** de selecție a fișierelor al sistemului de operare curent — aspect consecvent cu restul aplicațiilor de pe acel sistem, spre deosebire de restul componentelor Swing, desenate integral de Java (Subsecțiunea 22.1.1). `showSaveDialog` oferă varianta simetrică, pentru alegerea unei locații de salvare, cu aceeași convenție de rezultat (`APPROVE_OPTION/CANCEL_OPTION`).

## 23.5 JTable și JTree (introducere)

### 23.5.1 JTable

`JTable` afișează date tabelare, urmând strict tiparul Model-View-Controller introdus în Subsecțiunea 22.4.1 — datele nu se dau direct componentei, ci printr-un `TableModel` intermediar:

Listing 23.12: Un `TableModel` propriu, peste lista de conturi a băncii

```

1 public class ModelConturi extends AbstractTableModel {
2     private List<ContBancar> conturi;
3     private String[] coloane = { "Titular", "Sold" };
4
5     public ModelConturi(List<ContBancar> conturi) {
6         this.conturi = conturi;
7     }
8
9     @Override
10    public int getRowCount() { return conturi.size(); }
11
12    @Override
13    public int getColumnCount() { return coloane.length; }
14
15    @Override
16    public String getColumnName(int col) { return coloane[col]; }
17
18    @Override

```

```

19     public Object getValueAt(int rand, int coloana) {
20         ContBancar cont = conturi.get(rand);
21         return coloana == 0 ? cont.getTitular() : cont.
           getSold();
22     }
23 }

1 JTable tabelConturi = new JTable(new ModelConturi(conturi));
2 fereastra.add(new JScrollPane(tabelConturi));
3 // JTable, la fel ca JTextArea, necesita scroll pane

```

Listing 23.13: Folosirea modelului propriu, pentru afișarea listei de conturi

#### Notă

Cele patru metode suprascrise (getRowCount, getColumnCount, getColumnName, getValueAt) sunt **singurele** strict necesare — AbstractTableModel (o clasă abstractă, Subsecțiunea 6.6.1, exact tiparul deja familiar din Subsecțiunea 22.3.1) oferă implementări implicite rezonabile pentru restul interfeței TableModel. Deoarece tabelConturi referă direct lista conturi, orice actualizare a listei (urmată de un apel explicit fireTableDataChanged() pe model) se reflectă automat în tabelul afișat — exact mecanismul de notificare model→vizualizare discutat conceptual în Subsecțiunea 22.4.1, aplicat acum unei componente reale din bibliotecă.

### 23.5.2 JTree (context)

JTree afișează date ierarhice — utilă, de exemplu, pentru o organizare a conturilor pe categorii:

```

1 DefaultMutableTreeNode radacina = new DefaultMutableTreeNode(
   "Toate conturile");
2
3 DefaultMutableTreeNode nodStandard = new
   DefaultMutableTreeNode("Standard");
4 nodStandard.add(new DefaultMutableTreeNode("Ion Ionescu"));
5
6 DefaultMutableTreeNode nodEconomii = new
   DefaultMutableTreeNode("Economii");
7 nodEconomii.add(new DefaultMutableTreeNode("Ana Popescu"));
8
9 radacina.add(nodStandard);
10 radacina.add(nodEconomii);
11
12 JTree arbore = new JTree(radacina);
13 fereastra.add(new JScrollPane(arbore));

```

Listing 23.14: Un JTree minimal, cu structură ierarhică simplă

#### Notă

La fel ca JTable, JTree este construit peste propriul model (TreeModel), implicit derivat automat din structura de DefaultMutableTreeNode folosită mai sus — pentru date ierarhice mai complexe sau generate dinamic, un TreeModel propriu, analog ModelConturi de mai sus, oferă control complet. Menționăm JTree aici mai ales introductiv, ca element de recunoaștere — o tratare completă (editare de noduri, randare personalizată) depășește scopul acestui curs introductiv.

### Rezumat

- ▶ `JTextField`/`JPasswordField` preiau text pe o linie (parola ca `char[]`, din motive de securitate); `JTextArea` necesită `JScrollPane` explicit pentru scroll.
- ▶ `JCheckBox` e independent; `JRadioButton` grupate în `ButtonGroup` sunt exclusive; `JComboBox` se populează direct dintr-un `enum`; `JSlider` lucrează doar cu valori înt.
- ▶ Meniurile (`JMenuBar`/`JMenu`/`JMenuItem`) și toolbar-urile (`JToolBar`) pot reutiliza aceleași obiecte `Action`, sincronizate automat.
- ▶ `JOptionPane` oferă dialoguri rapide predefinite; `JDialog` permite formulare complet personalizate, modale sau nu; `JFileChooser` oferă dialogul nativ de selecție a fișierelor.
- ▶ `JTable`/`JTree` urmează strict Model-View-Controller, prin `TableModel`/`TreeModel` proprii sau predefinite.

### Exerciții

1. Construiți formularul complet de creare cont (titular, PIN, tip cont via `JComboBox`), într-un `JDialog` modal, care afișează un `JOptionPane` de confirmare la creare reușită.
2. Adăugați un meniu „Cont” cu elementele „Depune”, „Retrage”, „Ieșire”, fiecare declanșând acțiunea corespunzătoare deja implementată în exercițiile capitolelor anterioare.
3. Afișați lista completă de conturi ale băncii într-un `JTable`, folosind un `TableModel` propriu, și adăugați o a treia coloană, calculată, pentru tipul contului.
4. Adăugați tooltips descriptive pentru fiecare câmp al formularului de creare cont construit la exercițiul 1.
5. Folosind `JFileChooser`, permiteți utilizatorului să aleagă un fișier text din care se încarcă lista de conturi (reutilizând codul de parsare din Cap. 17 sau 18).

## Introducere

Am folosit deja `@Override` (Subsecțiunea 6.1.3), `@FunctionalInterface` (Subsecțiunea 7.2.2) și `@SuppressWarnings` (Subsecțiunea 18.2.1), fără să explicăm riguros mecanismul din spatele lor. Acest capitol formalizează *adnotările* — metadata atașate codului, procesabile de compilator, unelte sau la execuție — și introduce *Java Platform Module System* (JPMS), mecanismul modern de organizare a codului la scară mai mare decât un singur pachet.

## 24.1 Adnotări: concept și adnotări predefinite

### 24.1.1 Conceptul de adnotare

#### Definiție — Adnotare

O *adnotare* este o formă de metadata atașată unui element de cod (clasă, metodă, câmp, parametru) — nu modifică direct comportamentul programului, dar poate fi citită de compilator, de instrumente externe, sau la execuție, prin reflecție (Secțiunea 8.3).

```
1 @Override
2 public String toString() {
3     return "ContBancar[titular=" + titular + ", sold=" + sold
4         + "];";
5 }
```

**Listing 24.1:** Sintaxa de bază: o adnotare precede elementul pe care îl descrie

#### Notă

Distincția esențială față de un comentariu obișnuit (Subsecțiunea 2.1.1): un comentariu este ignorat complet de compilator, în timp ce o adnotare face parte din structura formală a programului, verificabilă și procesabilă programatic. `@Override` nu „face” nimic la execuție — valoarea ei este pur la **compilare**, semnalând o eroare dacă metoda adnotată nu suprascrie, de fapt, nimic din superclasă.

### 24.1.2 Adnotări predefinite

Tabela 24.1: Adnotările predefinite, din `java.lang`, deja folosite pe parcursul cursului.

| Adnotare                          | Rol                                                             |
|-----------------------------------|-----------------------------------------------------------------|
| <code>@Override</code>            | verifică, la compilare, că metoda suprascrie una din superclasă |
| <code>@Deprecated</code>          | marchează un element ca depășit, cu alternativa recomandată     |
| <code>@SuppressWarnings</code>    | suprimă un avertisment specific al compilatorului               |
| <code>@FunctionalInterface</code> | verifică exact o metodă abstractă (Subsecțiunea 7.2.2)          |

```
1 @Deprecated(since = "2.0", forRemoval = true)
2 public void metodaVeche() {
3     // ... implementare pastrata pentru compatibilitate,
```

**Listing 24.2:** `@Deprecated`, cu detalii suplimentare despre versiune și eliminare

```

4 | // dar nerecomandata ...
5 | }

```

### Notă

Parametrii `since/forRemoval` ai `@Deprecated` (din Java 9) oferă context suplimentar față de simpla marcare — `forRemoval = true` avertizează explicit că elementul va fi **eliminat** într-o versiune viitoare, nu doar înlocuit cu o alternativă preferată. Codul care apelează o metodă `@Deprecated` primește un avertisment de compilare, nu o eroare — rămâne funcțional, dar semnalează o datorie tehnică de rezolvat.

## 24.2 Definirea adnotărilor proprii; procesare; adnotări la runtime

### 24.2.1 Definirea unei adnotări proprii

O adnotare proprie se declară cu `@interface`, configurată prin *meta-adnotări* care controlează unde poate fi aplicată și cât timp rămâne disponibilă:

Listing 24.3: O adnotare proprie, pentru marcarea metodelor care trebuie auditate

```

1 | @Retention(RetentionPolicy.RUNTIME) // disponibila la
2 |                                           // EXECUTIE, prin
3 |                                           // reflectie
4 | @Target(ElementType.METHOD) // aplicabila DOAR pe metode
5 | public @interface Auditabil {
6 |     String descriere() default "Operatiune bancara";
7 | }

```

Listing 24.4: Aplicarea adnotării proprii, pe metode ale `ContBancar`

```

1 | public class ContBancar {
2 |     @Auditabil(descriere = "Depunere de fonduri")
3 |     public void depune(double suma) {
4 |         sold = sold + suma;
5 |     }
6 |
7 |     @Auditabil(descriere = "Retragere de fonduri")
8 |     public void retrage(double suma) throws
9 |         FonduriInsuficienteException {
10 |         // ...
11 |     }
12 | }

```

### Notă

`@Retention(RetentionPolicy.RUNTIME)` este esențial dacă doriți să citiți adnotarea prin reflecție, ca în Subsecțiunea 24.2.3 — alternativ, `SOURCE` (disponibilă doar în codul sursă, eliminată la compilare) și `CLASS` (păstrată în bytecode, dar inaccesibilă la execuție), sunt folosite mai ales de instrumente de compilare, nu de cod care citește adnotări la runtime. `@Target` restrânge unde poate fi aplicată adnotarea — fără el, ar fi validă pe orice element (clasă, câmp, metodă, parametru).

### 24.2.2 Procesoare de adnotări (context)

Un *procesor de adnotări* rulează în timpul **compilării**, inspectând adnotările din codul sursă și, opțional, generând cod suplimentar sau semnalând

erori personalizate:

#### Notă

Scrierea unui procesor de adnotări propriu (implementând `javax.annotation.processing.Processor`, înregistrat prin `ServiceLoader`, Subsecțiunea 8.4.3) este un subiect avansat, dincolo de scopul acestui curs introductiv. Este însă util să recunoașteți **conceptul**: multe biblioteci populare (generatoare de cod boilerplate, cadre de validare) funcționează exact prin acest mecanism — citesc adnotările voastre la compilare și generează automat cod suplimentar, vizibil abia în artefactele compilate.

### 24.2.3 Adnotări la runtime, prin reflecție

Cu `RetentionPolicy.RUNTIME`, adnotarea proprie devine vizibilă prin reflecție (Secțiunea 8.3), permițând construirea unui mecanism simplu de auditare automată:

```
1 for (Method metoda : ContBancar.class.getDeclaredMethods()) {
2     if (metoda.isAnnotationPresent(Auditabil.class)) {
3         Auditabil adnotare = metoda.getAnnotation(Auditabil.
4             class);
5         System.out.println("Metoda auditabila: " + metoda.
6             getName()
7             + " - " + adnotare.descriere());
8     }
9 }
```

Listing 24.5: Detectarea metodelor adnotate, și invocarea lor cu logging automat

```
1 public Object invocaCuAudit(ContBancar cont, String
2     numeMetoda, Object... argumente)
3     throws Exception {
4     Method metoda = ContBancar.class.getMethod(numeMetoda,
5         double.class);
6
7     if (metoda.isAnnotationPresent(Auditabil.class)) {
8         Auditabil adnotare = metoda.getAnnotation(Auditabil.
9             class);
10        LOG.info("Se executa: " + adnotare.descriere());
11    }
12
13    return metoda.invoke(cont, argumente);
14    // invocare dinamica, din Sec. 8.4.1
15 }
```

Listing 24.6: Un „wrapper” simplu, care loghează automat orice apel către o metodă adnotată

#### Notă

Acest tipar — inspectarea adnotărilor prin reflecție, urmată de o acțiune condiționată de prezența lor — este exact mecanismul din spațiile multor cadre de lucru enterprise: logging automat, validare de parametri, gestionare de tranzacții (Subsecțiunea 21.5.1) sau securitate, toate aplicate „din exterior”, fără a modifica direct codul metodelor adnotate. Combinația reflecție + adnotări proprii este, de altfel, fundația conceptuală a injectiei de dependențe și a programării orientate pe aspecte, tehnici larg răspândite în ecosistemul Java enterprise, dincolo de scopul acestui curs.

## 24.3 Java Platform Module System: module, module-info.java

### 24.3.1 Conceptul de modul

Introdus în Java 9, JPMS adaugă un nivel de organizare **peste** pachete (Secțiunea 5.1): un *modul* este o colecție numită de pachete, cu dependențe și API public declarate **explicit**, nu doar implicit prin modificatorul `public`:

#### Definiție — Modul

Un modul este o unitate de cod auto-descriptivă, care declară explicit ce alte module îi sunt necesare (`requires`) și ce pachete ale sale sunt accesibile din exterior (`exports`). Pachetele **neexportate** rămân complet inaccesibile din afara modulului, indiferent dacă clasele lor sunt declarate `public`.

#### Notă

Această ultimă observație este esențială și surprinde adesea la prima întâlnire: `public` (Subsecțiunea 4.4.1) nu mai înseamnă, automat, „vizibil de oriunde” — înseamnă doar „vizibil din alte pachete **ale aceluiași modul**, sau din module care importă explicit pachetul, prin `exports`”. JPMS oferă astfel o încapsulare mai puternică decât cea disponibilă la nivel de pachet (Subsecțiunea 5.1.4), unde o clasă `public` era necondiționat accesibilă oricărui cod de pe class path.

### 24.3.2 module-info.java; requires/exports

Un modul se declară printr-un fișier special, `module-info.java`, plasat la rădăcina codului sursă:

**Listing 24.7:** Declararea modulului băncii, cu dependențe și export explicit

```

1 module ro.unitbv.mip.banking {
2     requires java.sql;           // avem nevoie de JDBC (Cap. 21)
3     requires java.desktop;      // avem nevoie de Swing
4                                 // (Cap. 22-23)
5
6     exports ro.unitbv.mip.banking; // acest pachet devine
7                                     // vizibil altor module
8     // ro.unitbv.mip.banking.intern (daca ar exista)
9     // ar RAMANE ascuns, neexportat
10 }
```

#### Notă

`requires java.sql` declară explicit dependența de modulul standard care conține JDBC — fără această declarație, codul care importă `java.sql.Connection` nu ar compila, chiar dacă acea clasă există în JDK, exact pentru a forța declararea completă și explicită a tuturor dependențelor unui modul. Numele modulului (`ro.unitbv.mip.banking` de mai sus) urmează, prin convenție, aceeași structură ca numele pachetului rădăcină al proiectului (Subsecțiunea 5.1.1), deși tehnic cele două sunt concepte independente.

```

1 javac -d bin --module-source-path src -m ro.unitbv.mip.
   banking
2 java --module-path bin -m ro.unitbv.mip.banking/ro.unitbv.mip
   .banking.ContBancar

```

**Listing 24.8:** Compilarea și rularea unei aplicații modulare

#### Notă

Observați `--module-path`, în locul lui `-cp/-classpath` (Subsecțiunea 5.1.3) folosit până acum — JVM-ul tratează diferit codul de pe *module path* (organizat în module explicite, cu verificări de vizibilitate stricte) față de codul de pe *class path* tradițional (fără nicio structură modulară, tratat ca un singur „modul necunoscut” uriaș).

## 24.4 Module automate vs. explicite; migrarea codului legacy

### 24.4.1 Module automate

Nu toate bibliotecile externe au fost actualizate cu propriul fișier `module-info.java`. Un JAR obișnuit, fără această declarație, plasat totuși pe *module path*, devine automat un *modul automat* — JPMS îi generează implicit un nume (derivat din numele fișierului JAR) și îi acordă acces la toate celelalte module, exportând, la rândul lui, toate pachetele sale:

#### Notă

Un modul automat este o punte de tranziție, nu o soluție permanentă — oferă compatibilitate imediată pentru biblioteci mai vechi, dar fără garanțiile stricte de încapsulare pe care JPMS le oferă modulelor explicite (orice pachet al unui modul automat rămâne, implicit, complet exportat). Numele generat automat pentru un modul automat poate să difere ușor între versiuni ale aceleiași biblioteci — pentru dependențe stabile pe termen lung, un modul explicit, cu `module-info.java` propriu, rămâne alegerea mai sigură.

### 24.4.2 Migrarea codului legacy

Codul scris înaintea JPMS, fără `module-info.java`, plasat pe *class path* tradițional, aparține *modulului necunoscut* (*unnamed module*):

#### Notă

Modulul necunoscut poate citi orice alt modul (compatibilitate maximă cu codul existent), dar niciun modul **explicit** nu poate citi, implicit, modulul necunoscut — o asimetrie intenționată, care protejează codul modern, bine structurat, de dependențe accidentale pe cod legacy nestructurat. Strategia tipică de migrare este **incrementală**: proiectul rămâne pe *class path* (fără `module-info.java` propriu) cât timp depinde de biblioteci încă nemodularizate, trecând la module explicite doar atunci când întregul lanț de dependențe o permite — rareori o decizie de tip „totul odată”, pentru un proiect existent, de dimensiune reală.

### Rezumat

- O adnotare este metadată procesabilă de compilator, instrumente sau la execuție (prin reflecție), nu doar un comentariu decorativ; `@Override`, `@Deprecated`, `@SuppressWarnings`, `@FunctionalInterface` sunt cele mai frecvente predefinite.
- O adnotare proprie (`@interface`) se configurează prin `@Retention` (cât timp rămâne disponibilă) și `@Target` (unde poate fi aplicată); `RetentionPolicy.RUNTIME` este necesară pentru citire prin reflecție.
- Reflecția (`isAnnotationPresent`, `getAnnotation`) combinată cu adnotări proprii stă la baza multor cadre de lucru (logging automat, validare, injecție de dependențe).
- JPMS organizează codul în module, cu dependențe (`requires`) și API public (`exports`) declarate explicit — public nu mai înseamnă automat vizibil din orice alt cod, ci doar din module care importă explicit pachetul.
- Module automate oferă compatibilitate temporară pentru biblioteci nemodularizate; modulul necunoscut (class path tradițional) poate citi orice, dar nu poate fi citit implicit de module explicite — migrarea este, de regulă, incrementală.

### Exerciții

1. Definiți o adnotare proprie `@Validat`, aplicabilă pe metode, cu un atribut regula (`String`), și aplicați-o pe cel puțin două metode ale clasei `ContBancar`.
2. Scrieți un mic program care, prin reflecție, listează toate metodele adnotate cu `@Validat` dintr-o clasă dată, afișând valoarea atributului regula pentru fiecare.
3. Organizați pachetul `ro.unitbv.mip.banking` ca modul explicit, cu propriul `module-info.java`, declarând corect dependențele necesare (`java.sql` dacă folosiți exercițiile Cap. 21, `java.desktop` dacă folosiți Cap. 22–23).
4. Încercați să accesați, dintr-un alt modul, o clasă dintr-un pachet **neexportat** al modulului vostru, și observați eroarea de compilare rezultată.
5. Explicați, în cuvinte proprii, de ce un modul automat **exportă** implicit toate pachetele sale, spre deosebire de un modul explicit, unde exportul e opțional și selectiv.

### Introducere

Am folosit deja `LocalDate` în Subsecțiunea 4.2.2, ca prim exemplu de clasă imutabilă din biblioteca standard, dar fără o tratare completă. Acest capitol acoperă riguros API-ul `java.time` (introdus în Java 8, înlocuind complet clasele `Date/Calendar` discutate doar pentru context istoric, în Secțiunea 25.4): puncte pe linia timpului, date și ore locale, fusuri orare, formatare și parsare.

## 25.1 Linia timpului; `Instant`; ajustori de date

### 25.1.1 `Instant`: linia timpului absolută

`Instant` reprezintă un punct pe linia timpului universală (UTC), independent de orice fus orar sau calendar — potrivit pentru marcarea temporală precisă a evenimentelor:

```
1 Instant acum = Instant.now();
2 Instant maiTarziu = acum.plus(Duration.ofMinutes(30));
3
4 Duration durata = Duration.between(acum, maiTarziu);
5 System.out.println(durata.toMinutes()); // 30
```

Listing 25.1: `Instant`, pentru marcarea temporală a unei tranzacții

### Notă

`Instant` este util mai ales pentru *marcarea temporală* a evenimentelor (de exemplu, momentul exact al unei tranzacții bancare) sau pentru măsurarea intervalelor scurse — **nu** pentru afișarea către utilizator, unde o dată/oră locală, sensibilă la fusul orar, este aproape întotdeauna ceea ce se dorește de fapt. `Duration` reprezintă o cantitate de timp (ore, minute, secunde), spre deosebire de `Period`, folosit pentru intervale calendaristice (ani, luni, zile), introdus în secțiunea următoare.

### 25.1.2 `LocalDate`, recapitulat; ajustori de date

Reamintim operațiile de bază, deja întâlnite în Subsecțiunea 4.2.2, apoi extindem cu *ajustori* — transformări predefinite, aplicabile unei date:

```
1 LocalDate dataDeschidere = LocalDate.of(2024, 3, 15);
2 LocalDate peste30Zile = dataDeschidere.plusDays(30);
3 Period varsta = Period.between(dataDeschidere, LocalDate.now());
4
5 import static java.time.temporal.TemporalAdjusters.*;
6
7 LocalDate azi = LocalDate.now();
8
9 LocalDate primaZiLuna = azi.with(firstDayOfMonth());
10 LocalDate ultimaZiLuna = azi.with(lastDayOfMonth());
11 LocalDate urmatoareaLuniZi = azi.with(next(DayOfWeek.MONDAY));
12
13 // urmatoarea zi de luni
```

Listing 25.2: Recapitulare rapidă: creare și aritmetică de bază pe `LocalDate`

Listing 25.3: Ajustori de dată, pentru calcule calendaristice frecvente

```

9
10 LocalDate dataDobandaUrmatoare = azi.with(TemporalAdjusters.
    firstDayOfNextMonth());

```

#### Notă

Ajustorii (din `java.time.temporal.TemporalAdjusters`) exprimă direct, declarativ, calcule care ar necesita altfel logică manuală de calendar („care e ultima zi a lunii curente, ținând cont de anii bisecți?”) — exact stilul deja preferat în acest curs pentru operații pe colecții (`Stream`, Capitolul 15) sau pe `Optional` (Secțiunea 15.4), aplicat acum calculului de date. Metoda `with(ajustor)` returnează, ca orice metodă a unei clase imutabile (Subsecțiunea 4.2.2), o nouă instanță, lăsând data originală neschimbată.

## 25.2 `LocalTime`; `ZonedDateTime`

### 25.2.1 `LocalTime`

`LocalTime` reprezintă o oră din zi, fără dată și fără fus orar — utilă, de exemplu, pentru programul de funcționare al băncii:

Listing 25.4: `LocalTime`, pentru verificarea programului de lucru

```

1 LocalTime oraDeschidere = LocalTime.of(9, 0);
2 LocalTime oraInchidere = LocalTime.of(17, 0);
3
4 LocalTime acum = LocalTime.now();
5 boolean bancaEsteDeschisa = !acum.isBefore(oraDeschidere) &&
    acum.isBefore(oraInchidere);

```

#### Notă

La fel ca `LocalDate`, `LocalTime` este complet independent de context geografic — `LocalTime.of(9, 0)` reprezintă ora 9:00 „undeva”, fără nicio legătură cu un fus orar anume. Pentru scenarii care implică efectiv fusuri orare diferite (de exemplu, o bancă cu sucursale în mai multe țări), `ZonedDateTime`, secțiunea următoare, este clasa potrivită.

### 25.2.2 `ZonedDateTime`; fusuri orare

`ZonedDateTime` combină o dată, o oră și un fus orar explicit — necesar ori de câte ori momentul exact, la nivel global, contează:

Listing 25.5: `ZonedDateTime`, pentru o tranzacție cu fus orar explicit

```

1 ZonedDateTime tranzactieBucuresti = ZonedDateTime.of(
2     LocalDate.of(2026, 9, 15), LocalTime.of(14, 30), ZoneId.
3     of("Europe/Bucharest"));
4
5 ZonedDateTime aceeașiTranzactieNewYork = tranzactieBucuresti.
6     withZoneSameInstant(
7     ZoneId.of("America/New_York"));
8
9 System.out.println(tranzactieBucuresti);
10 // 2026-09-15T14:30+03:00[Europe/Bucharest]
11
12 System.out.println(aceeașiTranzactieNewYork);
13 // 2026-09-15T07:30-04:00[America/New_York]

```

**Notă**

`withZoneSameInstant` păstrează **momentul absolut** (același punct pe linia timpului, Subsecțiunea 25.1.1), recalculând doar reprezentarea locală pentru noul fus orar — spre deosebire de `withZoneSameLocal`, care ar păstra aceleași cifre de dată/oră afișate, mutând efectiv momentul absolut reprezentat. Confuzia dintre aceste două operații este o sursă frecventă de erori subtile în aplicații cu utilizatori din fusuri orare diferite — alegeți explicit, conștient, care comportament e cel dorit în fiecare situație.

## 25.3 Formatare și parsare

### 25.3.1 `DateTimeFormatter`: formatare

Reprezentarea implicită (`toString()`) a claselor `java.time`, folosită în exemplele anterioare, urmează formatul standard ISO — rareori formatul dorit pentru afișare către utilizator:

```
1 LocalDate dataDeschidere = LocalDate.of(2024, 3, 15);
2
3 String iso = dataDeschidere.format(DateTimeFormatter.
4     ISO_LOCAL_DATE);
5     // "2024-03-15"
6
7 DateTimeFormatter tiparRomanesc = DateTimeFormatter.ofPattern
8     ("dd.MM.yyyy");
9
10 String formatat = dataDeschidere.format(tiparRomanesc);
11     // "15.03.2024"
12
13 DateTimeFormatter tiparLocalizat = DateTimeFormatter
14     .ofLocalizedDate(FormatStyle.LONG)
15     .withLocale(new Locale("ro", "RO"));
16
17 String localizat = dataDeschidere.format(tiparLocalizat);
18     // "15 martie 2024"
```

**Listing 25.6:** Formatare cu tipare predefinite și personalizate

**Notă**

`ofPattern` folosește litere cu semnificație fixă (d = zi, M = lună, y = an, H = oră în format 24h etc.) — numărul de repetiții controlează formatul exact (M = 3, MM = 03, MMM = Mar, MMMM = Martie). Pentru text prietenos, dependent de limbă și regiune, `ofLocalizedDate` cu un `Locale` explicit produce automat formatul convențional al acelei zone, fără a construi manual tiparul — discutăm `Locale` riguros în capitolul următor, dedicat internaționalizării.

### 25.3.2 Parsare

Operația inversă — reconstituirea unei date dintr-un text — folosește același `DateTimeFormatter`:

```
1 DateTimeFormatter tiparRomanesc = DateTimeFormatter.ofPattern
2     ("dd.MM.yyyy");
3
4 try {
5     LocalDate data = LocalDate.parse("15.03.2024",
6         tiparRomanesc);
```

**Listing 25.7:** Parsarea unei date, dintr-un text cu format cunoscut

```

5 } catch (DateTimeParseException e) {
6     System.out.println("Format de data invalid: " + e.
7         getMessage());
8 }

```

#### Notă

`DateTimeParseException` este o excepție **neverificată** (Subsecțiunea 9.1.2), la fel ca `NumberFormatException` întâlnită deja implicit la `Double.parseDouble` — textul de intrare, mai ales dacă provine de la utilizator, trebuie tratat ca potențial invalid, exact principiul discutat pentru `Scanner` în Subsecțiunea 3.3.1. Tiparul de parsare trebuie să corespundă **exact** formatului textului — `LocalDate.parse("15.03.2024", DateTimeFormatter.ISO_LOCAL_DATE)` ar eșua, fiindcă formatul ISO așteaptă cifre separate prin liniuțe, nu prin puncte.

## 25.4 Interoperare cu API-ul legacy (Date, Calendar)

Înainte Java 8, singurele clase disponibile pentru dată/timp erau `java.util.Date` și `java.util.Calendar` — ambele considerate azi exemple clasice de design API deficitar: **mutabile** (spre deosebire de toate clasele `java.time`, imutabile, Subsecțiunea 4.2.2), cu o numerotare a lunilor pornind de la 0 (o sursă notorie de erori off-by-one), și fără suport corect pentru fusuri orare.

**Listing 25.8:** Conversie între `Instant` (modern) și `Date` (legacy)

```

1 Instant acum = Instant.now();
2
3 Date dataVeche = Date.from(acum);
4     // java.time -> API legacy
5 Instant inapoi = dataVeche.toInstant();
6     // API legacy -> java.time

```

#### Notă

Aceste metode de conversie (`Date.from/toInstant`) există exclusiv pentru **interoperare** cu biblioteci mai vechi, care încă așteaptă explicit `java.util.Date` în semnăturile lor de metode — nu ca alternativă viabilă la API-ul modern. Pentru cod nou, folosiți întotdeauna `java.time`; converțiți către/dinspre API-ul legacy doar la **granița** cu o bibliotecă externă care nu a fost încă actualizată, izolând acest cod de interoperare cât mai aproape posibil de acea graniță.

#### Rezumat

- `Instant` reprezintă un punct absolut pe linia timpului (UTC), potrivit pentru marcarea temporală a evenimentelor, nu pentru afișare directă către utilizator.
- `LocalDate/LocalTime` sunt independente de fus orar; ajustorii (`TemporalAdjusters`) exprimă declarativ calcule calendaristice frecvente.
- `ZonedDateTime` combină dată, oră și fus orar explicit; `withZoneSameInstant` păstrează momentul absolut, spre deosebire de `withZoneSameLocal`.
- `DateTimeFormatter` formatează și parsează date/ore, cu ti-

pare predefinite, personalizate (`ofPattern`) sau localizate (`ofLocalizedDate`); `DateTimeParseException` e neverificată, la fel ca alte excepții de parsare din bibliotecă.

- `java.util.Date/Calendar` sunt API-uri legacy, mutabile și deficitar proiectate; metodele de conversie (`Date.from/toInstant`) servesc exclusiv interoperării cu cod existent, nu ca alegere pentru cod nou.

### Exerciții

1. Adăugați un câmp `dataDeschidere` (`LocalDate`) la clasa `ContBancar`, inițializat automat cu `LocalDate.now()` la construcție, și afișați-l formatat ca `dd.MM.yyyy`.
2. Scrieți o metodă care calculează data următoarei plăți de dobândă, folosind `TemporalAdjusters.firstDayOfNextMonth()`, pornind de la data curentă.
3. Scrieți o metodă care verifică dacă un moment dat (`ZonedDateTime`) se încadrează în programul de lucru al băncii (`LocalTime`, Subsecțiunea 25.2.1), ținând cont corect de fusul orar al sucursalei.
4. Scrieți o metodă care primește o dată ca `String`, în formatul `"dd/MM/yyyy"`, o parsează, și tratează explicit cazul unui format invalid, afișând un mesaj prietenos.
5. Explicați, în cuvinte proprii, de ce mutabilitatea claselor `Date/Calendar` era considerată o problemă serioasă de design, revenind la discuția despre imutabilitate din Subsecțiunea 3.1.3 și Subsecțiunea 4.4.3.



## Introducere

Ultimul capitol al acestui curs adaugă o dimensiune pe care am ignorat-o până acum, din motive didactice: aplicația noastră bancară a presupus implicit un singur format de dată, o singură convenție numerică, o singură limbă. O aplicație reală, folosită de utilizatori din regiuni diferite, trebuie să se adapteze — fără a rescrie logica de business, doar prin schimbarea unui singur parametru, `Locale`. Vom relua, pe rând, formatarea numerelor, a datelor (deja atinsă parțial în Secțiunea 25.3), a mesajelor, și, în final, localizarea completă a textului unei aplicații, prin *resource bundles*.

## 26.1 Locale

### 26.1.1 Specificarea unui `Locale`

Un `Locale` identifică o combinație de limbă și, opțional, regiune/țară — parametrul central al oricărei operații sensibile la localizare:

```
1 Locale romania = Locale.of("ro", "RO");
2 // limba romana, Romania (Java 19+)
3 Locale franta = Locale.FRANCE;
4 // constanta predefinita, pentru locale-uri comune
5 Locale dinTag = Locale.forLanguageTag("en-US");
6 // din notatia standard BCP 47
```

**Listing 26.1:** Crearea unui `Locale`, în diverse moduri

## Notă

`Locale.of(limba, tara)` (introdus în Java 19) este forma modernă recomandată, înlocuind constructorul public `new Locale(...)`, considerat azi depășit (Subsecțiunea 24.1.2) — deși constructorul rămâne funcțional pentru compatibilitate, preferați metoda factory. Codul limbii urmează standardul ISO 639 ("ro", "en", "fr"); codul țării, ISO 3166 ("RO", "US", "FR") — aceeași limbă poate avea convenții diferite în regiuni diferite (engleza din SUA versus cea din Regatul Unit, de exemplu).

### 26.1.2 Locale implicit; denumiri afișate

```
1 Locale implicit = Locale.getDefault();
2 // configurat de sistemul de operare/JVM
3 System.out.println(implicit.getDisplayName());
4
5 Locale romania = Locale.of("ro", "RO");
6 System.out.println(romania.getDisplayLanguage(Locale.of("en"))
7 // "Romanian", afisat in engleza
8 System.out.println(romania.getDisplayLanguage(romania));
9 // "română", afisat in romana
```

**Listing 26.2:** `Locale`-ul implicit al mașinii virtuale, și denumiri afișate în diverse limbi

**Notă**

`getDisplayLanguage` (și metodele înrudite, `getDisplayCountry`) acceptă, opțional, un al doilea `Locale` — cel în care se dorește afișat numele primului. Această distincție subtilă contează: numele limbii române, afișat pentru un utilizator vorbitor de engleză, ar trebui să fie "Romanian", nu "română". `Locale.getDefault()` reflectă configurarea sistemului — nu vă bazați pe el fără verificare explicită într-o aplicație care trebuie să suporte utilizatori din regiuni diferite simultan (de exemplu, un server web, unde fiecare cerere poate proveni dintr-o altă regiune).

## 26.2 Formate numerice și monetare

### 26.2.1 `NumberFormat`, `DecimalFormat`

Convenția de scriere a numerelor diferă semnificativ între regiuni — cea mai frecventă diferență, separatorul zecimal ( . versus , ):

**Listing 26.3:** Formatarea aceluiași număr, în convenții diferite

```
1 double sold = 1234.5;
2
3 NumberFormat formatRomania = NumberFormat.getInstance(Locale.
  of("ro", "RO"));
4 System.out.println(formatRomania.format(sold));
5 // "1.234,5" - punct pentru mii, virgula pentru zecimale
6
7 NumberFormat formatSUA = NumberFormat.getInstance(Locale.US);
8 System.out.println(formatSUA.format(sold));
9 // "1,234.5" - exact invers!
```

**Notă**

Aceasta este exact aceeași valoare numerică (1234.5), formatată diferit — o eroare frecventă și costisitoare este **codificarea manuală** a formatului (de exemplu, `String.format("%.2f", sold)`, care presupune implicit convenția unei singure regiuni) în loc de folosirea unui `NumberFormat` sensibil la `Locale`. Pentru control mai fin asupra tiparului (număr de zecimale, grupare), `DecimalFormat` (o subclasă concretă a lui `NumberFormat`) acceptă un tipar explicit, similar sintaxei `DateTimeFormatter.ofPattern` din Subsecțiunea 25.3.1.

**Listing 26.4:** `DecimalFormat`, cu tipar explicit

```
1 DecimalFormat formatPersonalizat = new DecimalFormat("
  #,##0.00");
2 System.out.println(formatPersonalizat.format(sold));
3 // "1,234.50" - simboluri implicite platformei
```

### 26.2.2 Formatarea monedelor

**Listing 26.5:** Formatarea unui sold ca sumă monetară, în diverse monede regionale

```
1 double sold = 1234.5;
2
3 NumberFormat formatRON = NumberFormat.getCurrencyInstance(
  Locale.of("ro", "RO"));
4 System.out.println(formatRON.format(sold));
5 // "1.234,50 RON" (sau simbolul corespunzător)
6
```

```

7 | NumberFormat formatUSD = NumberFormat.getCurrencyInstance(
    |     Locale.US);
8 | System.out.println(formatUSD.format(sold));
9 |     // "$1,234.50"

```

#### Notă

`getCurrencyInstance` deduce automat moneda implicită asociată regiunii din `Locale` (leul românesc pentru `Locale.of("ro", "RO")`, dolarul american pentru `Locale.US`) — pentru afișarea unei sume într-o monedă diferită de cea implicită regiunii (de exemplu, solduri în euro, afișate unui utilizator din România), folosiți explicit `setCurrency(Currency.getInstance("EUR"))` pe formatul obținut, suprascriind moneda implicită dedusă din `Locale`.

## 26.3 Formatare dată/timp internaționalizată; collation

### 26.3.1 Formatare dată/timp internaționalizată

Am folosit deja, pe scurt, `ofLocalizedDate` în Subsecțiunea 25.3.1. Extindem acum la date și ore, cu control explicit al nivelului de detaliu:

```

1 | LocalDateTime tranzactie = LocalDateTime.of(2026, 9, 15, 14,
    |     30);
2 |
3 | DateTimeFormatter formatRomanesc = DateTimeFormatter
    |     .ofLocalizedDateTime(FormatStyle.MEDIUM)
    |     .withLocale(Locale.of("ro", "RO"));
4 |
5 | System.out.println(tranzactie.format(formatRomanesc));
6 |     // "15 sept. 2026, 14:30:00"
7 |
8 |
9 | DateTimeFormatter formatAmerican = DateTimeFormatter
    |     .ofLocalizedDateTime(FormatStyle.MEDIUM)
    |     .withLocale(Locale.US);
10 |
11 | System.out.println(tranzactie.format(formatAmerican));
12 |     // "Sep 15, 2026, 2:30:00 PM"
13 |

```

**Listing 26.6:** Formatare localizată completă, dată și oră, cu grad de detaliu configurabil

#### Notă

Observați diferența dincolo de simpla traducere a numelui lunii: convenția americană folosește formatul de 12 ore (AM/PM), cea românească, formatul de 24 ore — o diferență pe care `FormatStyle` + `Locale` o gestionează automat, fără nicio configurare manuală suplimentară. `FormatStyle` oferă patru niveluri (SHORT, MEDIUM, LONG, FULL), fiecare cu o cantitate diferită de detaliu, toate respectând convențiile regiunii specificate prin `Locale`.

### 26.3.2 Collation

Sortarea alfabetică „naturală” a lui `String` (`compareTo`, folosită implicit de `TreeSet/Collections.sort` din Subsecțiunea 11.3.2) compară codurile Unicode ale caracterelor — corect pentru text englez simplu, dar adesea greșit pentru limbi cu diacritice:

**Listing 26.7:** Sortare „naturală” (incorectă) versus sortare cu Collator (corectă)

```

1 List<String> titulari = new ArrayList<>(List.of("Ana", "
   Ștefan", "Andrei", "Șerban"));
2
3 Collections.sort(titulari);
4 // sorteaza dupa codul Unicode - 'Ș' ajunge dupa 'z',
5 // GRESIT alfabetic!
6
7 Collator collatorRomanesc = Collator.getInstance(Locale.of("
   ro", "RO"));
8 titulari.sort(collatorRomanesc::compare);
9 // sorteaza CORECT, conform alfabetului romanesc

```

**Notă**

Codul Unicode al literei Ș este numeric mai mare decât cel al literelor obișnuite a–z, deci o comparație `compareTo` simplă ar plasa toate numele începând cu Ș după orice nume care începe cu literă neaccentuată, indiferent de ordinea alfabetică reală așteptată de un vorbitor de română. Collator rezolvă exact această problemă, comparând text conform regulilor lingvistice ale unui Locale specific, nu conform valorilor numerice brute ale caracterelor — esențial pentru orice sortare de text vizibilă utilizatorilor, dincolo de identificatori interni sau chei tehnice.

## 26.4 Formatare de mesaje; frontiere de text

### 26.4.1 MessageFormat; *choice formats*

MessageFormat construiește text din tipare cu substituenți poziționali — utilă pentru mesaje ale căror componente variază (sumă, titular, dată), fără concatenare manuală:

**Listing 26.8:** MessageFormat, cu substituenți poziționali

```

1 String tipar = "Tranzactie confirmata: {0} lei, cont {1}, la
   data {2}";
2
3 String mesaj = MessageFormat.format(tipar, 150.0, "Ana
   Popescu", LocalDate.now());
4 // "Tranzactie confirmata: 150 lei, cont Ana Popescu, la
5 // data 2026-09-15"

```

Pentru pluralizare corectă — o problemă notorie de internaționalizare, fiindcă regulile diferă radical între limbi — ChoiceFormat selectează textul potrivit în funcție de o valoare numerică:

**Listing 26.9:** ChoiceFormat, integrat într-un MessageFormat, pentru pluralizare corectă

```

1 String tiparCuAlegere =
   "Aveti {0,choice,0#nicio tranzactie|1#o tranzactie|2#{0}
   tranzactii}";
2
3
4 System.out.println(MessageFormat.format(tiparCuAlegere, 0));
5 // "Aveti nicio tranzactie"
6 System.out.println(MessageFormat.format(tiparCuAlegere, 1));
7 // "Aveti o tranzactie"
8 System.out.println(MessageFormat.format(tiparCuAlegere, 5));
9 // "Aveti 5 tranzactii"

```

**Notă**

Sintaxa `{0,choice,0#...|1#...|2#...}` definește intervale (0, 1, 2 și peste) și textul corespunzător fiecăruia, separate prin `|` — fără acest mecanism, codul ar necesita o cascadă manuală de `if/else if` pentru fiecare formă de plural, repetată pentru fiecare limbă suportată. Chiar și așa, regulile de plural ale limbii române (cu o formă specială pentru numere între 1 și 19, alta pentru restul) sunt mai complexe decât simpla distincție „singular/plural” a englezei — pentru localizare completă și corectă, resource bundles (Secțiunea 26.5) combinate cu biblioteci specializate de pluralizare rămân, în practică, mai robuste decât `ChoiceFormat` singur.

### 26.4.2 Frontiere de text (context)

Determinarea corectă a „unde se termină un cuvânt” sau „unde se termină o propoziție” nu este universală — unele limbi (chineză, japoneză) nu folosesc spații între cuvinte. `BreakIterator` oferă o soluție sensibilă la Locale:

```
1 BreakIterator iteratorCuvinte = BreakIterator.getWordInstance
  (Locale.of("ro", "RO"));
2 iteratorCuvinte.setText("Ana Popescu a depus 150 lei.");
3
4 int inceput = iteratorCuvinte.first();
5 for (int sfarsit = iteratorCuvinte.next(); sfarsit !=
  BreakIterator.DONE;
6     inceput = sfarsit, sfarsit = iteratorCuvinte.next())
7 {
8     System.out.println(inceput + "-" + sfarsit);
9 }
```

Listing 26.10: `BreakIterator`, pentru identificarea corectă a limitelor de cuvinte

**Notă**

Pentru text din limbi cu spații între cuvinte (ca româna, folosită în exemplul de mai sus), diferența față de un simplu `split("`

`s+")` (Subsecțiunea 18.4.1) pare minoră — beneficiul real al `BreakIterator` apare la text multilingv sau la limbi fără separatori vizibili între cuvinte, unde o soluție bazată pe expresii regulate ar eșua complet. Menționăm această clasă mai ales pentru completitudine conceptuală, rar necesară direct în aplicații care operează într-o singură limbă cunoscută.

## 26.5 Resource bundles: localizarea aplicațiilor

Toate mecanismele de până acum (formatare numerică, dată/timp, mesaje) adaptează automat `date`, dar textul fix al interfeței — etichetele butoanelor din aplicația bancară a Cap. Capitolul 22–Capitolul 23, de exemplu — rămâne, până acum, scris direct în cod, într-o singură limbă. *Resource bundles* rezolvă exact această problemă finală.

```
1 buton.depune=Depune
2 buton.retrage=Retrage
3 titlu.fereastra=Banca - Cont {0}
```

Listing 26.11: Fișier de proprietăți pentru limba română (`mesaje_ro-RO.properties`)

**Listing 26.12:** Fișierul echivalent, pentru engleză (mesaje\_en\_US.properties)

```

1| buton.depune=Deposit
2| buton.retrage=Withdraw
3| titlu.fereastra=Bank - Account {0}

```

**Listing 26.13:** Încărcarea și folosirea unui ResourceBundle, în construcția interfeței

```

1| Locale localeCurent = Locale.of("ro", "RO");
2| ResourceBundle mesaje = ResourceBundle.getBundle("mesaje",
   |     localeCurent);
3|
4| JBUTTON butonDepunere = new JBUTTON(mesaje.getString("buton.
   |     depune"));
5| JBUTTON butonRetragere = new JBUTTON(mesaje.getString("buton.
   |     retrage"));
6|
7| String titluFormatat = MessageFormat.format(mesaje.getString(
   |     "titlu.fereastra"), cont.getTitular());
8| fereastra.setTitle(titluFormatat);

```

**Notă**

`ResourceBundle.getBundle("mesaje", localeCurent)` caută automat fișierul cel mai specific pentru Locale-ul dat, cu un mecanism de *fallback* ierarhic: mai întâi `mesaje_ro_RO.properties` (limbă și țară), apoi `mesaje_ro.properties` (doar limba, dacă varianta specifică țării lipsește), apoi `mesaje.properties` (fișierul implicit, fără sufix de locale) — primul găsit este folosit, fără nicio configurare suplimentară din partea codului. Adăugarea suportului pentru o limbă nouă (de exemplu, franceza) necesită doar un fișier nou, `mesaje_fr_FR.properties`, fără nicio modificare de cod Java.

**Notă**

O alternativă la fișierele `.properties` (text simplu) este `ListResourceBundle` — o subclasă Java, cu datele returnate direct din cod, utilă atunci când valorile necesită logică (nu doar text static) sau când preferați siguranța de tip a verificării la compilare, în locul textului simplu, verificat abia la execuție. Fișierele `.properties` rămân însă alegerea implicită preferată, mai ales pentru echipe unde traducerea textului este realizată de persoane fără cunoștințe de programare — un fișier text simplu, cu perechi cheie-valoare, este mult mai accesibil decât cod Java compilabil.

**Rezumat**

- `Locale` identifică o combinație de limbă și regiune; `Locale.of(...)` este forma modernă recomandată; denumirile afișate pot fi cerute în orice altă limbă, prin al doilea parametru al metodelor `getDisplayX`.
- `NumberFormat/DecimalFormat` adaptează automat separatorii numerici la convenția regională; `getCurrencyInstance` formează sume monetare cu moneda implicită regiunii, sau una explicită.
- `DateTimeFormatter.ofLocalizedDateTime` combină `FormatStyle` cu `Locale` pentru formatare de dată/timp complet localizată; `Collator` sortează text corect, respectând regulile alfabetice ale unei limbi, spre deosebire de comparația brută pe coduri Unicode.

- ▶ `MessageFormat` construiește mesaje din tipare cu substituenți poziționali; `ChoiceFormat` tratează pluralizarea, deși pentru limbi cu reguli complexe rămâne adesea insuficient singur.
- ▶ `ResourceBundle` externalizează complet textul fix al unei aplicații în fișiere separate pe limbă, cu fallback ierarhic automat — mecanismul final, care completează localizarea completă a unei aplicații, dincolo de formatarea datelor variabile.

### Exerciții

1. Afișați soldul unui cont, formatat ca sumă monetară, simultan în trei convenții regionale diferite (România, SUA, Franța), și comparați rezultatele.
2. Construiți o listă de titulari cu nume conținând diacritice românești, apoi sortați-o atât cu `Collections.sort` obișnuit, cât și cu un `Collator` românesc, comparând vizual diferența de ordine rezultată.
3. Scrieți un mesaj de confirmare a tranzacției, folosind `MessageFormat` cu un `ChoiceFormat` imbricat, care tratează corect cazurile 0, 1 și mai multe tranzacții într-un extras de cont.
4. Creați două fișiere `.properties` (română și engleză) pentru textul interfeței aplicației bancare din Cap. 22–23, și modificați codul Swing existent să încarce etichetele din `ResourceBundle`, în loc de text fixat direct în cod.
5. Reflectați, în câteva propoziții: dintre toate capitolele acestui curs, care mecanisme discutate aici (`Locale`, formatare, `ResourceBundle`) ar fi cele mai relevante pentru o aplicație reală pe care ați dori să o construiți, și de ce.



# Bibliografie

Referințele sunt listate în ordinea citării.

- [1] Cay S. Horstmann, *Core Java, Volume I: Fundamentals*, a 14-a ed., Pearson Education, Inc., Hoboken, New Jersey: Addison-Wesley Professional, 2026 (cit. în pp. 19, 63, 137, 260).
- [2] Cay S. Horstmann, *Core Java, Volume II: Advanced Features*, a 14-a ed., Pearson Education, Inc., Hoboken, New Jersey: Addison-Wesley Professional, 2026 (cit. în pp. 157, 189, 205, 260).
- [3] Benjamin J. Evans, Jason Clark și David Flanagan, *Java in a Nutshell: A Desktop Quick Reference*, a 8-a ed., Sebastopol, California: O'Reilly Media, 2023 (cit. în pp. 37, 260).



# Index alfabetic

abstract, 73  
Action, 220  
ActionListener, 220  
adnotare, 233  
array-uri, 33  
ArrayList, 71  
assert, 102  
AtomicLong, 148  
autoboxing, 71  
AutoCloseable, 101  
AWT, 217

batch update, 212  
BigDecimal, 41  
BigInteger, 41  
BitSet, 135  
BlockingQueue, 149  
blocuri de inițializare, 48  
BorderLayout, 222  
bound (limită de tip), 111  
BreakIterator, 249  
BufferedInputStream, 174  
bytecode, 20

Calendar, 242  
Callable, 139  
callback, 79  
capturare wildcard, 115  
captură de variabile, 83  
cast (conversie de tip), 30  
catch, 99  
chaining de excepții, 100  
cheie autogenerată, 209  
clase, 43  
    anatomie, 45  
    publice, 27  
clase imbricate, 84  
clase wrapper, 71  
Class, 92  
classpath, 58  
clasă anonimă, 86  
clasă internă, 85  
clasă locală, 86  
Cloneable, 80  
codificare de caractere, 175  
collation, 247  
Collator, 247  
collect, 165  
Collection, 119  
Collections, 124  
Collectors, 165  
Color, 219  
commit, 210  
Comparator, 79  
CompletableFuture, 150  
computeIfAbsent, 129  
ConcurrentHashMap, 149  
condiție de cursă, 145  
Connection, 206  
connection pooling, 212  
constructor, 46  
constructor canonic, 53  
CopyOnWriteArrayList, 150  
căutare binară  
    pe List, 125

DataOutputStream, 175  
DataSource, 212  
java.util.Date, 242  
DateTimeFormatter, 241  
deadlock, 146  
debugging, 105  
DecimalFormat, 246  
default (interfață), 78  
Deque, 123  
design orientat pe obiecte, 43  
DocumentBuilder, 189  
DOM, 189  
downstream collector, 167  
driver JDBC, 205  
DriverManager, 206  
DTD, 190

egalitate  
    referință vs. conținut, 38  
element XML, 189  
entrySet, 130  
enum, 29  
EnumMap, 132  
excepție, 97  
    neverificată (unchecked), 97  
    verificată (checked), 97  
ExecutorService, 140  
exhaustivitate, 91  
expresie regulată, 184  
extends, 65

Field, 93  
file locking, 181  
FileLock, 181  
Files, 177  
filter, 158  
final, 50  
finally, 99

flag (expresii regulate), 186  
flatMap, 158  
Font, 219  
forEach, 159  
ForkJoinPool, 142  
framework de colecții, 119  
Future, 139

gardă (pattern matching), 91  
Gatherer, 168  
Graphics, 218  
groupBy, 166  
grup de captură, 185

Handler, 104  
hashCode, 70  
HashMap, 129  
HashSet, 121  
Hashtable, 135  
HttpClient, 200  
HttpRequest, 200  
HttpServer, 201

IdentityHashMap, 132  
Image, 219  
implements, 77  
import, 58  
InetAddress, 197  
injecție SQL, 207  
InputStream, 173  
instanceof, 67  
Instant, 239  
interfață, 77  
interfață funcțională, 81  
interrupt, 138  
IntStream, 169  
invariantă (generice), 113  
invokeAll, 141  
isAnnotationPresent, 235  
Iterator, 119

JAR, 59  
Java  
    istoric, 19  
java.util.function, 82  
Javadoc, 61  
    tag-uri, 61  
JavaMail, 202  
JCheckBox, 226  
JComboBox, 226  
JDBC, 205  
JDialog, 229  
JDK, 21  
    versiuni LTS, 20  
JFileChooser, 230  
JFrame, 217  
JMenu, 227  
JMenuBar, 227

JOptionPane, 228  
JPasswordField, 225  
JPMS, 236  
JRadioButton, 226  
JScrollPane, 225  
JShell, 24  
JSlider, 227  
JTable, 230  
JTextArea, 225  
JTextField, 225  
JToolBar, 228  
JTree, 231  
JVM, 20

lambda, 80  
layout manager, 222  
LinkedHashMap, 131  
LinkedList, 121  
List.of, 133  
listener, 220  
LocalDate, 45, 239  
Locale  
    implicit, 245  
Locale, 245  
LocalTime, 240  
Lock, 147  
Logger, 103  
logging, 103

manifest (JAR), 60  
Map, 129  
map, 158  
MappedByteBuffer, 181  
memory-mapped file, 181  
merge, 129  
MessageFormat, 248  
Method, 93  
metodă factory, 51  
migrare (JPMS), 237  
Model-View-Controller, 221  
modificatori de acces, 49  
modul (JPMS), 236  
modul automat, 237  
module-info.java, 236  
monedă (formatare), 246  
monitor, 145  
MouseListener, 221  
moștenire, 65  
MVC, 221

namespace XML, 192  
NIO.2, 177  
null  
    în switch, 92  
NumberFormat, 246

obiecte, 43  
Object, 69

- operații bulk, 125
- Optional, 160
- OutputStream, 173
- overloading, 48
  
- pachete, 57
- paintComponent, 218
- ParameterizedType, 116
- parametru de tip, 109
- parsare (date/timp), 241
- partitioningBy, 166
- Path, 177
- Pattern, 184
- pattern matching, 89
- platform thread, 139
- prefață, 15
- PreparedStatement, 207
- PriorityQueue, 123
- procesor de adnotări, 234
- ProcessBuilder, 152
- programare declarativă, 157
- programare generică, 109
- protected, 68
- Proxy, 95
- publicare sigură, 148
  
- Queue, 123
  
- race condition, 145
- RandomAccessFile, 176
- Reader, 174
- record, 53
- record pattern, 89
- reduce, 160
- ReentrantLock, 147
- referință (variabilă obiect), 44
- referință la constructor, 83
- referință la metodă, 82
- reflecție, 92
- replaceAll, 186
- ResourceBundle, 249
- ResultSet, 207
- ResultSetMetaData, 209
- RowSet, 210
- Runnable, 137
  
- Savepoint, 211
- SAX, 193
- Scanner, 39
- ScopedValue, 142
- sealed, 75
- sendAsync, 201
- Serializable, 182
- serializare, 182
- serialVersionUID, 184
- ServerSocket, 198
- ServiceLoader, 95
- Socket, 197
- SocketTimeoutException, 197
- split, 186
- SQL, 205
- SQLException, 208
- SSL, 199
- Stack, 135
- stack trace, 102
- stare thread, 137
- Statement, 207
- StAX, 193
- Stream, 157
- stream, 157
- stream de filtrare, 174
- stream paralel, 169
- String, 37
- subList, 134
- super, 66
- Swing, 217
- switch
  - pattern matching, 90
- synchronized, 145
  
- TableModel, 230
- TemporalAdjuster, 239
- text blocks, 39
- thenApply, 151
- thenCombine, 151
- this, 47
- thread, 137
- ThreadLocal, 141
- throw, 98
- Throwable, 97
- throws, 98
- tip atomic, 148
- tipuri primitive, 28
- TLS, 199
- toMap, 165
- tooltip, 228
- toString, 70
- transient, 183
- tranzacție, 210
- TreeMap, 131
- TreeSet, 122
- try, 99
- try-with-resources, 101
- type erasure, 112
  
- Unicode, 29
- unmodifiableList, 133
- URI, 200
- URL, 200
- URL JDBC, 206
- UTF-8, 175
  
- varargs, 72
- virtual thread, 139
- volatile, 147

WeakHashMap, 132  
wildcard, 113  
Writer, 174

XML, 189  
XML Schema, 191  
XPath, 192

XSD, 191  
XSLT, 194

ZipOutputStream, 177  
ZonedDateTime, 240  
ZoneId, 240