

## Expresii și instrucțiuni de control în Java

### 1. Scopul lucrării

Obiectivele acestei sesiuni de laborator sunt:

- Înțelegerea caracteristicilor specifice ale operatorilor Java și deprinderea utilizării lor corecte
- Utilizării instrucțiunilor de control al fluxului execuției în Java
- Acumularea de experiență nemijlocită în lucrul cu variabile și expresii prin dezvoltarea și executarea de programe de dimensiuni mici

### 2. Expresii

Expresiile constituie principalul mijloc pentru a crea valori. Expresiile sunt create prin combinarea literalilor (constantelor), a variabilelor, invocarilor de metode și utilizând operatori. Pentru a obține ordinea de evaluare corectă se vor folosi paranteze.

#### 2.1. Operanzi

##### Tipuri

Fiecare variabilă are asociat un *tip*. Sunt prevăzute două categorii de tipuri:

- **Tipuri primitive** : byte, short, char, int, long, float, double, boolean.
- **Tipuri obiect** : Tipurile String și tablou sunt tipuri predefinite, dar fiecare clasă care este definită creează un nou tip de obiect.

##### Constante literale

Oferă un mod de a scrie valori aparținând diferitelor tipuri (3, 3.0, true, 'a', "abc").

##### Variabile

Fiecare variabilă trebuie declarată ca și aparținând unui tip. Variabilele se pot încadra în trei categorii de bază diferite:

- Variabile locale declarate în metode .
- Variabile instanță (deseori numite câmpuri) declarate în obiecte.
- Variabile clasă (static) declarate în clase.

#### 2.2. Operatori

Operatorii sunt utilizați pentru a exprima diferite formule elementare de calcul sau prelucrare a informației și sunt folosiți alături de literalii, variabile, apeluri de metode și alte expresii. Operatorii se pot clasifica în următoarele categorii.

- Operatori aritmetici(+, -, \*, /, %, ++, --)
- Operatori relaționali (<, <=, ==, >=, >, !=)
- Operatori booleani (&&, ||, !, &, ^, |, &)
- Operatori la nivel de bit (&, |, ^, ~, <<, >>, >>>)
- Operatorul de concatenare a String-urilor (+)
- Operatori speciali (instanceof, ?:)
- Operatori de atribuire (=, +=, -=, \*=, ...)

În general, ordinea de evaluare a expresiilor este de la stânga la dreapta; excepție de la această regulă o fac operatorii de atribuire. Ordinea de evaluare poate fi schimbată utilizând paranteze (vezi tabelul de precedență al operatorilor din secțiunea 2.3).

### 2.2.1 Operatori relaționali

Toți operatorii standard funcționează pentru *valori de tip primitiv* (`int`, `double`, `char`, ...). Operatorii `==` (test de egalitate) și `!=` (test de inegalitate) pot fi utilizați și pentru a compara referințe la obiecte.

Rezultatul oricărei comparații este o valoare de tip *boolean* (`true` sau `false`).

operator	Semnificație
<	mai mic
<=	mai mic sau egal
==	egalitate
>=	mai mare sau egal
>	mai mare
!=	inegalitate (diferit de)

#### Erori comune

`0 < x < 100`

Operatorii de comparare nu pot exprima inegalități duble. Deși matematic formula este corectă, sintaxa Java nu permite scrierea ei. Exprimarea corectă în limbajul Java este scrierea a două comparații:

`0 < x && x < 100`

**= în loc de ==**

Utilizarea operatorului de atribuire în loc de test de egalitate va genera o eroare de compilare ușor de corectat.

**== pentru valori reprezentate în virgulă mobilă**

Deoarece numerele în virgulă mobilă nu oferă totdeauna reprezentarea exactă a valorilor va trebui să utilizați `>=` sau `<=` în loc de `==`. Spre exemplu, deoarece numărul zecimal 0.1 nu poate fi reprezentat exact în binar, expresia `(0.1 + 0.1 + 0.1)` **nu** va fi egală cu 0.3!

#### Recomandări pentru programatorii C/C++

Operatorii de comparare ai limbajului Java sunt la prima vedere identici cu cei din C/C++. Diferența constă în faptul că tipul rezultatului operației de comparare este boolean. Astfel, eroarea frecventă în limbajul C: utilizarea operatorului `=` în loc de `==` este aproape complet eliminată. Limbajul Java nu permite supraîncărcarea operatorului, aspect ce poate fi ignorat de unii programatori obișnuiți cu filozofia C++.

**==, .equals(), și compareTo()**

**Testul de egalitate : o variantă pentru tipuri primitive, patru variante pentru obiecte**

Comparație	Primitive	Obiecte
<code>a == b, a != b</code>	Valori egale	Referință spre același obiect
<code>a.equals(b)</code>	Nu se aplică	Compară valorile, dacă sunt definite pentru această clasă, dacă sunt definite pentru clasele respective, se comportă ca și pentru clasele predefinite Java. Dacă nu este definită pentru o clasă (utilizator) se comportă ca și operatorul <code>==</code> .
<code>a.compareTo(b)</code>	Nu se aplică	Compară valori. Clasa trebuie să aibă implementată interfața <i>Comparable</i> <T>. Toate clasele Java având relația de ordine naturală au implementată această interfață ( <i>String</i> , <i>Double</i> , <i>BigInteger</i> , ...) Face excepție clasa <i>BigDecimal</i> care poate produce rezultate imprevizibile. Obiectele <i>Comparable</i> pot fi utilizate de metodele <b>Collections</b> <i>sort()</i> și de structurile de date sortate implicit (d.e. <i>TreeSet</i> , <i>TreeMap</i> ).
<code>compareTo(a, b)</code>	Nu se aplică	Compară valori. Este disponibilă numai dacă este implementată interfața <i>Comparator</i> <T> care nu este specifică pentru clasele Java. De obicei se folosește pentru a defini un obiect de comparare care va fi transmis metodelor <b>Collections</b> <i>sort()</i> sau structurilor de date ordonate.

### Compararea referințelor la obiecte folosind operatorii == și !=

Cei doi operatori de comparare pentru referințe la obiecte sunt testul de egalitate (==) și testul de neegalitate (!=). Acești operatori compară două valori pentru a vedea dacă ele **referă același obiect**. Cu toate că este o operațiune foarte rapidă nu întotdeauna produce ceea ce așteptați. De obicei vreți să aflați dacă obiectele au aceeași *valoare*, și nu dacă două obiecte constituie o *referință* spre același obiect. Spre exemplu,

```
if (name == "Mickey Mouse")    // Aproape sigur o eroare
```

Aici, rezultatul este adevărat(true) dacă `name` este o referință către *același obiect* spre care face referință și "Mickey Mouse". În schimb, rezultatul va fi probabil false dacă String-ul din variabila `name` a fost citit din fișierul de intrare sau calculat (prin concatenare sau extragere subșir), chiar dacă `name` are într-adevăr exact aceleași caractere memorate în locația asociată.

Mai multe clase (d.e., `String`) definesc metoda `equals()` pentru a compara *valori* ale obiectelor.

### Compararea valorilor obiectelor utilizând metoda equals()

Se va folosi metoda `equals()` pentru a compara valori ale obiectelor. Metoda `equals()` furnizează o valoare de tip boolean. Exemplul de mai sus poate fi corectat scriind:

```
if (name.equals("Mickey Mouse")) // Compară valori, nu referințe
```

### Alte comparații – Interfața Comparable<T>

Metoda `equals` precum și operatorii `==` și `!=` sunt prevăzuți pentru testul de egalitate/inegalitate, dar nu oferă o metodă de a testa valori înrudite. Unele clase (d.e. `String` și alte clase având relația de ordine naturală) au implementat interfața `Comparable<T>`, care definește metoda `compareTo`. Va trebui să implementați `Comparable<T>` în clasele proprii dacă aveți nevoie să le utilizați în metodele `Collections.sort()` sau `Arrays.sort()`. Clasa `String` oferă și metode de comparație care nu fac diferență între literele mari și mici.

### Erori frecvente în utilizarea operatorului == în loc de equals() pentru obiecte

Dacă doriți să comparați obiecte, trebuie să știți că operatorul `==` se folosește pentru a afla dacă ele sunt *același obiect*, iar metoda `equals()` pentru a testa dacă ele sunt obiecte diferite, dar au *aceeași valoare*. O eroare produsă de această eventuală confuzie este greu de depistat.

### 2.2.2 Operatori la nivel de bit

Operatorii pe biți acționează asupra biților valorilor întregi (`int` și `long`). Dacă un operand este mai scurt decât un întreg, el este convertit la `int` înainte de a efectua operațiunea.

Este necesar a ști cum se reprezintă valorile întregi în sistemul binar. Spre exemplu numărul zecimal 3 este reprezentat 11 în baza 2 iar reprezentarea binară a lui 5 este 101. Numerele întregi negative sunt reprezentate utilizând codul *complementului față de 2*. Spre exemplu, -4 este reprezentat prin următoarea secvență de cifre binare : 1111 1111 1111 1111 1111 1111 1100.

Operator	Nume	Exemplu	Rezultat	Descriere
$a \& b$	și	$3 \& 5$	1	1 dacă ambii biți au valoarea 1.
$a   b$	sau	$3   5$	7	1 dacă cel puțin un bit are valoarea 1.
$a \wedge b$	sau exclusiv	$3 \wedge 5$	6	1 dacă biții au valori diferite.
$\sim a$	complement	$\sim 3$	-4	Inversează valoarea biților.
$n \ll p$	deplasare stânga fără semn	$3 \ll 2$	12	Deplasează biții lui $n$ la stânga cu $p$ poziții. Biți având valoarea zero sunt introduși în pozițiile de rang nesemnificativ.

$n \gg p$	deplasare dreapta cu semn	$5 \gg 2$	1	Deplasează biții lui $n$ la dreapta cu $p$ poziții. Dacă $n$ este un număr cu semn reprezentat în forma <i>complement față de 2</i> , semnul este propagat în pozițiile de rang semnificativ.
$n \ggg p$	deplasare dreapta fără semn	$-4 \ggg 28$	15	Deplasează biții lui $n$ la dreapta cu $p$ poziții. Biți având valoarea zero se introduc în pozițiile de rang semnificativ.

### Utilizarea împachetării / despachetării

O utilizare frecventă a operatorilor la nivel de bit (deplasări combinate cu „și” pentru extragere de valori și deplasări combinate cu „sau” pentru adăugare valori) este aceea de a codifica mai multe valori într-o singură valoare de tip întreg. Procedeu se numește împachetare, iar extragerea rezultatului despachetare. Câmpurile de biți oferă o altă metodă de a face același lucru. Spre exemplu, să presupunem că avem de a face cu următoarele variabile întregi : `virsta` (domeniu 0-127), `sex` (domeniu 0-1), `inaltime` (domeniu 0-255). Acestea pot fi împachetate/despachetate într-o singură variabilă `short` (întreg reprezentat pe doi octeți) după cum urmează (sau în variante similare).

```
int virsta, sex, inaltime;
short info_impachetata;

// împachetare
info_impachetata = (short)((((virsta << 1) | sex) << 8) | inaltime);

// despachetare
inaltime = info_impachetata & 0xff;
sex = (info_impachetata >>> 8) & 1;
virsta = (info_impachetata >>> 9) & 0x7f;
```

### Utilizarea biților fanion (flag)

Anumite funcții de bibliotecă interpretează valoarea unui întreg ca și un șir de biți în care fiecare reprezintă o valoare booleană (true/false) . Această presupunere are avantajul că economisește spațiu de memorare și în același timp poate să fie procesată rapid.

### Utilizarea deplasărilor: deplasarea la stânga – înmulțire cu 2; deplasarea la dreapta – împărțire cu 2

Operațiile de deplasare a biților sunt mai rapide decât înmulțirile respectiv împărțirile.

```
y = x << 3;          // Atribue lui y valoarea 8*x
y = (x << 2) + x;    // Atribue lui y valoarea 5*x
```

### Utilizarea operatorului sau-exclusiv pentru a comuta pe unu/zero

Uneori se folosește operatorul *sau-exclusiv* pentru a comuta valorile 1 și 0.

```
x = x ^ 1;          // sau în stil criptic x ^= 1;
```

Această instrucțiune plasată într-o buclă va schimba alternativ 0 cu 1.

### Utilizare îndoielnică : Schimbare de valori folosind operatorul sau-exclusiv

Se folosește operatorul *sau-exclusiv* pentru a schimba între ele două valori ( $x$  și  $y$ ). Aceasta este traducerea în Java a unei secvențe dintr-un limbaj de asamblare, unde nu se dispunea de spațiu de memorare pentru o variabilă temporară. Nu se recomandă să utilizați o schemă de acest gen, ea trebuie privită doar ca și o curiozitate.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

### Nu confundați operatorii && și &

Nu confundați operatorul `&&`, care produce operația *și logic*, cu operatorul `&`, care este o *transformare de biți*, numită „și pe biți”. Acesta din urmă este mai puțin obișnuit . Cu toate că operația *și pe biți* poate fi folosită împreună cu operanzi de tip boolean, acest lucru este extrem de rar și aproape întotdeauna generează o eroare de programare.

## 2.3. Rezumat pentru expresii

**Parantezele ()** au trei utilizări:

1. Gruparea pentru a controla ordinea de evaluare sau pentru claritate.  
Exemplu:  $(a + b) * (c - d)$
2. După o metodă pentru a cuprinde parametrii. Exemplu:  $x = \text{suma}(a, b);$
3. În jurul unui tip pentru a forma determina o conversie explicită (*tip*). Exemplu:  $i = (\text{int})x;$

**Ordinea de evaluare**

- Operatorii cu precedența mai mare se aplică înaintea celor cu precedență mai mică.
- La precedență egală, evaluarea este de la stânga la dreapta, cu excepția operatorilor unari, a atribuirii și operatorului condițional.

**Abrevieri folosite**

i, j – valori întregi (int, long, short, byte, char).  
m, n – valori numerice (întregi, double, sau float).  
b, c - boolean; x, y – orice tip sau obiect.  
s, t - String; a - tablou; o - obiect; co - clasă sau obiect

Precedența Operatorilor	
. [] (argumente) post ++ -- ! ~ op.unari + - pre ++ -- (tip) new * / % + - << >> >>> < <= > >= instanceof == != & ^   &&    ?: = += -= etc.	Memorați doar pentru operatorii unari, * / % + - comparații &&    = asignări Folosiți () pentru toate celelalte

Operatori aritmetici	
<i>Rezultatul operațiilor aritmetice este double dacă oricare dintre operanzi este double, altfel float dacă oricare dintre operanzi este float, altfel long dacă oricare dintre operanzi este long, altfel int.</i>	
++i	Adună 1 la i înainte de a folosi valoarea în expresia curentă
--i	Ca mai sus, dar scade
i++	Adună 1 la i după ce s-a folosit valoarea în expresia curentă
i--	Ca mai sus, dar scade
n + m	Adunare. D.e. 7+5 este 12, 3 + 0.14 este 3.14
n - m	Scădere
n * m	Înmulțire
n / m	Împărțire. D.e. 3.0 / 2 este 1.5 , 3 / 2 este 1
n % m	Restul (Mod) după împărțirea lui n prin m. D.e. 7 % 3 este 1

Compararea valorilor primitive	
<i>Rezultatul tuturor comparațiilor este boolean (true sau false).</i>	
== != < <= > >=	

<b>Operatori logici</b>	
<i>Operanzii trebuie să fie booleani. Rezultatul este boolean.</i>	
b && c	Condiționalul "și". Este true dacă amândoi operanzii sunt true, altfel false. Evaluarea se face cu scurtcircuitare. <i>D.e.</i> (false && orice) este false.
b    c	Condiționalul "sau". Este true dacă oricare dintre operanzi este true, altfel false. Evaluarea se face cu scurtcircuitare. <i>D.e.</i> (true    orice) este true.
!b	true dacă b este false, false dacă b este true.
b & c	"Și" care evaluează întotdeauna ambii operanzi ( <i>nu</i> scurtcircuit).
b   c	"Sau" care evaluează întotdeauna ambii operanzi ( <i>nu</i> scurtcircuit).
b ^ c	"Sau-exclusiv" La fel ca b != c

<b>Operatorul condițional</b>	
b ? x : y	dacă b este true, valoarea este x, altfel y. x și y trebuie să fie de același tip.

<b>Operatori de asignare (atribuire)</b>	
=	Partea stângă trebuie să fie o lvaloare (vezi C).
+= -= *= ...	Toți operatorii binari, cu excepția lui && și    pot fi combinați cu o asignare. <i>D.e.</i> a += 1 este la fel ca a = a + 1

<b>Operatori pe biți</b>	
<i>Operatorii pe biți operează pe biți sau int. Rezultatul este int.</i>	
i & j	Biților li se aplică operatorul "și". 1 dacă ambii biți sunt 1. 5 & 3 este 1.
i   j	Biților li se aplică "sau". 1 dacă oricare dintre biți este 1. 5   3 este 7.
i ^ j	Biților li se aplică "sau-exclusiv". 1 dacă biții sunt diferiți. 5 ^ 3 este 6.
~i	Biții sunt complementați (0 -> 1, 1 -> 0)
i << j	Biții din i sunt deplasați cu j biți spre stânga; sunt inserate zerouri prin partea dreaptă. 5 << 2 este 20.
i >> j	Biții din i sunt deplasați cu j biți spre dreapta. Se inserează biți ca bitul de semn prin stânga. 5 >> 2 este 1.
i >>> j	Biții din i sunt deplasați cu j biți spre dreapta. Prin stânga se inserează zerouri.

<b>Conversii explicite</b>	
<i>Folosii conversii explicite atunci când "îngustați" gama unei valori. De la cea mai restrânsă la cea mai largă, ordonarea tipurilor primitive este: byte, short, char, int, long, float, double. Obiectele se pot atribui fără precizare explicită a tipului în sus pe ierarhia de moștenire. Este nevoie de precizarea explicită a tipului pentru a merge în jos pe ierarhia de moștenire (downcasting).</i>	
(t)x	Face din x o variabilă convertită la tipul t (nu modifică pe x)

<b>Operatori asupra obiectelor</b>	
co.f	Membru. Câmpul f sau metoda unui obiect sa a clasei co.
x instanceof co	true dacă obiectul x este o instanță a clasei co, sau o instanță de clasa lui co.
a[i]	Acces la elementele unui tablou.
s + t	Concatenare de obiecte șir dacă unul sau ambii operanzi sunt String.
x == y	True dacă x și y se referă la același obiect, altfel fals (chiar dacă valorile obiectelor sunt identice).
x != y	Ca mai sus pentru inegalitate.
comparații	Compară <b>valori</b> de obiecte cu .equals() sau .compareTo()
x = y	Asignarea copiază <i>referința</i> , nu obiectul.

### 3. Controlul fluxului execuției

Cele ce urmează constituie un rezumat al sintaxei instrucțiunilor de control. Fiecare instrucțiune de control este o instrucțiune logică, care de multe ori include un *bloc* de instrucțiuni între acolade {}. Exemplele presupun că blocul conține mai mult de o instrucțiune.

**Indentarea** este esențială. Patru spații sunt cel mai frecvent utilizate.

#### 3.1. Selecția (*if*, *switch*)

##### 3.1.1. Instrucțiunea *if*

```
//----- instrucțiune if cu o ramură
if (expresie)
{
    instrucțiuni // execută aceste instrucțiuni dacă expresie se evaluează la
                  adevărat
}

//----- instrucțiune if cu două ramuri (pe adevărat și fals)
if (expresie)
{
    instrucțiuni // execută aceste instrucțiuni dacă expresie se evaluează la
                  adevărat
}
else
{
    instrucțiuni // execută aceste instrucțiuni dacă expresie se evaluează la
                  fals
}

//----- instrucțiuni if înlănțuite
if (expresie1)
{
    instrucțiuni // execută aceste instrucțiuni dacă expresie1 este adevărată
}
else if (expresie2)
{
    instrucțiuni // execută aceste instrucțiuni dacă expresie2 este adevărată
}
else if (expresie3)
{
    instrucțiuni // execută aceste instrucțiuni dacă expresie3 este adevărată
    . . .
}
else
{
    instrucțiuni // execută aceste instrucțiuni dacă nici o expresie n-a fost
                  adevărată
}
```

##### 3.1.2. Instrucțiunea *switch*

Efectul instrucțiunii **switch** este alegerea unor instrucțiuni de executat în funcție de valoarea întreagă a unei expresii. Același efect poate fi obținut cu o serie de instrucțiuni **if** înlănțuite, dar în anumite cazuri instrucțiunea **switch** este mai ușor de citit, iar unele în compilatoare poate produce cod mai eficient. Instrucțiunea **break** provoacă ieșirea din instrucțiune **switch**. Dacă nu există **break** la sfârșitul unui caz (**case**), execuția continuă în cazul următor, iar acest lucru este de multe ori o eroare de logică a programului.

```

switch (expr) {
    case c1:
        instrucțiuni // execută aceste instrucțiuni dacă expr == c1
        break;
    case c2:
        instrucțiuni // execută aceste instrucțiuni dacă expr == c2
        break;
    case c2:
    case c3:
    case c4: // cazurile au o tratare comună.
        instrucțiuni // execută aceste instrucțiuni dacă expr == oricare
                    dintre c-uri
        break;
    . . .
    default:
        instrucțiuni // execută aceste instrucțiuni dacă expr != oricare
                    dintre cele de mai sus
}

```

## 3.2. Instrucțiuni de ciclare

### 3.2.1. while

Instrucțiunea **while** testează *expresie*. Dacă *expresie* se evaluează la adevărat, se execută corpul instrucțiunii while. Dacă este falsă, execuția continuă cu instrucțiunea de după corpul instrucțiunii while. De fiecare dată după executarea corpului, execuția reîncepe cu testul. Acest proces continuă până când *expresie* devine falsă sau o altă instrucțiune (**break** sau **return**) întrerupe ciclarea.

```

while (testExpresie) {
    instrucțiuni
}

```

### 3.2.2. for

Multe bucle au o inițializare înainte de buclă și un fel de "incrementare" înainte de următoarea parcurgere a ciclului. Bucloa **for** reprezintă modalitatea standard pentru combinarea acestor trei părți.

```

for (instrucțiuneInițială; testExpr; instrucțiuneIncrement) {
    instrucțiuni
}

```

Aceasta este la fel cu (cu excepția faptului că instrucțiunea **continue** în bucla **for** va executa și incrementarea):

```

instrucțiuneInițială;
while (testExpr)
{
    instrucțiuni
    instrucțiuneIncrement
}

```

### 3.2.3. do

Aceasta este cel mai puțin folosită dintre instrucțiunile de ciclare, dar câteodată este necesară o buclă care se execută o dată înainte de test.

```

do
{
    instrucțiuni
}
while (testExpr);

```



### 3.2.4. Alte instrucțiuni de control pentru cicluri

Toate instrucțiunile de ciclare pot fi etichetate, astfel încât se poate folosi `break` și `continue` la orice adâncime de imbricare. Etichetele trebuie să le precedă folosirea.

```
break;           //ieși din bucla sau switch-ul cel mai interior

break eticheta; //ieși din bucla etichetată cu eticheta

continue;       //începe următoarea iterație a ciclului

continue eticheta; //începe următorul ciclu cu eticheta eticheta
```

Plasați eticheta urmată de două puncte în fața buclei.

```
exteriora: for ( . . . ) {
    . . .
    continue exteriora;
}
```

## 4. Mersul lucrării

Scrieți scurte programe în Java pentru:

- 4.1. Orice număr natural par mai mare decât 2 se poate scrie ca sumă a două numere prime – conjectura lui Goldbach. Scrieți un program Java care să verifice această conjectură pentru numere situate între  $m$  și  $n$ .
- 4.2. Implementați metoda Newton de găsire a rădăcinii pătrate a unui polinom de grad 2 de forma  $f(x) = ax^2 + bx + c = 0$ . Metoda Newton găsește o soluție aproximativă a ecuației, folosind un algoritm iterativ:
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
 Soluția este găsită atunci când algoritmul converge ( $x_{n+1} - x_n < \varepsilon$ ). Pornind de la o valoare inițială  $x_0$ , găsiți soluția polinomului, pentru diferite valori ale lui  $\varepsilon \in [0.01, 0.000001]$ . Afișați soluția găsită și numărul de iterații ( $n$ ).
- 4.3. Fiind dată o variabilă care stochează o referință către un String  $s$ , determinați:
  - numărul de consoane și vocale
  - indicii pentru o vocală introdusă din linia de comandă.
- 4.4. Calculați șansele de câștig la loterie (6 din 49).
- 4.5. Simulați extragerea numerelor la loterie (pentru generarea numerelor aleatoare folosiți `Math.random`).
- 4.6. Afișați în ordine crescătoare/descrescătoare numerele extrase la loterie fără a face sortări sau a folosi tablouri. (Sugestie: folosiți împachetarea/despachetarea/extragerea de biți stocați într-un long)