C4

Operatorii limbajului Java

- unari
- de atribuire
- aritmetici
- relationali (de comparatie)
- pe biti
- logici
- conditional

Obiective

După parcurgerea acestui curs ar trebuie sa puteţi:

- ☐ transcrie expresii aritmetice in expresii Java formate corect;
- ☐ scrie expresii logice ce asigura testarea unor condiţii de funcţionare a aplicaţiei;
- ☐ manipula la nivel de bit tipurile întregi ale limbajului;
- ☐ optimiza scrierea de expresii Java pentru viteza mai mare sau cod mai compact.

Tipuri de operatori

În Java avem 5 tipuri de operatori.

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

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

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

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

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

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

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

Atribuirea

Formă: `nume = expresie`

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

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

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

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

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

Operatorii aritmetici

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

```
int a,b,c,d,e;
a = 1 + 2; // + pentru ADUNARE
b = 1 - 2; // - pentru SCĂDERE
c = a * 2; // * pentru ÎNMULŢIRE
d = c / b; // / pentru ÎMPĂRŢIRE
e = a % 2; // % pentru RESTUL ÎMPĂRŢIRII
```

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

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

Aritmetica numerelor întregi

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

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

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

Promovarea și forțarea de tip

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

Promovare: conversie către un tip cu domeniu mai larg

Forţare: conversie către un tip cu domeniu mai îngust

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

Fie atriburile:

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

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

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

Promovarea de tip

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

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

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

Fie declaraţia:

```
long varsta = 37;
```

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

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

Promovările atutomate sunt prezentate în continuare:

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

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

Forţarea de tip

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

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

unde:

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

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

```
int n1 = 102;          //32 de biti pentru stocarea valorii
int n2 = 13;           //32 de biti pentru stocarea valorii
byte n3;               //8 biti de memorie rezervata
```

```
n3 = (byte)(n1 + n2); // nu exista pierdere de date
```

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

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

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

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

```
public class fortareConversii {

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

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

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

Rezultate:

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

Promovarea la întregi

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

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

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

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

Promovarea la reali

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

```
float f1 = 23.5;
```

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

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

Incrementarea şi decrementarea

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

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

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

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

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

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

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

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

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

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

Rezultate:

```
10 11 12 12
```

Fie codul:

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

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

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

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

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

Operatori relaţionali (de comparare)

Întorc rezultate de tipul boolean

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

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

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

rezultat = *expresie1* `operatorrelational` *expresie2*

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

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

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

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

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

    }
}
```

Rezultatul:

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

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

Operatori logici (booleeni)

Permit formarea de expresii logice pe baza rezultatelor operatorilor relaţionali

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

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

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

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

Există şi operatori logici care se evaluează fără scurtcircuitare. Ei au o scriere diferită, astfel pentru ŞI se scrie & iar pentru SAU se scrie |. Utilizarea lor se face în situaţii în care scurtcircuitarea poate produce rezultate ciudate, de exemplu fie expresia **`func1() && func2()`**, dacă funcţia

`func1()` va întoarce rezultatul **`false`** funcţia **`func2()`** nu va mai fi apelată. Vor exista, deci, cazuri în care **`func2()`** nu va fi apelată, iar dacă aceasta face ceva semnificativ în cod lipsa apelului va duce la rezultate eronare.

Tabel de adevăr pentru **ŞI**

&&	false	true
false	false	false
true	false	true

Tabel de adevăr pentru **SAU**

||	false	true
false	false	true
true	true	true

Tabel de adevăr pentru **SAU EXCLUSIV**

^	false	true
false	false	true
true	true	false

Operatori pe biti

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

Operează cu decompunerea binară a operanzilor

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

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

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

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

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

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

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

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

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

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

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

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

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

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

semnificativ bit se pune valoarea 0.

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

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

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

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

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


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

        c = a << 2;
        System.out.println( a + " << 2 = " + c);
```

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

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


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

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

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

    }

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

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

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

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

Rezultate:

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

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

```
00000000000000000000000000001110

14 ^ 4 = 10
00000000000000000000000000001110 ^
00000000000000000000000000000100
--------------------------------
00000000000000000000000000001010

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

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

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

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

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

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

Atribuirea compusă

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

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

se poate scrie :

expresie1 *op* = expresie2

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

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

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

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

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

Prioritatea operatorilor Java

Prioritatea determină ordinea de rezolvare a operatorilor

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

Utilizarea parantezelor rotunde redefineşte priorităţile

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

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

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

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

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

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

Fie expresia:

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

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

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

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

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

se va evalua astfel:

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

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

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

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

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

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

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

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

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



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

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

    }
}
```

Rezultate:

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

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

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

Conversii
(int)42.65 = 42

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

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

Functii matematice si constante
sqrt(3) = 1.7320508075688772
```

```
sin(3) = 0.1411200080598672
cos(3) = -0.9899924966004454
tan(3) = -0.1425465430742778
atan(3) = 1.2490457723982544
exp(3) = 20.085536923187668
log(3) = 1.0986122886681096
pow(3,3) = 27.0
PI = 3.141592653589793
E = 2.718281828459045
```

C4- Întrebari

1. Explicaţi diferenţele intre formele prefixate si postfixate in cazul operatorilor de incrementare.
2. Explicaţi conversiile implicite ce sa realizarea in evaluarea expresiei: 1+2.0/3.f
3. Utilizati operatorul condiţional pentru a calcula minimul si maximul dintre doua numere reale.
4. Scrieţi codul pentru toate formele de incrementare a unei variabile simple întregi si explicaţi fiecare varianta de implementare.

BIBLIOGRAFIE

1. http://www.oracle.com/technetwork/java/javase/documentation/index.html

2. http://docs.oracle.com/javase/6/docs/

3. Stefan Tanasa, Cristian Olaru, Stefan Andrei, Java de la 0 la expert, Polirom, 2003, ISBN: 973-681-201-4.

4. Herber Schild, Java 2 - The Complete Reference, Fourth Edition, Osborne, 2001, ISBN: 0-07-213084-9.

5. Deitel H.M., Deitel P. J., Java - How to programm, Fith Edition, Prentice Hall, 2003, ISBN: 0-13-120236-7.

6. http://www.east.utcluj.ro/mb/mep/antal/downloads.html