Laborator Java - continuare
-2 -

Metode pentru afişarea pe ecran a şirurilor

`System.out.println(<şir>)` cu trece la linie nouă şi
`System.out.println(<şir>)` cu afişarea în continuare pe aceeaşi linie

Unităţile lexicale ale limbajului Java

Unităţile lexicale, numite şi *lexeme* (engl. *token, lexeme*) sunt construcţiile elementare ale limbajului ("atomii" acestuia). Acestea sunt simboluri, formate din unul sau mai multe caractere, care au o anumită semnificaţie în limbaj. După rolul îndeplinit, unităţile lexicale sunt: *identificatori, cuvinte cheie, cuvinte rezervate, literali, separatori, operatori, comentarii şi spaţii.*

De exemplu, în programul:

```
class AfisareSiruri {
   public static void main(String args[]) {
     System.out.println("sirul 1");
     System.out.println("sirul 2"); // se afiseaza sub sirul 1
     System.out.println("AB"+"CDE"); // se afiseaza ABCDE
     System.out.println("ab"+"cd"+"ef"); // se afiseaza abcdef
     System.out.println(("ab"+"cd")+"ef"); // asociativitate
     System.out.println("ab"+("cd"+"ef"));
     /* Urmatoarele trei instructiuni afiseaza in continuare,
        pe o singura linie  */
     System.out.print("pqrst"); // nu se trece la linie noua
     System.out.print("UVW");   // se afiseaza in continuare
     System.out.print("xyz\n"); // echivalent cu println("xyz")
     /* Trecerea la linia urmatoare se face datorita prezentei
        caracterului de control \n in sirul "xyz\n" */
     System.out.println("ultima linie afisata");
     System.out.println((int)'A'+" "+(int)'q');
   }
 }
```

Distingem urmatoarele unităţi lexicale:

- cuvinte cheie: `class, public, static, void`;
- identificatori: `AfisareSiruri, main, String, args, System, out, print, println`;
- literali: `"sirul 1", "sirul 2", "AB", "CDE", "ab", "cd", "ef";`
-separatori:`{}()[],;`
−operator:+;
-comentarii: `/* Exersarea metodelor print si println */`
`// se afiseaza sub sirul 1`

Vom analiza acum separat fiecare din aceste categorii de unităţi lexicale.

Identificatori

Numele date programelor sau componentelor acestora (clase, variabile, metode etc.) se numesc ***identificatori***. Identificatorii se aleg de către programator, respectând anumite reguli.

În limbajul Java, identificatorii sunt şiruri formate din litere, cifre şi caractere de subliniere ('_'), care încep cu o literă. Lungimea identificatorului nu prezintă importanţă, însă acesta nu poate conţine spaţii libere sau alte caractere, decât cele menţionate aici.

Exemple de identificatori valabili:

```
PrimaClasa
aplha
viteza
v15XB7
pretDeVanzare
pret_de_vanzare
```

Este clar acum că şi exemplele date anterior (`Afisari, main, String, args, System, out, print, println)` sunt, de asemenea, identificatori.

Se obişnuieşte ca numele de clase să înceapă cu literă majusculă. De asemenea, se obisnuieşte ca separarea între cuvinte, în cadrul identificatorilor compuşi din mai multe cuvinte ale limbajului natural, să se facă începând fiecare cuvant nou cu literă majusculă, dar se poate face şi prin caracterul de subliniere '_'. Acestea nu sunt însă reguli sintactice, ci doar convenţii neobligatorii.

Programatorul poate adopta orice identificatori care respectă regulile şi convenţiile de mai sus şi care nu sunt cuvinte cheie sau cuvinte rezervate. Desigur însă că folosirea unor identificatori care au semnificaţie pentru om, cum ar fi *viteza* sau *PretDeVanzare* este preferabilă celor fără semnificaţie, cum ar fi *v15XB7,* deoarece uşurează înţelegerea şi urmărirea programului. Amintim însa că, pentru calculator, identificatorii nu au nici o alta semnificaţie, deci, din acest punct de vedere, toate exemplele de identificatori date aici sunt la fel de bune.

Cuvinte cheie

În orice limbaj de programare, există un set de cuvinte, numite ***cuvinte cheie,*** care sunt considerate simboluri sintactice şi nu pot fi folosite în program ca identificatori.

În limbajul Java, există următoarele **cuvinte cheie**:

`abstract`	`double`	`int`	`strictfp`
`boolean`	`else`	`interface`	`super`
`break`	`extends`	`long`	`switch`
`byte`	`final`	`native`	`synchronized`
`case`	`finally`	`new`	`this`
`catch`	`float`	`package`	`throw`
`char`	`for`	`private`	`throws`
`class`	`goto`	`protected`	`transient`
`const`	`if`	`public`	`try`
`continue`	`implements`	`return`	`void`

default	import	short	volatile
do	instanceof	static	while

Dintre acestea, `const` şi `goto` nu sunt folosite în prezent, dar ele au fost introduse în tabela cuvintelor cheie în vederea unei eventuale utilizări viitoare.

Observăm acum că toate exemplele de cuvinte cheie date la începutul acestei secţiuni (`class, public, static, void`) sunt prezente în tabela de mai sus.

Cuvinte rezervate

Se consideră cuvinte rezervate acele cuvinte, care nu pot fi folosite ca identificatori, având semnificaţii speciale. Cuvintele cheie sunt şi ele considerate în majoritatea limbajelor, inclusiv Java, drept cuvinte rezervate. În afară de acestea, în limbajul Java există urmatoarele ***cuvinte rezervate:*** `true, false, null.`

Primele două sunt valorile logice *adevărat* şi *fals*, iar al treilea are semnificaţia de referinţă nulă. De fapt, aceste cuvinte rezervate sunt forme speciale de *literali*.

Literali

Literalii sunt *reprezentările* în fişierele sursă ale *valorilor constante*. Exemple de literali:
-caractere: `'a', 'A', '+', '$', '5'`;
-şiruri de caractere: `"sir de caractere", "abc$79.28#^z"`;
-numere întregi: `14726, -25413`;
-numere reale: `12.7389, -0.05673, 2.3075E12, -1.4237E-5`;
-valori logice: `true, false`;
-referinţa nulă: `null`.

Separatori

Separatorul este un caracter care delimitează formele sintactice sau le separă între ele. În limbajul Java se folosesc următorii ***separatori***:

`{}()[];,.`
Spaţiul liber şi operatorii indeplinesc, de asemenea, rolul de separatori.

Operatori

Operatorii sunt simboluri ale unor o*peraţii*. Am folosit deja simbolul + ca operator de concatenare (deci simbol al operaţiei de concatenare). Operatorul poate fi format din unul sau mai multe caractere. Entitatea asupra căreia se aplică operatorul se numeşte *operand*. După numărul de operanzi, operatorii pot fi *unari*, *binari* sau *ternari*.

După numărul de operanzi deosebim:

operatori unari, care se aplică unui singur operand, de ex. operatorul - în expresia `-x`; utilizarea operatorului unar se face, de regula, sub forma "prefix" `operator operand`, în care operatorul se pune în faţa operandului; uneori însă se foloseste şi forma "postfix" `operand operator`

în care operatorul se pune după operand;

operatori binari, care se aplică asupra a doi operanzi, operatorul fiind situat între aceştia; de ex. operatorul de concatenare + în expresia `"acesta este " + "un exemplu"` sau operatorul de adunare a două numere + în expresia `17+28`. Operatorii binari se folosesc sub forma "infix"
`operand1 operator operand2`
în care operatorul este situat între cei doi operanzi;

operatori ternari, care se aplică asupra a trei operanzi; în limbajul Java există un singur operator ternar (`? :`) folosit în expresiile condiţionale. Operatorul ternar se scrie sub forma
`operand1 ? operand2 : operand3`

în care `operand1` are o valoare logică (`true` sau `false`), iar ceilalti doi operanzi sunt expresii aritmetice sau logice (dar de acelaşi tip).

Din exemplele de mai sus, observăm că semnificaţia unui operator poate să depindă de context, ca în cazul operatorului +, care a fost folosit atât pentru operaţia de concatenare a şirurilor, cât şi pentru cea de adunare a numerelor.

Din punct de vedere matematic, operatorii sunt **funcţii** cu unul, două sau trei argumente (argumentele fiind operanzii). De exemplu, expresia `a+b`, în care + este un operator binar, iar `a` şi `b` sunt operanzi, este o *funcţie* de argumente `a` si `b`, care are ca valoare suma valorilor celor două argumente.

După efectul operatorului asupra operanzilor, operatorii pot fi *fără efect lateral,* care lasa valorile operanzilor nemodificate, şi *cu efect lateral*, care modifică valorile operanzilor. Astfel, operatorul + din exemplul anterior, este un operator fără efect lateral. În schimb, în expresia `++a` operatorul de incrementare ++ are efect lateral deoarece, în urma efectuarii operaţiei, valoarea operandului `a` creşte cu o unitate.

Dăm aici o listă a operatorilor folosiţi în limbajul Java.

Operatori unari:

```
+          -          +          --          new          !          ~
() [] {}
```

Operatori binari

```
 + - * / %
 == <= >= !=
 &  |   ^
 &&  ||
 <<  >>  >>>
 =  +=  -=  *=  /=  &=  |= ^= ~= <<=  >>=  >>>=
 .  instanceof
```

Operator ternar

```
?:
```

Remarcăm că operatorii + şi - pot fi atât unari (ca în expresiile `+a` sau `-a`), cât şi binari (ca în

expresiile `a+b` sau `a-b)`.

Variabile

Valoarea variabilei trebuie să fie reprezentată în memoria calculatorului la o anumită *adresă* şi să ocupe acolo un anumit spaţiu (un anumit număr de biţi). În consecinţş, numim **variabilă** o *zonă de memorie* care poarta un *nume* şi care conţine o anumită *valoare*, apartinând unui *tip de date*.
Expresia `System.out.println(a)` are ca efect afişarea pe ecran a *valorii* variabilei `a`.

În limbajul Java, orice variabilă trebuie *declarată* înainte de a fi utilizată. Prin *declararea variabilei* se înţelege precizarea, pentru compilator, a *tipului* şi *numelui* acesteia. *Iniţializarea variabilei* se face atunci, când acesteia i se dă pentru prima dată o valoare şi deci i se alocă spaţiu în memorie. Dacă, la declararea variabilei, aceasta nu este şi iniţializată în mod explicit, atunci ea este iniţializata cu o *valoare implicită*, care va fi specificată la descrierea fiecărui tip.

Iată un exemplu de declaraţie de variabilă:
```
int alpha, beta=3702, k;
```
Aceasta este o *instrucţiune* din program, prin care se specifică următoarele:

- `alpha, beta şi k` sunt numele unor variabile de tipul `int`;
- variabilei `beta` i se dă *valoarea iniţiala* `3702`;
- variabilelor `alpha` si `k` li se dau *valori iniţiale implicite* corespunzătoare tipului `int` (în cazul de faţă valoarea `0`).

Prin convenţie, în limbajul Java numele de variabile încep întotdeauna cu literă mică.

În limbajul Java, se numesc *variabile finale* acele "variabile", ale căror valori nu pot fi modificate prin program. Acestea sunt deci, de fapt, nişte *constante cu nume*. Ele se aseamănă cu variabilele propriu-zise prin faptul că sunt tot perechi nume - valoare, numai că valoarea lor se dă o singură dată, sub forma de iniţializare în declaraţia de tip sau sub forma de atribuire, după care nu mai poate fi modificată.
De exemplu, declaraţia

```
final int ALPHA=17, BETA=-1453;
```
serveşte pentru a specifică faptul că `ALPHA şi BETA` sunt *variabile finale* de tip `int`, ale caror valori sunt, respectiv, `17` si `-1453 şi` nu mai pot fi ulterior modificate (deci `ALPHA şi BETA` sunt, de fapt, nişte constante).

Cuvântul cheie FINAL poate fi folosit pentru o variabilă şi înseamnă ca acea variabilă nu poate fi modificată, sau pentru o metodă şi înseamnă că acea metodă nu poate fi suprascrisă, sau pentru o clasă şi înseamnă că acea clasă nu poate fi extinsă.

Tipuri de date

În limbajul Java, tipurile de date se împart în două categorii: *tipuri**primitive*** şi *tipuri **referinţă***.

Tipurile de date primitive sunt predefinite în limbaj. Aceasta înseamnă că numele, mulţimea de valori, mulţimea de operaţii şi tipul rezultatului operaţiilor pentu fiecare tip primitiv sunt impuse prin limbaj şi, deci, nu trebuie definite şi nu pot fi modificate de programator.

Tipurile de date primitive în limbajul Java se clasifică astfel:

- tipul `boolean`;
- tipurile numerice
  - tipuri intregi: `byte, short, int, long`;
  - tipuri reale: `float` si `double`;
  - tipul `char`

Tipul boolean

Operatorii booleeni

Operatorul de negaţie este un operator unar fără efect lateral şi se reprezintă prin simbolul ! (semnul exclamării). Expresia `!a,` în care `a` este un operand boolean, se citeşte *non*-`a` şi se interpretează ca negaţia lui `a`: daca `a` are valoarea `true`, atunci `!a` are valoarea `false` şi invers.

Operatorii logici binari sunt operatori fără efect lateral, prin care se realizează operaţiile logice ŞI, SAU şi SAU-EXCLUSIV.
- Operatorii `&` si `&&` realizeaza operatia logica ŞI. Expresiile `a&b` şi `a&&b`, în care `a` şi `b` sunt operanzi de tip boolean, are valoarea `true`(adevărat) dacă şi numai dacă atât `a` cât şi `b` au valoarea true. În celelalte cazuri expresia are valoarea `false`.
- Operatorii `|` si `||` realizează operaţia logică SAU. Expresiile `a|b` şi `a||b` , în care `a` şi `b` sunt operanzi de tip boolean, are valoarea `false` dacă şi numai dacă ambii operanzi au valoarea `false`. În celelalte cazuri expresia are valoarea `true`.
- Operatorul `^` realizează operatia logică SAU-EXCLUSIV. Expresia `a^b` , în care `a` şi `b` sunt operanzi de tip boolean, are valoarea `true` dacă şi numai dacă cei doi operanzi au valori diferite (unul este adevărat, iar celălalt fals). Dacă cei doi operanzi au valori identice, valoarea expresiei este `false`.

Deosebirile între operatorii `&` şi `&&`, respectiv între `|` şi `||` sunt următoarele:
- în cazul operatorilor `&` şi `|` se evaluează în mod obligatoriu ambii operanzi;
- în cazul operatorului `&&`, numit ŞI-condiţional, evaluarea celui de al doilea operand se face numai dacă primul operand are valoarea `true`; altfel, se consideră că operaţia dă valoarea `false`, fără a se mai evalua valoarea celui de al doilea operand;
- în cazul operatorului `||`, numit SAU-condiţional, evaluarea celui de al doilea operand se face numai dacă primul operand are valoarea `false`; altfel, se consideră ca operaţia dă valoarea `true`, fără a se mai evalua valoarea celui de al doilea operand.

Acţiunea operatorilor logici este prezentată sintetic în tabela de mai jos, în care `a` şi `b` sunt doi operanzi logici.

a	b	a&b	a&&b	a|b	a||b	a^b
true	true	true	true	true	true	false
true	false	false	false	true	true	true
false	false	false	false	false	true	true
false	false	false	false	false	false	false

Testaţi următorul program:

```
class TipBoolean {
 public static void main(String args[]) {
     boolean alpha=true, beta=false, p, q, r,s;
     p=!alpha;
```

```
      q=alpha&&beta;
      r=alpha||beta;
      s=alpha^beta;

      System.out.println("   alpha="+alpha+"   beta="+beta+"   p="+p+
 " q="+q+" r="+r+" s="+s);
      System.out.println("alpha&&beta="+(alpha&&beta)+
"alpha||beta="+(alpha||beta));
      System.out.println("alpha==beta: "+(alpha==beta));
      System.out.println("alpha!=beta: "+(alpha!=beta));
      }
}
```

Tipuri numerice

Tipurile de date numerice în Java sunt următoarele:

- tipuri întregi: `byte, short, int, long`;
- tipuri reale (în virgulă mobilă): `float` şi `double`;
- tipul `char`

În limbajul Java, **conversia de tip implicită** se face atunci când prin conversie nu se pierde informaţie. De exemplu, dacă în expresia a=b variabila a este de tip int, iar b este de tip short sau byte, valoarea variabilei b va fi automat convertită la tipul int înainte de atribuire.

În tabela de mai jos sunt indicate cu X toate conversiile de tip care se pot realiza inplicit. În coloana din stânga este tipul datei care este supusa conversiei, iar în capul tabelei (pe prima linie) tipul către care se face conversia.

	byte	short	int	long	float	double
byte		X	X	X	X	X
short			X	X	X	X
char			X	X	X	X
int				X	X	X
long					X	X
float						X

Conversia de tip explicită se face prin operatorul *unar* numit ***cast***, care are forma (*tip*), adică este format din numele tipului către care se face conversia, cuprins între paranteze.

Operaţiile aritmetice

Operaţiile aritmetice sunt cele care se aplică unor operanzi numerici, având ca rezultate tot numere. După numărul de operanzi, ele pot fi unare sau binare. Unele operaţii aritmetice unare au şi efect lateral.

Operatori unari *fără* efect lateral:

Operator	Exemplu de expresie	Valoarea expresiei
+	+a	aceeaşi cu valoarea operandului
-	-a	valoarea operandului cu semn schimbat

Operatori unari *cu* efect lateral

Operator	Expresie	Operatie	Valoarea expresiei	Efect lateral
`++`	`++a`	preincrementare	`a+1`	valoarea variabilei `a` creste cu 1
`++`	`a++`	postincrementare	`a`	valoarea variabilei `a` creste cu 1
`--`	`--a`	predecrementare	`a-1`	valoarea variabilei `a` scade cu 1
`--`	`a--`	postdecrementare	`a`	valoarea variabilei `a` scade cu 1

Operatori binari

Operatorii binari nu au efect lateral - deci nu modifică valorile operanzilor - şi sunt daţi în tabela de mai jos.

Operator	Expresie	Operatie	Valoarea expresiei
`+`	`a+b`	adunare	suma valorilor operanzilor
`-`	`a-b`	scadere	diferenta valorilor operanzilor
`*`	`a*b`	inmultire	produsul valorilor operanzilor
`/`	`a/b`	impartire	catul (rezultatul impartirii) primului operand la al doilea
`%`	`a%b`	modulo	restul impartirii intregi a primului operand la al doilea

Operaţii de atribuire compusă

Urmând tradiţia limbajului C, în limbajul Java există şi operatori de atribuire compusă, în care operaţia de atribuire este combinată cu una din operaţiile aritmetice.

Operatorii de atribuire compusă sunt următorii: `+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=`.
Se observă că fiecare din aceşti operatori are forma `op=` în care `op` este un operator aritmetic binar.
Expresia

```
variabila op= operand
```

în care `op` este un operator aritmetic binar, este echivalentă cu expresia

```
variabila = variabila op operand
```

şi se evaluează astfel:
- se calculează mai întâi valoarea expresiei `(variabila op operand)` în care `variabila` intră cu valoarea sa anterioară;
- valoarea astfel calculată se *atribuie* ca *noua valoare* a variabilei din partea stânga. Aceasta nouaă valoare este, totodată, şi valoare a expresiei

Exemplu
Fie `x=3.72` si `y=0.19` două variabile de tip double. Expresia `x+=y` se calculeaza, la fel ca expresia `x=x+y`, în modul următor:

- se calculează valoarea expresiei `x+y`, care este 3.91;
- se atribuie variabilei `x` valoarea 3.91 (efectul lateral);
- valoarea astfel atribuită (3.91) este considerată şi drept valoare a expresiei x+=y.

Comparaţia

Comparaţiile sunt operaţii *binare* fără efect lateral, în care se compară două numere, obţinându-se ca rezultat o valoare de tip `boolean`. Operatorii prin care se efectuează comparaţia a două numere se numesc ***operatori relaţionali*** şi sunt daţi în tabela de mai jos.

Operator	Semnificatie
<	mai mic decât
<=	mai mic decât sau egal cu
>	mai mare decât
>=	mai mare decât sau egal cu
==	este egal cu
!=	este diferit de

Operatorii relaţionali se aplică tuturor tipurilor de date numerice, inclusiv celor de tip `char`.

Tipuri de date întregi

Tipurile de date întregi sunt `byte, short, int, long şi char`. Conceptual, datele care aparţin tipurilor `byte, short, int şi long` sunt *numere întregi*, în timp ce tipul `char` conţine caractere (litere, cifre, semne de punctuaţie etc).

Întrucât caracterele se codifică în memoria calculatorului prin numere întregi fără semn, în limbajul Java asupra lor se pot aplica toate operaţiile pentru numere întregi. Totusi, datorită particularităţilor pe care le prezintă, vom trata tipul `char` separat.

Tipul	Lungimea	Intervalul de valori
byte	1 octet (8 biţi)	[-128, 127]
short	2 octeţi (16 biţi)	[-32768, 32767]
int	4 octeţi (32 biţi)	[-2147483648, 2147683647]
long	8 octeţi (64 biţi)	[-9223372036854775808, 9223372036854775807]

Operaţii şi operatori pentru date de tip întreg

```
/* Verificarea actiunii operatorului (cast) in cazul numerelor intregi
*/

class CastInt {
  public static void main(String args[]) {
    /* Declararea si initializarea variabilelor */
    byte b1=73, b2=-107, b3, b4, b5;
    int i1=91, i2=-103, i3=22195, i4, i5;
    /* Atribuiri cu conversie implicita de la byte la int
    */
    i4=b1;
    i5=b2;
    /* Atribuiri care necesita conversie explicita de la int la byte
    */
    b3=(byte)i1;
    b4=(byte)i2;
```

```
    b5=(byte)i3;
    /* Afisarea rezultatelor */
    System.out.println("i4="+i4+" i5="+i5+"\nb3="+b3+" b4="+b4+
      " b5="+b5);
  }
}
```

a) Atribuirea `i4=b1` unde `b1` are valoarea `73`.

`b1` se reprezinta intern prin octetul `01001001`. Când se face conversia de la `byte` la `int` se extinde aceasta reprezentare externa prin adaugarea la stânga a trei octeţi nuli. Se obţine astfel numărul de tip `int` `00000000000000000000000001001001` care este valoarea 73 pe 4 octeţi.

b) Atribuirea `i5=b2` unde `b2` are valoarea *negativă* `-107`.

`b2` se reprezintă intern prin octetul `10010101` (folosind codul complementar pentru numere negative). Intrucât primul bit este 1, când se face conversia de la `byte` la `int` se adaugă la stânga trei octeţi completaţi în întregime cu cifra 1, obţinându-se reprezentarea internă `11111111111111111111111110010101`. În acest fel *s-a conservat semnul*, obţinându-se tot numărul `-107`, dar reprezentat pe 4 octeţi.

c) Atribuirea `b3=(byte)i1` unde `i1` are valoarea `91`. Reprezentarea internă a lui `i1` este `00000000000000000000000001011011` şi se extinde pe 4 octeţi. Prin conversia de la int la byte se elimină cei trei octeţi *din stânga*, obţinându-se reprezentarea pe un octet `01011011`. Întrucât s-au eliminat numai zerouri din stânga, valoarea a rămas neschimbată.

d) Atribuirea `b4=(byte)i2` unde `i2` are valoarea negativă `-103`. Reprezentarea internă a lui `i2` este `11111111111111111111111110011001` fiind complementul lui 103 extins pe 4 octeţi. Prin conversie la tipul byte se elimină cei trei octeţi din stânga, obţinându-se `10011001` care este tot numărul `-103`, dar reprezentat pe un singur octet.

e) Atribuirea `b5=(byte)i3`, unde `i3` are valoarea `22195`. Reprezentarea internă a lui `i3` pe patru octeti este `00000000000000001010110110110011`. Prin conversie la tipul byte se elimina cei trei octeţi din stânga, reţinându-se octetul `10110011` care se atribuie ca valoare a lui `b5`. Deoarece începe cu bitul 1, aceasta se interpreteaza ca valoarea negativă `-77`. Constatăm astfel că ***prin conversie s-au modificat atât valoarea, cât şi semnul***. Aceasta s-a întâmplat întrucât valoarea de la care a pornit conversia nu se încadra în mulţimea de valori a tipului byte, iar la eliminarea celor trei octeţi din partea stangă s-au pierdut cifre semnificative.

```
/* Testarea operatiilor cu numere intregi */

class Intregi {
  public static void main(String args[]) {
    byte b1=73, b2=-109, b3, b4, b5;
    short s1=9000, s2=-11000, s3, s4;
    int i1=900000, i2=-1100000, i3, i4, i5;
    long m1=10000000000L, m2=-200000000000L, m3, m4;
    b3=(byte)(-b2);
    b4=(byte)(b1+b2);
    b5=++b1;
    s3=(short)(s2/s1);
    s4=(short)(s2%s1);
    i3=i1+i2;
    i4=i1*i2; // restul impartirii s2/s1
    i5=(int)(m2/i1);
    m3=m2-m1;
    m4=m2*m1;
    System.out.println("b3="+b3+" b4="+b4+" b5="+b5);
```

```
    System.out.println("s3="+s3+" s4="+s4);
    System.out.println("i3="+i3+" i4="+i4+" i5="+i5);
    System.out.println("m3="+m3+" m4="+m4);
    System.out.println("b5*b2="+(b5*b2)+" s1*s2="+(s1*s2));
  }
```

- s-a obţinut s3=-1 si nu s3=-1.222222... deoarece în expresia s2/s1 ambii operanzi sunt de tipuri întregi, deci rezultatul este întreg (împărţire întreagă). Din acelaşi motiv s-a obţinut i5=-222222 si nu -222222.222222....;
- la calcularea lui i4 nu s-a obţinut valoarea corectă -990000000000, ci valoarea 2137445376. Cauza este că valoarea depăşeşte marginea superioară a valorilor de tip int, deci s-a produs o depăşire binară şi s-au trunchiat octeţii cei mai semnificativi. Tot o depăşire binară s-a produs şi la calcularea lui m4 (de data aceasta pentru tipul long);
- valorile produselor b5*b2 şi s1*s2 au fost calculate şi afisate corect, deşi ele depăşesc limitele mulţimilor de valori de tip byte, respectiv de tip short. Aceasta s-a produs deoarece expresiile respective sunt de tip int.

```
/* Testarea operatorilor de comparatie in cazul tipurilor
 de date intregi
*/

class ComparInt {
  public static void main(String args[]) {
    byte b1=48, b2=-17;
    short s1=2765, s2=-12970;
    int i1=762983, i2=48, i3=-12970;
    long m1=876432906528L, m2=48;
    System.out.println(b1==b2);
    System.out.println(b1==i2);
    System.out.println(s1!=i2);
    System.out.println(i2!=m2);
    System.out.println(m2>i2);
    System.out.println(m2>=i2);
    System.out.println(s2<i3);
    System.out.println(i3<=s2);
  }
}

/* Testarea operatiilor de deplasare binara */

class Deplasari {
  public static void main(String args[]) {
    byte b1=15, b2;
    int i1=1024, i2=-1024;
    b2=(byte)(b1<<3);
    System.out.println("b1="+b1+" b2="+b2);
    System.out.println("i1<<4="+(i1<<4)+" i1>>4="+(i1>>4)+
      " i1>>>4="+(i1>>>4));
    System.out.println("i2<<4="+(i2<<4)+" i2>>4="+(i2>>4)+
      " i2>>>4="+(i2>>>4));
  }
}
```

Operatorii de deplasare şi efectele lor sunt prezentate în tabela de mai jos, în care `a` şi `s` sunt operanzi care aparţin unor tipuri întregi.

perator	Expresie	Efect
<<	a<<s	deplasare la stânga cu s poziţii binare

>>	a>>s	deplasare la dreapta cu s poziţii binare, *cu conservarea semnului*
>>>	a>>>s	deplasare la dreapta *fără semn*, cu s poziţii binare

Deplasarea la stânga cu s poziţii este echivalenta cu înmulţirea numărului cu 2^s. Dacă s este suficient de mare, poate avea loc o depăşire binară, la fel ca în cazul înmulţirii aritmetice (cu operatorul *).

Deplasarea la dreapta cu s poziţii cu operatorul >> este echivalenta cu împărţirea întreagă la 2^s.

Deplasarea biţilor la dreapta cu s pozitii cu operatorul >>> are *asupra numerelor pozitive* acelaşi efect ca cel al operatorului >>. În schimb, în cazul operanzilor negativi, în cazul operatorului >>> nu se mai conservă semnul, iar modulul numărului se modifică. Aceasta se întâmplă, întrucât pe poziţiile eliberate din partea stângă se înserează bitul 0.

Pentru a inţelege mai bine efectul operatorilor de deplasare, să urmărim ce se întâmplă în exemplul de mai sus la nivelul reprezentării interne a datelor (în binar).

Având în vedere că $1024=2^{10}$, reprezentările interne ale operanzilor i1 şi i2 sunt următoarele:
```
i1=00000000000000000000010000000000
i2=11111111111111111111110000000000
```
După aplicarea operatorului << (deplasare binară la stânga cu inserare de zerouri pe spaţiile libere din dreapta) obţinem:
- pentru operaţia `i1<<4:` `00000000000000000100000000000000`
- pentru operaţia `i2<<4:` `11111111111111111100000000000000`

ceeace reprezintă în zecimal numerele 2^{14}=16384 si, respectiv, -2^{14}=-16384.

După aplicarea operatorului >> (deplasare la dreapta cu conservarea semnului) se obţine:
- pentru operaţia `i1>>4:` `00000000000000000000000001000000`
- pentru operaţia `i2>>4:` `11111111111111111111111111000000`

ceeace reprezintă în sistemul zecimal numerele 2^6=64 şi, respectiv, -2^6=-64. Remarcăm că în cazul operandului negativ i2, la deplasarea la dreapta s-a înserat pe poziţiile libere bitul 1 pentru a se conserva semnul.

După aplicarea operatorului >>> (deplasare la dreapta fără conservarea semnului) se obţine:
- pentru operaţia `i1>>>4:` `00000000000000000000000001000000`
- pentru operaţia `i2>>>4:` `00001111111111111111111111000000`

Pentru operandul pozitiv i1 s-a obţinut un rezultat identic cu cel precedent. În schimb, în cazul operandului negativ i2, pe poziţiile eliberate din partea stângă s-a înserat bitul 0, ceeace a dus la schimbarea semnului şi a valorii rezultatului faţă de cazul deplăsarii cu conservarea semnului.

Operaţii logice pe biţi

Operatorul *unar* ~ exprimă negaţia logică, deci înlocuirea lui 0 cu 1 şi invers.
Operatorii *binari* &, | si ^ acţionează la fel ca în cazul tipului boolean, numai că se aplică la nivel de bit.

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

```
/* Testarea operatiilor logice pe biti */

class LogicaBiti {
  public static void main(String args[]) {
    byte b1=17, b2=-95, b3;
    b3=(byte)(b1&b2); // este necesara conversie de la int la byte
    System.out.println("~b1="+(~b1)+" b1&b2="+b3+" b1|b2="+(b1|b2)+
      " b1^b2="+(b1^b2));
  }
}
```

b1: 00010001 echivalent cu 17
b2: 10100001 echivalent cu -95
~b1: 11101110 echivalent cu -18
b1&b2: 00000001 echivalent cu 1
b1|b2: 10110001 echivalent cu -79
b1^b2: 10110000 echibalent cu -80

Bineînţeles că toate aceste rezultate ar trebui exprimate pe 32 biţi, prin prelungire la stânga cu cate 18 cifre de 0 sau de 1 după caz, astfel încât să se conserve semnul.

```
/* Testarea operatorilor de atribuire compusa */

class AtribComp {
  public static void main(String args[]) {
    int m=176, b=-15, c=16, d=-28;
    System.out.println(m+" "+(m+=b)+" "+m);
    System.out.println((m/=d)+" "+m);
    System.out.println(b+" "+(b<<=2)+" "+b);
    System.out.println(c+" "+(c|=b)+" "+c);
  }
}
```

Prima linie afişată conţine:
- valoarea variabilei `m` *înainte* de evaluarea expresiei `m+=b`;
- valoarea expresiei `m+=b`; ea este egală cu `m+b` adică cu `176-15`;
- valoarea variabilei `m` *după* calcularea acestei expresii.

Tipuri de date în virgulă mobilă

Tipul	Lungimea	Intervalul de valori
`float`	4 octeti (32 biti)	[-3.402347e+38f, ... 3.402347e38f]
`double`	8 octeti (64 biti)	[-1.7976931348623157e+308, ... 1.7976931348623157e308]

*Exemple de **literali** în virgulă mobilă*
a/ literali de tip float fără exponent: `123.7f, -1876.53f`;
b/ literali de tip float cu exponent:
`7.28765e12f` (echivalent cu 7.28765×10^{12})

`5.0754286e-8f` (echivalent cu 5.075428×10^{-8})

`-3.9104e3f` (echivalent cu -3.9104×10^{3})

c/ literali de tip double fără exponent: `154236789.20973, -3401.76398653;`

d/ literali de tip double cu exponent:

`2935.87628091e-12`

`-1.20937543872E23`

`12e23` (echivalent cu 12.0×10^{23})

```
/* testarea operatiilor aritmetice cu numere in virgula mobila */

class VirgulaMobila {
  public static void main(String args[]) {
    double a=12.7865, b=-158.07, c=0.0, d=-0.0, inf1, inf2, n;
    float p=3476.15f, q=0.00237621f, r=0.0f, s=-0.0f;
    System.out.println("a/b="+(a/b)+" b/a="+(b/a)+" b%a="+(b%a));
    System.out.println("a/c="+(a/c)+" b/c="+(b/c));
    System.out.println("a/d="+(a/d)+" b/d="+(b/d));
    System.out.println("c/d="+(c/d)+" d/c="+(d/c));
    System.out.println("p/q="+(p/q)+" q/p="+(q/p)+" p%q="+(p%q));
    System.out.println("p/r="+(p/r)+" p/s="+(p/s));
    System.out.println("a/q="+(a/q)+" p/b="+(p/b));
    inf1=a/c; inf2=a/d; n=c/d;
    System.out.println("a/inf1="+(a/inf1)+" a/inf2="+(a/inf2));
    System.out.println("inf1/inf2="+(inf1/inf2)+" inf2/inf1="+
      (inf2/inf1));
    System.out.println("a*inf1="+(a*inf1)+" c*inf1="+(c*inf1));
  }
}
```

Din aceste rezultate constatăm că:

- operaţiile cu infiniţi şi nedeterminări respectă regulile cunoscute din matematică (împărţirea la zero dă rezultat infinit, iar 0/0, infinit impartit la infinit şi zero înmulţit cu infinit sunt nedeterminări);
- dacă cel puţin un operand este în dublă precizie (`double`), rezultatul este în dublă precizie; dacă ambii operanzi sunt în simplă precizie (`float`), rezultatul este `float`;
- se verifică egalitatea `[b/a]*a+(b%a)=b`, in care `[b/a]` este partea intreagă a câtului `b/a`, iar `b%a` este restul împărţirii întregi a lui `b` la `a`; de exemplu, `(-12)*12.7865-4.632=-158.07`.

Tipul `char`

```
/* Operatii cu date de tip char */

class TestChar {
  public static void main(String args[]) {
    char c1='A', c2='a', c3='b', c4='1', c5='*', c6='\n';
    System.out.println(c1+" "+c2+" "+c6+" "+c4+" "+c5+" "+c3);
    System.out.println((int)c1+" "+(int)c2+" "+(int)c6+" "+
      (int)c4+" "+(int)c5);
    System.out.println(c1+c2);
    System.out.println(c1*c2);
    System.out.println((char)107);
    System.out.println('2'+" "+(int)'2'+" "+'3'+" "+(int)'3'+
      " "+('2'+'3'));
  }
```

```
}
```

După afişarea caracterelor A şi a s-a transmis către dispozitivul de afişare caracterul `'\n'`, care a fost interpretat drept comanda de trecere la linie nouă. După aceasta au fost afşsate caracterele 1, * şi b. Pe linia următoare sunt afişate caracterele deja afişate anterior, convertite în numere întregi. Se observă că în Unicode caracterul 'A' este reprezentat prin numarul 65, caracterul 'a' prin numarul 97, caracterul '\n' prin 10 etc.

Pe următoarele două linii sunt afişate rezultatele operaţiilor `'c1'+'c2'` si `'c1'*'c2'`, obţinându-se respectiv valorile numerice 162 si 6305. Se observă că, întrucât caracterele au fost folosite ca operanzi ai unor operatori aritmetici, rezultatele obţinute sunt numerice (de tip `int`). Pe linia următoare se afişează numărul 107 convertit din `int` în `char` şi interpretat deci drept 'k'; Pe ultima linie putem observa deosebirea dintre forma externă a caracterului (literalul de tip `char`) şi codificarea lui interna. Literalul de tip char `'2'` este *caracterul 2* si nu numarul 2. În consecinţă, acest caracter nu se reprezinta intern prin numarul întreg 2, ci prin numărul întreg 50. În mod corespunzator, caracterul `'3'` se reprezinta intern prin numarul 51. Suma codurilor numerice ale celor doua caractere este, evident, numărul 101.

Exerciţii:

1. Creaţi un proiect Java application şi scrieţi în clasa principală după metoda main următorul cod:

```
public static int recursiv(int nr)
{
    // verifica valoarea parametrului
    if(nr<=0)
        return nr;
    else
    {
        // decrementeaza valoarea si apeleaza din nou functia
        nr--;
        System.out.println(nr);
        return(recursiv(nr));
    }
}
```

În funcţia main apelaţi funcţia creată:

```
int rezultat = recursiv(10);
System.out.println("rezultat" + rezultat);
```

2. Sortarea prin selecţie. Creează un nou program şi introdu următorul cod în metoda main.

//array cu date nesortate

```
int [] myData = {3, 1, 9, 5, 7};
```

//parcurge array-ul intr-o bucla for

```
for(int pos=0; pos < myData.length-1; pos++)
{
        /* gaseste si retine pozitia
         * celui mai mic element
         * incepe de la pos si cauta in restul array-ului
         * (pos indica inceputul sectiunii nesortate)
         */
    int minIndex = pos;

        // parcurge sectiunea nesortata
    for(int next=pos+1; next< myData.length; next++)
    {
        if(myData[minIndex]>myData[next])
            minIndex=next;
    }

        //inverseaza pozitia elementelor daca este necesar
    if(minIndex!=pos)
    {
            //retine elementul curent indicat de pos
        int temp = myData[pos];
            //copiaza cel mai mic element la pozitia pos
        myData[pos] = myData[minIndex];
            //copiaza elementul inlocuit la pozitia minIndex
        myData[minIndex] = temp;
    }

        // Folosind for afişaţi array-ul sortat !
    .......
}
```

3. Sortarea prin interclasare. O metoda mai eficienta de sortare a continutului unui array este sortarea prin interclasare, care utilizeaza un algoritm recursiv. Aceasta se bazeaza pe ideea ca unificarea a doua array-uri deja sortate este o problema mai simpla decat ordonarea tuturor datelor dintr-un array nesortat. Deci, algoritmul sorteaza un array prin divizarea sa continua in doua parti si sortarea fiecareia, unificand rezultatele intr-un array sortat.

```
/* Functia mergeSort sorteaza datele dintr-un array
* -parametrii transmisi sunt array-ul cu datele ce trebuiesc sortate,
*  un array temporar (temp) utilizat la calcule
*  si doua valori intregi care indica inceputul si sfarsitul
*  sectiunii din array ce trebuie ordonata
*/
public static void mergeSort(int[] dataArray, int[] tempArray, int start, int
end)
{
    // daca start = end, array-ul a fost sortat
    if(start==end)
        return;
    // altfel imparte array-ul in 2 sectiuni
    else
    {
        //calculeaza centrul array-ului
        int center = (start+end)/2;

        //apeleaza mergeSort pentru prima jumatate
        mergeSort(dataArray, tempArray, start, center);

        //apeleaza mergeSort pentru cea de-a doua jumatate
        mergeSort(dataArray, tempArray, center+1, end);

        //uneste jumatatile intr-un singur array
        int firstIndex = start;
        int secondIndex = center+1;
        int thirdIndex = start;

        //parcurge ambele parti
        while(thirdIndex<=end)
        {
                /* parcurge partile
                 * inserand cel mai mic element
                 * in array-ul temp
                 */
            if(secondIndex>end ||
            (firstIndex<=center &&
            dataArray[firstIndex]<=dataArray[secondIndex]))
            {
                tempArray[thirdIndex]=dataArray[firstIndex];
                thirdIndex++;
                firstIndex++;
            }
            else
            {
                tempArray[thirdIndex]=dataArray[secondIndex];
                thirdIndex++;
                secondIndex++;
            }
        }

        //copiaza continutul array-ului temp in array-ul principal
        for(int i=start; i<=end; i++)
            dataArray[i]=tempArray[i];
    }
```

Acum, apeleaza noua metoda din main:

```
//array-ul ce trebuie ordonat
int[] someData = {4, 3, 7, 6, 2, 8, 5, 9, 1};
//array suplimentar (ca ajutor)
int[] extraArray = new int[someData.length];
//apeleaza functia mergesort
mergeSort(someData, extraArray, 0, someData.length-1);

//afiseaza rezultatul

…….
```