Cursul 5: Structuri de date I

Array

Un sir de date sau un array este o structura ce contine valori multiple de acelasi tip de data. Lungimea unui sir de date este stabilita la crearea array-ului si va fi fixa pe intreaga existenta a acestuia.

In figura de mai jos si anume figura 5.1 am reprezentat un array cu zece elemente si indexul incepand cu pozitia zero. Evident ultimul element poate fi accesat la pozitia data de lungimea sirului minus 1.

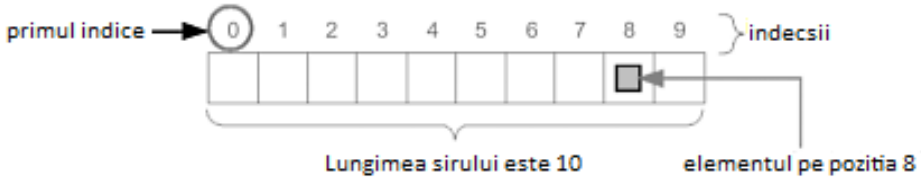


Figura 5.1 un sir cu 10 elemente.

Cum se declara un array?

Sa consideram faptul ca un sir este un obiect ce contine un set de elemente. Aceste elemente pot fi, fie de tip primitiv (int, float, char etc), fie de tip referinta (deriva din Object). Pentru a declara un sir de un date anumit tip se specifica acel tip de data urmat de paranteze:

```
int[] un_sir_de_date;
```

Odata ce un sir este declarat, el poate fi instantiat. Este necesar, ca inainte de folosirea sirului, acesta sa fie instantiat. Ca orice referinta, se va folosi operatorul new pentru instantiere, urmat de tipul de data si numarul de elemente al sirului:

```
int[] un_sir_de_date = new int[10];
```

Iata o functie de exemplu ce foloseste la instantierea unui sir de o anumita marime transmisa ca parametru:

```
int[] creeazaSir(int marime)
{
      return new int[marime];
}
```

Daca incercam sa cream un sir a carui lungime este negativa atunci va fi aruncata eroarea *NegativeArraySizeException*. Sirurile de lungime zero sunt insa corecte din punct de vedere sintactic.

Siruri de tipuri de date primitive

Atunci cand se creeaza un sir de tipuri de data primitv, sirul ca contine valorile acelor elemente. De exemplu, pentru un sir cu patru intregi prezentat in figura de mai jos avem si durata obiectului in memorie: in acest caz pe heap se aloca un numar fix si anume patru ori marimea tipului int.

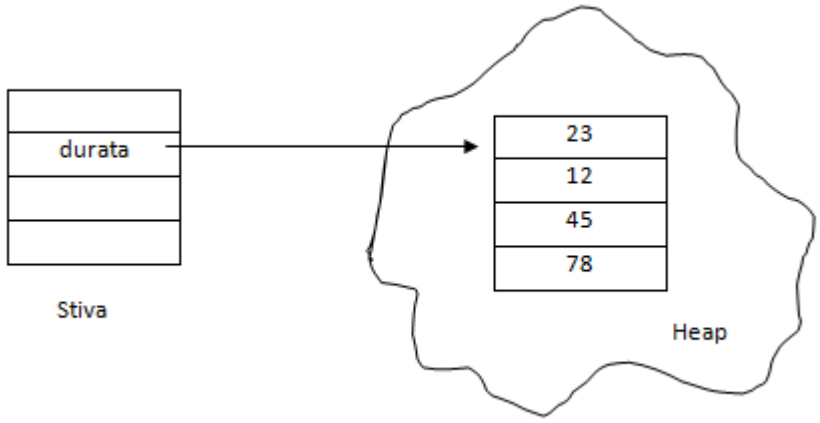


Figura 5.2 Stiva si heap in cazul unui sir de primitive

In figura 5.2 am reprezentat cum arata un sir in heap (memoria dinamica a programului).

Acestui sir ii pot fi atribuite aceste valori in doua moduri:

```
int[] sir = new sir[3];
sir[0] = 23;
sir[1] = 12;
sir[2] = 45;
sir[3] = 78;
```

Sau

```
int[] sir = {23,12,45,78};
```

Siruri de date de tip referinta

Spre deosebire de tipurile de data primitive, atunci cand cream un sir de obiecte, acestea nu sunt stocate intr-un masiv. Array-ul va stoca doar referintele catre obiectele propriu-zise, si initial fiecare referinta este *null*. Pentru a reprezenta instantierea unui sir de obiecte avem figura 5.3.

De aceasta data fiecare obiect se afla in alta zona de memorie (in heap), alta decat ce in care este declarat sirul. Elementele din care este compus sirul sunt referintele acestor obiecte, care sunt sau pot fi, de tipuri diferite.

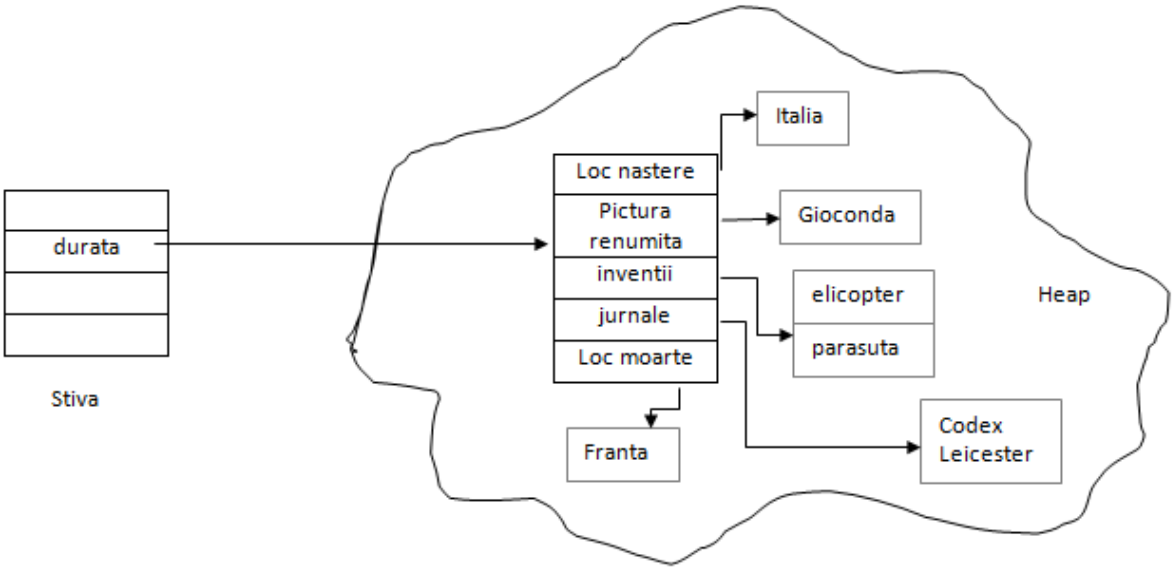


Figura 5.3 Reprezentare Stivei si heap-ului in cazul unui sir de obiecte

Inainte de a trece mai departe sa vedem un exemplu din fiecare sir de date mentionat mai sus. Pentru inceput vom alege sirul de primitive de tip int, si anume o simpla parcurgere a unui sir cu patru elemente si afisarea acestora:

```
public class ArrayDemo {
    public static void main(String[] args) {
        // declar un sir de int
          int[] sir = {23,12,45,78};

        // assign a value to each array element and print
        for (int i = 0; i < sir.length; i++) {
            System.out.print(sir[i] + " ");
        }
        System.out.println();
    }
}
```

Dupa cum vedem in acest exemplu toate elementele din sir sunt de tipul int.

Pe de alta parte, daca utilizam un sir de obiecte, putem intalni elemente de clase diferite. Totusi, in general un sir de referinte va contine elemente de aceeasi clasa specifica.

```
class Country
{
       String name;
       long population;
}
class Invention
{
       String name;
       String description;
```

```
}
class Painting
{
       String name;
       String technique;
}

public class ArrayDemo {
    public static void main(String[] args)
    {
        // declar un sir de int
          Object[] sir = new Object[4];
          sir[0] = new Country();
          ((Country)sir[0]).name = "Italia";
          ((Country)sir[0]).population = 6000000000001;
          sir[1] = new Painting();
          ((Painting)sir[1]).name = "Mona Lisa";
          ((Painting)sir[1]).technique ="Sfumatto";
          sir[2] = new Invention();
          ((Invention)sir[2]).name = "Elicopter";
          ((Invention)sir[2]).description =
                  "masina pentru zburat propulsata" +
                  " de un rotor orizontal";
          sir[3] = new Country();
          ((Country)sir[0]).name = "Franta";
          ((Country)sir[0]).population = 6500000000001;

    }
}
```

Siruri multidimensionale

Din moment ce un sir de date poate contine referinte catre alte obiecte, de ce nu ar contine referinte catre alte siruri? Atunci cand un sir se refera, sau contine alt sir avem de a face cu siruri multidimensionale. Iata un exemplu clasic de sir multidimensional:

```
int  matrix[][];
```

Pentru a instantia un astfel de sir se vor specifica, de obicei, ambele dimensiuni si anume:

```
int  matrix[][] = new int[3][2];
```

In acest caz avem un sir bidimensional, astfel: 3 siruri care la randul lor contin siruri de cate doua elemente de tip int.

Accesarea acestor elemente se poate face prin specificarea ambilor indici:

```
matrix[1][2] = 34;
```

Un exemplu simplu pentru a intelege mai bine lucrul cu matrici este urmatorul:

```
public class MatrixDemo
{
public static void main(String[] args)
    {
        int[][] matrix = new int[3][3];
        matrix[0][0]=1;
        matrix[1][1]=1;
        matrix[2][2]=1;
        for(int i=0;i<3;i++)
        {
            for(int j=0;j<3;j++)
                System.out.print(matrix[i][j]);
            System.out.println();
        }
    }
}
```

In acest exemplu se instantiaza o matrice de 3 x 3 elemente. Apoi se asigneaza valoarea 1 elementelor pe diagonala si mai departe se afiseaza valorile din matrice, parcurgand mai intai pe linii si apoi pe coloane.

Acest mod de folosirea a sirurilor multidimensionale nu este unic. Putem initializa subsiruri de diverse dimensiuni ca in exemplul de mai jos:

```
public class SirCrescator {
    public static void main(String args[])
    {
        float sir[][] = new float[4][];
        //orice subsir poate avea numar diferit de elmente
        //urmeaza instantierea fiecarui subsir
        sir[0] = new float[5];
        sir[1] = new float[]{2.3f,5,6.7f,11};
        sir[2] = new float[]{1f,4.2f,7f,4.1f,10f,9f};
        sir[3] = new float[20];
        //urmeaza partea de accesare a datelor
        //vom modifica doar primul subsir
        sir[0][1] = 1;
        sir[0][1] = 43;
        sir[0][3] = 89;
        //si ultimul subsir
        //deoarece celelalte contin deja valori
        sir[3][5] = 14;
        sir[3][10] = 5;
        sir[3][17] = 19;
```

```
            //mai intai parcurgem sirul "mare" pana
            //la lungimea acestuia (pe linii)
            for (int i = 0; i < sir.length; i++)
            {
                System.out.print("Element( "+ i +"  ): ");
                //aici parcurgem subsirurile
                for (int j = 0; j < sir[i].length; j++)
                {
                    System.out.print(" "+ sir[i][j]);
                }
                System.out.println();
            }
        }
    }
```

Afisare:

```
Element( 0  ):  0.0 43.0 0.0 89.0 0.0
Element( 1  ):  2.3 5.0 6.7 11.0
Element( 2  ):  1.0 4.2 7.0 4.1 10.0 9.0
Element( 3  ):  0.0 0.0 0.0 0.0 0.0 14.0 0
```

In acest exemplu avem un sir de 4 subsiruri de tip float. Acestea pot avea dimensiuni diferite. Primul subsir initializat si anume `sir[0]` va fi initializat ca un sir de *float* de 5 elemente. Vom avea acces l a elementele acestuia de la `sir[0][0]` pana la `sir[0][4]`. Mai departe urmatorul subsir este `sir[1], care` este automat instantiat si intializat cu valori astfel:

```
sir[1] = new float[]{2.3f,5,6.7f,11};
```

In acest caz subsirul de pe pozitia 1 are 4 elemente cu valorile dintre acolade.

Accesarea unui anumit element din acest sir bidimensional se face prin ambii indecsi si anume

```
sir[3][5] = 14;
```

adica elementul din subsirul 3 cu indexul 5 in acel subsir.

Apoi se poate parcurge ca in cazul matricii, insa, de data aceasta se tine cont de lungimea fiecarui subsir in parte:

```
for (int j = 0; j < sir[i].length; j++)
```

pentru ca de aceasta data fiecare subsir are o lungime variabila.

Aplicatiile acestor siruri multidimensionale provin din diversele necesitati matematice, cum ar fi de exemplu calculul triunghiului lui Pascal, sau reprezentarea valorilor din figuri geometrice, piramide, trapez, triunghi etc.

Initializarea sirurilor

Atunci cand se creeaza un array, sistemul Java va initializa fiecare element al sirului cu o valoare, zero daca este vorba de siruri numerice, false daca vorbim de siruri cu elemente de tip boolean, sau zero pentru cele care contin referinte.

Totusi, se pot specifica valorile direct la instantierea obiectelor:

```
String[] names = {"Ion","Vasile","Andrei","Radu"};
```

Acesta este un sir de *String*-uri care contine 4 elemente. Din acest moment el nu mai poate fi modificat, adica nu se pot sterge sau adauga elemente in sir. El va contine pe toata durata sa fix 4 elemente. Valorile din aceste elemente se pot, totusi, modifica.

Pentru siruri multidimensionale aceasta initializare poate avea loc astfel:

```
String[][] sir_nume = {{"Ion","Vasile","Adrian"},{"Andrei","Radu"}};
     System.out.println(sir_nume[0][2]);
```

In acest caz avem, doua subsiruri de *String*-uri, primul cu trei elemente si al doilea cu doua.

Se va afisa al treilea elment din primul subsir.

Mai mult se poate ca un sir sa fie trimis ca parametru si creeat la apelul metodei astfel:

```
public static void functie(String[][] sir)
{
    for(int i=0;i<sir.length;i++)
    {
        for(int j=0;j<sir[i].length;j++)
            System.out.print(" " +sir[i][j]);
        System.out.println();
    }
}
```

Aceasta este functia ce primeste un sir de siruri de *String*-uri si ce afiseaza fiecare element din acele subsiruri. Mai jos avem si apelul acestei functii

```
functie (new String[][]{{"Ion","Vasile","Adrian"},{"Andrei","Radu"}});
```

Inainte ca functia sa fie apelata, se creeaza un obiect de tip String[][] cu valorile dintre acolade, si acel obiect este transmis functiei ca parametru.

Copierea, clonarea sirurilor

Un sir poate fi copiat in alt sir, sau poate fi clonat in alt sir. Aceasta copiere se poate face in mai multe feluri. Daca trebuie sa crestem numarul de elemente ale unui sir, trebuie sa cream un sir nou cu numarul dorit de elemente si sa copiem element cu element vechiul sir in noul sir. Daca dorim totusi, sa lasam marimea intacta, si sa modificam valorile din sir, trebuie sa clonam acel sir.

Metoda *arraycopy()* a clasei *System* permite copierea elementelor dintr-un sir in altul. Sirul destinatie trebuie sa fie mai mare sau egal ca numar de elemente, cu sirul sursa. In caz contrar, se arunca o exceptie de tip *ArrayIndexOutOfBoundsException* la rulare.

Metoda *arraycopy()* are cinci parametrii si anume:

`arraycopy (Object sourceArray, int sourceOffset, Object destinationArray, int destinationOffset, int numberOfElementsToCopy).`

Mai jos sunt explicatiile pentru fiecare dintre parametri:

`sourceArray` sirul sursa din care se copiaza elementele

`sourceOffset` pozitia din sir de la care consideram elementele sursa

`destinationArray` sirul destinatie, in care se copiaza elementele
`destinationOffset` pozitia din destinatie de incepe copierea elementelor
`numberOfElementsToCopy` cate elemente vor fi copiate.
Mai jos este un exemplu de folosire a functiei *arraycopy*:

```
int[] sir_sursa = {23,12,46,8};
int[] sir_destinatie = new int[6];
System.arraycopy(sir_sursa,0, sir_destinatie, 0,sir_sursa.length);
```

Atentie, atunci cand copiati elemente dintr-un sir in altul, ele trebuie sa fie de acelasi tip de data, sau compatibile, altfel va fi aruncata o exceptie de tip *ArrayStoreException.*

Un exemplu de siruri incompatibile ar fi doua siruri unul de obiecte si unul de primitive.

Pentru a intelege mai bine cum se foloseste arraycopy avem clasa de mai jos:

```
public class copiere {
    public static void main(String args[])
    {
        int array1[] = {1, 2, 3, 4, 5};
        int array2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        System.out.println("Original size: " + array1.length);
        printArray(array1);
        System.out.println("New size: " + doubleArray(array1).length);
        printArray(doubleArray(array1));
        System.out.println("Original size: " + array2.length);
        printArray(array2);
        System.out.println("New size: " + doubleArray(array2).length);
        printArray(doubleArray(array2));
    }
    static int[] doubleArray(int original[])
    {
        int length = original.length;
        int newArray[] = new int[length*2];
        System.arraycopy(original, 0, newArray, 0, length);
        return newArray;
    }
    public static void printArray(int[] sir)
    {
        for(int i=0;i<sir.length;i++)
        {
            System.out.print(" " +sir[i]);
        }
        System.out.println();
    }
}
```

In acest exemplu, functia `doubleArray` apeleaza in interiorul ei, functia `System.arraycopy(original, 0, newArray, 0, length);` cu destinatia `newArray` un sir de doua ori mai mare decat originalul. Functia returneaza acest sir cu lungime dubla.

Functia `printArray` afiseaza elementele sirului transmis ca parametru.

In main se folosesc cele doua functii pentru a dubla numarul de elemente ale sirurilor originale si anume `array1` si `array2`.

Sirurile sunt imutabile

Daca declaram un sir fina, si static:

```
final static int array[] = {1, 2, 3, 4, 5};
```

nu suntem constransi sa nu putem modifica elementele lui. Ce nu putem insa modifica este insasi structura sirului, si anume numarul de elemente.

Adica nu putem avea in continuare o astfel de atribuire:

```
array = new int[] {6, 7, 8, 9};
```

Insa putem modifica orice element din sir: `array[2]=8;`

In general obiectele imutabile sunt cele a caror stare nu poate fi modificata. De exemplu pentru modificarea marimii unui sir, suntem fortati sa cream un alt obiect, adica sir, si sa copiem elementele din vechiul sir. Aceasta proprietate, imutabilitate are multe implicatii si avantaje despre care vom vorbi mai tarziu.

Cand efectuam o atribuire a unui sir catre alt sir, adica `int[] b = a;` cele doua siruri vor pointa catre acelasi obiect, aceeasi referinta.

Verificarea egalitatii a doua siruri

Sirurile sunt obiecte, deci respecta regulile privind egalitatea. Atunci cand se foloseste operatorul == pentru aceasta verificare se vor compara referintele.

```
int array1[] = {1, 2, 3, 4, 5};
int array2[] = array1;
System.out.println(array1==array2);
```

In acest caz rezultatul va fi true dar daca am fi avut:

```
int array1[] = {1, 2, 3, 4, 5};
int array2[] = {1, 2, 3, 4, 5};
System.out.println(array1==array2);
```

Verificarea corecta a *valorilor* dintr-un sir se face folosind *equals*. Aceasta metoda nu este cea suprascrisa din *Object* ci este din clasa *java.util.Array*. Astfel comparatia de mai sus devine:

```
int array1[] = {1, 2, 3, 4, 5};
int array3[] = {1, 2, 3, 4, 5};
System.out.println(java.util.Arrays.equals(array1,array3));
```

Aceasta clasa mai contine metode de sortare a elementelor unui sir, de umplere a sirurilor si de cautare binara.

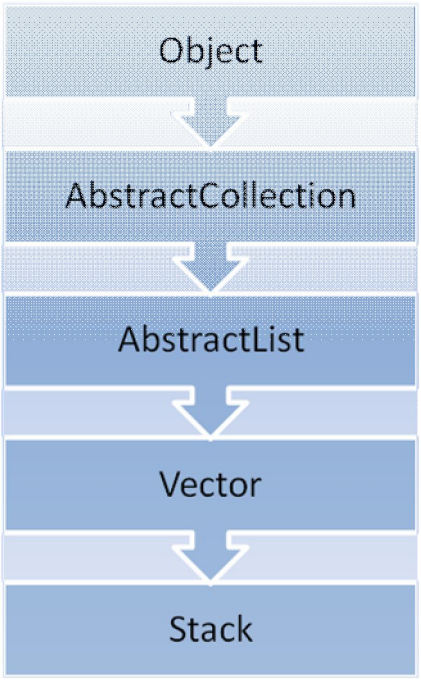
Sirurile sunt foarte potrivite cand stim apriori cate elemente ne trebuie pentru a rezolva o anumita problema. Dar daca nu stim? Putem crea siruri de lungime zero si sa folosim arraycopy pentru a redimensiona. Totusi aceasta implica crearea altor siruri, ocuparea de memorie in plus. Pentru aceasta Java ofera o alta clasa si anume *Vector*.

Clasa Vector

Clasa *Vector* face parte din pachetul java.util si are urmatoarea declaratie:

```
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

Astfel putem concluziona sirul de mosteniri, si legatura dintre Vector si Stack astfel:



Pe langa aceste clase pe care le mosteneste, *Vector* mai implementeaza si interfetele *Cloneable* si *Serialize*. Interfata Serialize permite ca orice obiect care o implementeaza sa poata fi serializat pentru a fi transmis catre diverse destinatii, sau deserializat, pentru a putea fi corect preluat.

Metodele de baza

Mai jos avem variabilele clasei Vector si descrierea pentru fiecare dintre ele:

Variabila	**Descriere**
capacityIncrement	Marimea pentru incrementarea capacitatii unui vector
elementCount	Numarul de elemente dintr-un vector.
elementData	sirul de date intern al vectorului.
modCount	mostenita din *AbstractList*: folosit de Iterator pentru a verifica modificari concurente.

Capacitatea unui vector reprezinta numarul maxim de elemente admis.

Urmeaza o lista cu metodele din clasa Vector:

Metoda	**Descriere**
add()	Adauga un element intr-un vector.
addAll()	Adauga o colectie de elemente intr-un vector.
addElement()	asemenea metodei add()
capacity()	Returneaza capacitatea adica marimea sirului intern.
clear()	sterge toate elementele unui vector.
clone()	Creeaza o clona a vectorului.
contains()	Verifica daca un vector contine un element.
containsAll()	Verifica daca un vector contine o colectie.
copyInto()	Copiaza elementele unui vector intr-un array.
elementAt()	Returneaza elementul de la pozitia specificata.
elements()	Returneaza un obiect al vectorului care permite vizitarea tuturor cheilor vectorului.
ensureCapacity()	Verifica si se asigura ca marimea buffer-ului intern sa fie de o anumita marime.
equals()	verifica egalitatea cu un obiect.
firstElement()	returneaza primul element.
get()	returneaza un element de la o anumita pozitie.
indexOf()	cauta prima aparitie a unui element in sir.
insertElementAt()	Insereaza un element in sir.
isEmpty()	verifica daca un vector este gol.
iterator()	returneaza un iterator, adica un obiect ce permite vizitarea elementelor din sir
lastElement()	Returneaza ultimul element din sir
lastIndexOf()	Cauta pozitia ultimului element din sir care este egal cu obiectul specificat
listIterator()	Returneaza un obiect care permite ca toate elementele sa fie vizitate secvential.
remove()	Sterge un anumit element din vector.
removeAll()	Sterge toate elementele specificate in colectia data ca parametru.
removeAllElements()	Sterge toate elementele din sir si seteaza marimea acestui cu zero.
set()	Schimba un element de la o anumita pozitie.
setElementAt()	Acelasi lucru ca si set.
setSize()	modifica marimea buffer-ului intern
size()	returneaza numarul de elemente din sir
subList()	returneaza o sectiune din sir.
toArray()	returneaza elementele vectorului ca array.
trimToSize()	„taie” o portiune din sir, astfel ca el sa ramana de marimea specificata.

Crearea Vectorilor

Se pot folosi patru constructori pentru a creea un Vector. Folosind unul din primii trei constructori, se poate creea un vector gol, cu o capacitate specificata. Cand spatiul devine insuficient, vectorul isi va dubla marimea.

```
public Vector()
public Vector(int initialCapacity)
public Vector(int initialCapacity, int capacityIncrement)
```

Motivul pentru care exista cei doi constructori cu parametru, este de a permite initializarea cu un numar de elemente a sirului, in cazul in care stim apriori de cate elemente este nevoie.

Crearea unui nou sir si copierea elementelor ocupa timp, de aceea este mai eficient sa redimensionam un array. Daca marimea vectorului specificata prin `initialCapacity` este negativa se arunca o exceptie de tip *IllegalArgumentException*.

Al patrulea constructor se refera la utilizarea unei colectii pentru initalizarea vectorului. Practic, se iau elementele din colectie si se construieste un array cu acestea. Vom aborda subiectul colectii in cele ce urmeaza.

```
public Vector(Collection c)
```

Vectorul nou creeat are cu 10% mai mult decat numarul de elemente din colectia *c*.

Cum copiem atunci rapid un sir clasic intr-un Vector?

Pentru aceasta, se poate transforma sirul intr-o colectie cu functia *Arrays.asList()* si apoi folosi constructorul de mai sus:

```
Vector v = new Vector(Arrays.asList(array));
```

Manipularea elementelor din vector

Pentru a adauga elemente avem mai multe „oferte” din partea Java, si a clasei Vector.

Adaugarea la sfarsit

Se apeleaza metoda *add* sau *addElement* dupa cum sunt declarate:

```
public boolean add(Object element)
public void addElement(Object element)
```

Pentru a demonstra adaugarea la sfarsit vom copia dintr-un sir de String-uri intr-un vector, unul cate unul adaugand la sfarsit:

```
import java.util.Vector;
public class InsertVector
{
	public static void main (String args[])
```

```
        {
                Vector v = new Vector();
                for (int i=0, n=args.length; i<n; i++)
                {
                        v.add(args[i]);
                }
        }
}
```

Adaugarea undeva in interior

Exista momente, cand dorim sa inseram elementele la anumite pozitii in sir, mutand celelalte elemente mai la dreapta. Exista doua functii care fac acest lucru:

```
public void add(int index, Object element)
public void insertElementAt(Object element, int index)
```

Motivul pentru care exista doua doua metode care fac acelasi lucru, este ca, clasa Vector implementeaza interfata List, deci este parte din Collection. Aceeasi eroare va fi aruncata daca indexul este negativ, si anume *IllegalArgumentException.*

Trebuie facuta diferenta intre *add* si *set*: ultima metoda doar modifica elementul de la pozitia specificata. Pentru a intelege mai bine adaugarea am luat un vector cu patru elemente.

Ce se intampla de exemplu cand inseram un element nou in sirul de mai jos:

primul	al doilea	al treilea	al patrulea

In urma inserarii unui element prin comanda:

```
vector.add(2,"nou");
```

se obtine:

primul	al doilea	nou	al treilea	al patrulea

Adaugarea unei colectii

Ultima metoda de a adauga elemente in vector este *addAll*:

```
public boolean addAll(Collection c)
public boolean addAll(int index, Collection c)
```

Practic se copiaza elementele dintr-un obiect de tip Collection in vector. Aceste elemente se pot adauga fie la sfarsitul vectorului, folosind prima metoda, fie se pot insera la incepand cu un anumit

index, folosind cea de a doua metoda. Este mai eficient sa adaugam toate elementele dintr-un bloc de elemente decat sa incercam sa le adaugam unul cate unul.

Daca indexul este invalid atunci se va furniza o exceptie de tip *IllegalArgumentException.*

Deoarece vectorii au fost ganditi sa lucreze cu obiecte, pentru a lucra cu primitive trebuie sa mai depunem un efort.

Vectori de tipuri primitive

Pentru a lucra cu valori de tip primitv trebuie sa convertim acestea in clase wrapper. Spre exemplu int are clasa wrapper Integer, sau float are clasa wrapper Float. Aceste specificatii sunt in cursul numarul 1. Exemplul de mai jos prezinta modul de lucru cu tipuri primitive:

```
import java.util.Vector;
public class PrimVector
{
	public static void main (String args[])
	{
		Vector v = new Vector();
		for (int i=1; i<=10; i++)
		{
			v.add(new Integer(i));
		}
	}
}
```

La preluarea valorilor din vector va trebui de asemenea sa transformam din Integer in int, deci din tip referinta in primitiv.

Afisarea vectorilor

Ca orice Object, si clasa vector implementeaza metoda *toString(),* ceea ce inseamna ca putem folosi direct obiectul de tip vector pentru afisare:

```
System.out.println(v);
```

unde *v* este vectorul din exemplul de mai sus. Aceasta comanda va afisa urmatorul sir:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Stergerea elementelor

Ca si in cazul adaugarii, exista multe feluri in care se pot sterge elemente din vectori, si vom discuta despre acestea in cele ce urmeaza.

Cel mai simplu mod de a sterge, este de a elimina toate elementele folosind:

```
public void clear()
public void removeAllElements()
```

Pentru a sterge un element de la o anumita pozitie se vor folosi metodele:

```
public Object remove(int index)
public void removeElementAt(int index)
```

Atata timp, cat indexul este mai mare ca zero si nu depaseste lungimea sirului, nu vom primi o exceptie de tip *ArrayIndexOutOfBoundsException*.

Atunci cand stergem un element, ramane un loc vid, si intregul subsir ce urmeaza elementului sters va fi deplasat la stanga:

De exemplu daca rulam comanda

```
vector.remove(2);
```

rezultatul este urmatorul

Daca nu se cunoaste pozitia, indexul elementul in sir, dar se cunoaste valoarea acestuia, atunci se va folosi una din metodele:

```
public boolean remove(Object element)
public boolean removeElement(Object element)
```

Aceste metode cauta in sir, primul obiect care este egal cu obiectul dat ca parametru. Pentru comparatia obiectelor (cele din sir si parametru) se foloseste metoda *equals*.

Pe langa aceste metode de stergere, mai exista una care permite eliminarea unei colectii de elemente:

```
public boolean removeAll(Collection c)
```

Metoda va prelua ca parametru colectia *c* si va sterge fiecare element din vector care se gaseste in colectia *c.*

Alta metoda de a sterge, mai precisa, este *removeRange* si permite stergerea elementelor din subvectorul delimitat de doua margini: *fromIndex* si *toIndex*:

```
protected void removeRange(int fromIndex, int toIndex)
```

Marimea unui vector

Pentru a dimensiona sau a afla marimea unui sir exista metodele:

```
public int size()
public void setSize(int newSize)
```

Prima metoda returneaza marimea sirului, adica numarul de elemente, iar a doua permite setarea marimii vectorului la un numar de elemente specificat ca parametru. Vom vedea ca marimea unui sir poate fi diferita de capacitatea acestuia.

Capacitatea vectorului

Capacitatea reprezinta numarul de elemente pe care un vector le poate contine. Aceasta capacitate se poate modifica prin ajustarea marimii vectorului, insa este de preferat sa folosim aceasta metoda de cat mai putine ori, pe cat este posibil. Metoda care returneaza capacitatea este:

```
public int capacity()
```

Daca marimea vectorului trebuie sa fie mai mare decat capacitatea lui, vectorul va creste dinamic. Atunci cand trebuie sa adaugam elemente in vector, este bine sa verificam daca avem capacitatea necesara. Pentru aceasta exista metoda *ensureCapacity* care tocmai asta face:

```
public void ensureCapacity(int minCapacity)
```

Daca deja capacitatea este suficient de mare, nu se intampla mai nimic. Daca in schimb, capacitatea este mai mica decat numarul de elemente ce se va obtine in urma adaugarii, vectorul isi va dubla marimea. Se poate insa stabili si numarul de elemente care va fi atribuit marimii vectorului in cazul apelarii metodei *ensureCapacity*.

Cautarea in vector

Aceasta cautare si redare a unui element se poate face in mai multe feluri.

Preluarea dupa index

Pentru a prelua pe baza indexului un element dintr-un vector exista doua metode:

```
public Object get(int index)
public Object elementAt(int index)
```

De exemplu , folosind functia *get* putem prelua un element de pe orice pozitie:

```
Vector v = new Vector();
v.add("Hello");
v.add("World");
String s = (String)v.get(1);
```

Preluare dupa pozitie in sir

Pentru aceasta exista doua functii pentru returnarea primului si ultimului element din sir:

```
public Object firstElement()
public Object lastElement()
```

Enumerarea elementelor

Pentru ca si cautarea sa aiba loc cat mai eficient, si anume sa parcurgem tot sirul pe indecsi pentru a gasi un element, exista un mod mai flexibil, si anume folosirea unui obiect de tip *Enumeration*.

Obiectele de tip Enumeration au doua functii *nextElement()* si *hasMoreElement()* si sunt ideale pentru iterari prin colectii.

Pentru a transforma un vector intr-un obiect de tip Enumeration exista metoda:

```
public Enumeration elements()
```

Mai jos avem un exemplu de folosire a acestui tip de obiect si anume *Enumeration*:

```
Vector v = . . .
Enumeration e = v.elements();
while (e.hasMoreElements())
{
      process(e.nextElement());
}
```

In acest while se va parcurge atata timp cat functia `e.hasMoreElements()` va returna *true*. `process` este o functie care va lucra cu elementul din sir, si anume fiecare element pana la epuizarea marimii sirului.

Pentru a intelege si mai bine lucrul cu *Enumeration* avem exemplul de mai jos:

```
import java.util.Enumeration;
import java.util.Vector;
class enumerare
{
    public static void main(String args[])
    {
        Vector v=new Vector(10,10);
        for(int i=0;i<20;i++)
            v.addElement(new Integer(i));
        System.out.println("Vector in ordinea originala");
        //se initializeaza obiectul e cu v.elements
        //astfel ca putem parcurge enumeratorul e
        for(Enumeration e=v.elements();e.hasMoreElements();)
        {
            //nu exista conditie de incrementare in for
            //dar e.nextElmement() se deplaseaza
```

```
            //la urmatorul element
            System.out.print(e.nextElement()+" ");
        }
        System.out.println();
         //afisarea vectorului original in ordine inversa
         System.out.println("\nVector in ordinea inversa");
         for(int i=v.size()-1;i>=0;i--)
             System.out.print(v.elementAt(i)+" ");

    }
}
```

Gasirea elementelor intr-un sir

Mai intai exista functia *contains* care verifica daca obiectul dat ca parametru exista in sir:

```
public boolean contains(Object elment)
```

Exista doua functii care se ocupa cu gasirea unui obiect in sir si anume:

```
public int indexOf(Object element)
public int indexOf(Object element, int index)
```

Incepand cu primul element din sir (in cazul primei functii), sau cu elementul de la pozitia `index` (in cazul celei de-a doua), se cauta obiectul `element` in sir si se returneaza indexul unde a fost gasit prima data.

Aproape acelasi lucru este realizat de cele doua functii de mai jos, cu precizarea ca, cautarea incepe de la sfarsit catre primul element din sir.

```
public int lastIndexOf(Object element)
public int lastIndexOf(Object element, int index)
```

Evident pentru a compara doua elemente si anume parametrul si obiectele din sir se foloseste metoda *equals()*. Mai jos avem un exemplu pentru cautarea in vectori:

```
public class finding {
    static String members[] =
    {"Andrei", "Ion", "Radu",
     "Mircea", "David","Radu", "Gheorghe"};
    public static void main (String args[])
    {
        Vector v = new Vector();
        for (int i=0, n=members.length; i<n; i++)
        {
            v.add(members[i]);
        }
```

```
            System.out.println(v);
            System.out.println("Contine UnNume?: " +
            v.contains("UnNume"));
            System.out.println("Contains Mircea?: " +
            v.contains("Mircea"));
            System.out.println("Unde e Radu?: " +
            v.indexOf("Radu"));
            System.out.println("Unde e DAvid?: " +
            v.indexOf("DAvid"));
            System.out.println("Unde e Radu de la sfarsit?: " +
            v.lastIndexOf("Radu"));
        }
    }
```

Rezultatul rularii acestui program este:

```
Contine UnNume?: false
Contains Mircea?: true
Unde e Radu?: 2
Unde e DAvid?: -1
Unde e Radu de la sfarsit?: 5
```

Copierea unui vectorilor

Exista mai multe feluri de copiere a unui vector. Se poate clona ca orice obiect, sau se poate copia. Mai intai sa vedem cum se cloneaza:

```
Vector v1 = . . .;
Vector v2 = (Vector)v1.clone();
```

In acest moment avem doi vectori diferiti insa cu aceleasi elmente.
Alt mod de a copia este cel folosind metoda *toArray* si apoi *copyInto*:

```
public Object[] toArray()
public Object[] toArray(Object[] a)
public void copyInto(Object[] anArray)
```

Adica, vom converti un vector in array si apoi copiere in alt vector prin *copyInto.*
Mai jos avem transformarea unui vector in sir:

```
Vector v = . . .;
String array[] = new String[0];
array = (String[])v.toArray(array);
```

dupa ce apeleaza metoda *toArray()*, sirul este redimensionat cu numarul de elemente din *v*.
Apoi se poate transforma din sir in vector astfel:

```
Vector v = . . .;
String array[] = new String[v.size()];
array = (String[])v.toArray(array);
```

Cel mai indicat mod de a copia este folosind metoda *copyInto:*

```
Vector v = . . .;
String array[] = new String[v.size()];
v.copyInto(array);
```

Pe langa vectori putem folosi structuri dedicate de date cum ar fi stiva sau coada, pe care le vom analiza in cele ce urmeaza.

Stiva

O stiva este o structura de tip LIFO, ultimul intrat primul iesit. Mai intai sa vedem cum functioneaza o stiva. Exista doua functii de introducere in stiva *push* si *pop* de eliminare a unui element din stiva. Introducerea unui element se va face „peste ultimul" element din stiva iar eliminarea va fi a ultimului element din stiva:

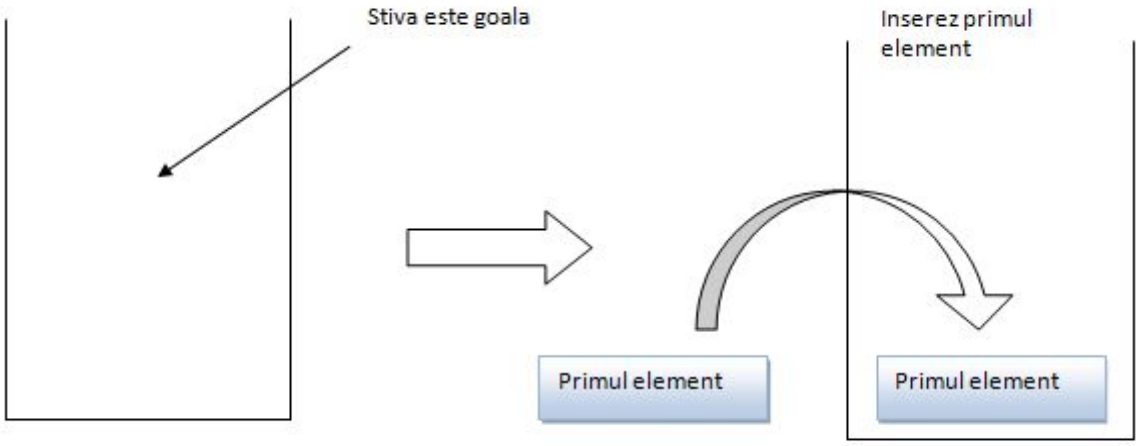


Figura 5.4 Stiva este goala, se introduce un elment folosind functia *push.*

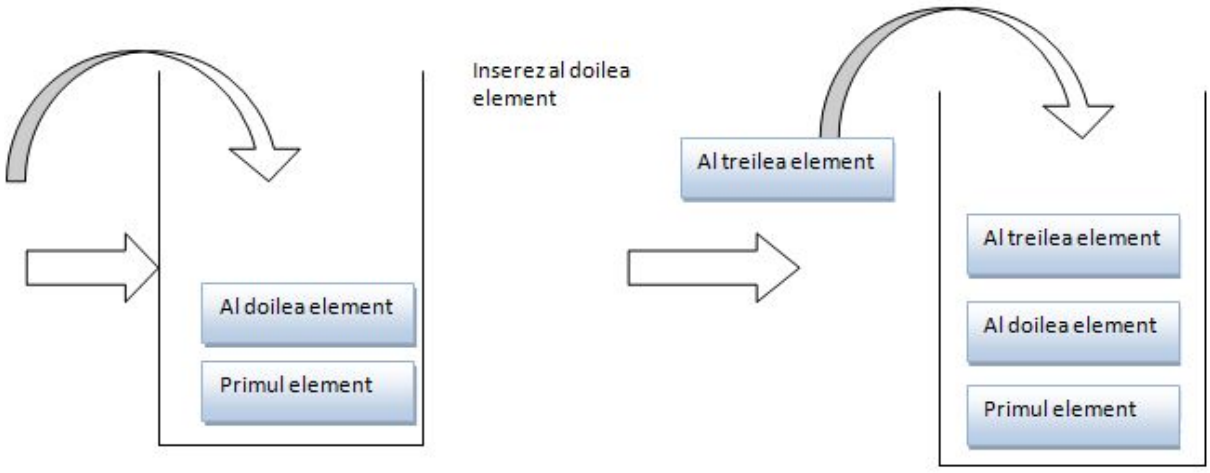


Figura 5.5 Stiva contine un element, se introduce al doilea element folosind functia *push*. Apoi se introduce al treilea element.

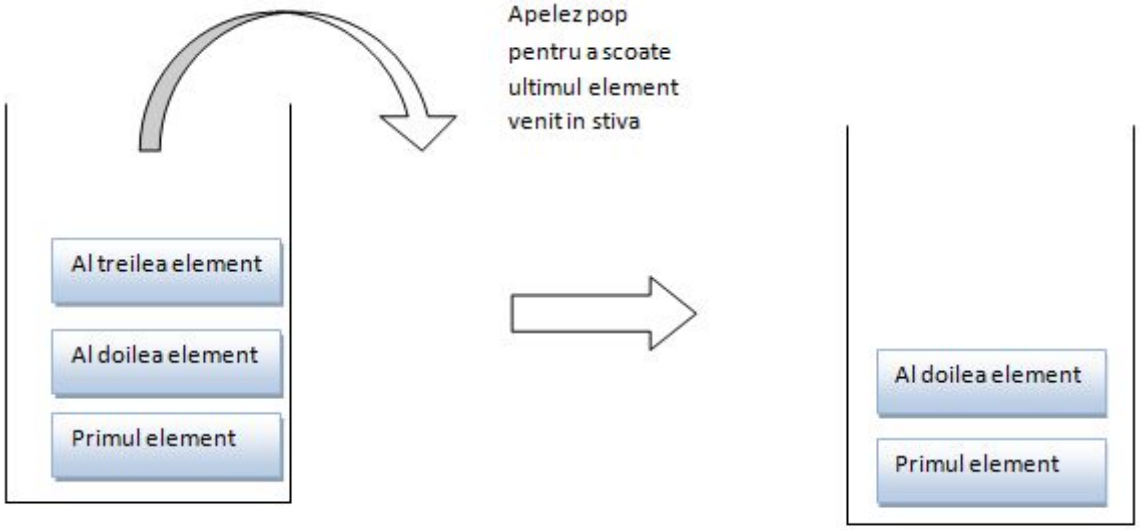


Figura 5.6 Se scoate elementul din varf cu functia *pop*, stiva va contine doua elemente.

Din cele trei figuri se poate intelege functionarea unei stive. Evident se pot scoate si celelalte elemente, adica sa revenim la starea initiala si anume stiva goala. Daca se incearca introducerea unui element atunci cand deja stiva este plina, va apare o eroare care sa semnaleze acest lucru. Evident, incercarea de a scoate dintr-o stiva goala duce la o eroare.

Clasa Stack

Clasa Stack in Java, este clasa copil a clasei Vector. Totusi exista cateva operatiuni specifice clasei *Stack* si anume:

Adaugarea elementelor

Aceasta operatiune se face prin metoda *push()*:

```
public Object push(Object element)
```

Aceasta va adauga un element la sfarsitul sirului din Vector.

Stergerea unui element

Aceasta operatiune se va face prin apelul functiei *pop()*:

```
public Object pop()
```

Aceasta va prelua primul element din stiva, il va sterge din stiva, si va returna valoarea lui. Daca stiva este goala vom primi o eroare de tipul *EmptyStackException*.

Pentru a verifica daca o stiva este goala sau nu avem functia:

```
public boolean empty()
```

pentru a prelua elementul din varful stivei exista functia *peek()*:

```
public Object peek()
```

To search an element in a stack we have the following function:

```
public int search(Object element)
```

Spre deosebire de lucrul cu vectori sau siruri, acest mecanism al stivei functioneaza astfel:

Elementul din varf este pe pozitia 1, pe pozitia 2 fiind elementul de dedesubt si asa mai departe. Daca nu este gasit obiectul cautat atunci o eroare corespunzatoare este returnata. Un exemplu pentru a ilustra mai bine folosirea stivelor este dat mai jos:

```
import java.util.Stack;
public class Stiva
{
    public static void main (String args[])
    {
        Stack s = new Stack();
        s.push("Primul element");
```

```
            s.push("Al doilea element");
            s.push("Al treilea element");
            System.out.println("Next: " + s.peek());
            s.push("Al patrulea element");
            System.out.println(s.pop());
            s.push("Al cincilea element");
            s.push("Al saselea element");
            //afisarea stivei
            System.out.println(s);
            // Gasim al doilea element
            int count = s.search("Al doilea element");
            // scot din stiva tot pana la el
            while (count != -1 && count > 1)
            {
                s.pop();
                count--;
            }
            // scot din stiva pe el si il afisez
            System.out.println(s.pop());
            //este stiva goala?
            System.out.println(s.empty());
            System.out.println(s);
        }
    }
```

In acest exemplu se vor introduce sase elemente cu scoaterea celui de al patrulea:

```
            s.push("Al patrulea element");
            System.out.println(s.pop());
```

practic imediat dupa introducere. In continuare se cauta pozitia celui de-al doilea element in stiva, se scoate pe rand de la varf, toate elementele pana la el si apoi se scoate si el:

```
    Next: Al treilea element
    Al patrulea element
    [Primul element, Al doilea element, Al treilea element, Al cincilea
element, Al saselea element]
    Al doilea element
    false
    [Primul element]
```

Enumerator

Acest mecanism este unul eficient, si comporta doar doua metode: *hasMoreElements()* ce verifica daca mai exista elemente in enumeratie si returneaza un boolean. Cealalta metoda *nextElement()*, returneaza urmatorul element din colectie ca si Object. Daca nu mai sunt elemente si totusi este apelata metoda *nextElement()*, atunci este aruncata exceptia *NoSuchElementException*.

Pentru a parcurge orice structura de acest tip, pasii sunt:

```
Enumeration enum = . . .;

while (enum.hasMoreElements()) {
Object o = enum.nextElement();
processObject(o);
}
```

Daca folosim for in loc de while:

```
for (Enumeration enum = . . .; enum.hasMoreElements(); )
{
      Object o = enum.nextElement();
      processObject(o);
}
```

Intrebarea este, de unde gasim aceste tipuri de enumerare?

Un raspuns ar fi din colectii sau din vectori dupa cum am vazut mai sus. Alt raspuns este ca putem creea propriile noastre enumerari. Pentru aceasta clasa care reprezinta o enumerare va trebui sa implementeze o interfata si anume *Enumeration*. In exemplul de mai jos clasa care realizeaza acest lucru este ArrayEnumeration:

```
import java.util.*;

 class ArrayEnumeration implements Enumeration
 {
    private final int size;
    private int cursor;
    private final Object array;
    public ArrayEnumeration(Object obj)
    {
        Class type = obj.getClass();
        if (!type.isArray())
        {
            throw new IllegalArgumentException("Invalid type: " +
type);
        }
        //o metoda care preia lungimea unui sir
        //nu intram in detalii deocamdata
        size = java.lang.reflect.Array.getLength(obj);
        array = obj;
    }

    public boolean hasMoreElements()
    {
        return (cursor < size);
    }
      //preluam elementul de pe pozitia
      //data de cursor si marim cursorul
    public Object nextElement()
```

```
    {
        return java.lang.reflect.Array.get(array, cursor++);
    }
 }

class Enumerare
{
    public static void main(String args[])
    {
        Enumeration enumerare = new ArrayEnumeration(args);
            //parcurg sirul de argumente
        while (enumerare.hasMoreElements())
        {
            System.out.println(enumerare.nextElement());
        }
        Object obj = new int[] {2, 3, 5, 8, 13, 21};
        enumerare = new ArrayEnumeration(obj);
            //parcurg un sir de obiecte
        while (enumerare.hasMoreElements())
        {
            System.out.println(enumerare.nextElement());
        }
        try
        {
            //incerc sa instantiez o enumerare
            //ce are la baza un obiect ce nu e de tip Array
            enumerare = new ArrayEnumeration(ArrayEnumeration.class);
        }
        catch (IllegalArgumentException e)
        {
            System.out.println("Oops: " + e.getMessage());
        }
    }
}
```

O clasa utila care implementeaza aceasta interfata si anume Enumeration este StringTokenizer. Scopul aceste clase este de a imparti un String in diferite elemente, iar delimitarea se face pe baza unui caracter anume (de obicei spatiu sau virgula).

Astfel poate fi parcurs acest sir de siruri obtinut:

```
StringTokenizer tokenizer = new StringTokenizer("This is a test");
while (tokenizer.hasMoreElements())
{
      Object o = tokenizer.nextElement();
      System.out.println(o);
}
```

Pe langa metodele `hasMoreElements` si `nextElement` pe care le mosteneste din *Enumeration*, clasa *StringTokenizer* mai are si metodele `hasMoreTokens` si `nextToken` care fac cam acelasi lucru, in plus expun in loc de Object, direct String-uri.

```
StringTokenizer tokenizer = new StringTokenizer("This is a test");
while (tokenizer.hasMoreTokens())
{
      String s = tokenizer.nextToken();
      System.out.println(s);
}
```

IO: scanare si formatare

Scanner

Scanarea, sau preluarea de informatii dintr-o sursa poate fi facuta mai elegant folosind clasa Scanner. Un obiect Scanner fragmenteaza intrarea in subelemente numite *token*, pe baza unor delimitator. Un exemplu de a citi numere din *System.in*:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

Acest obiect poate folosi si alti delimitatori diferiti de spatiu. Acest exemplu citeste elementele dintr-un sir separate prin *linia*:

```
import java.io.*;
import java.util.Scanner;

public class ScanTest {
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        try {
            InputStream input = new FileInputStream("test.txt");
            s = new Scanner(input);
            s.useDelimiter("linia");
            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Mai mult se pot folosi expresii regulate pentru a marca delimitatorii:

```
public class ScanTest {
    public static void main(String[] args) throws IOException {
       String input = "1 3,4 2,4";
     Scanner s = new Scanner(input).useDelimiter("[,\\s]");
     System.out.println(s.nextInt());
     System.out.println(s.nextInt());
     System.out.println(s.next());
     System.out.println(s.next());
     s.close();
    }
}
```

Formatarea

Obiectele stream care implementeaza formatarea sunt fie *PrintWriter* fie *PrintStream.*
Pentru afisarea datelor exista doua niveluri de formatare:

- print si println – modul standard
- format – formateaza orice numar pe baza unui string

Prima metoda este cunoscuta iar valorile afisate nu sunt formatate:

```
public class FormatSample {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is " + r + ".");
    }
}
```

Aceasta produce o afisare bruta a datelor:

```
The square root of 2 is 1.4142135623730951.
The square root of 5 is 2.23606797749979.
```

Metoda *format* permite formatarea argumentelor pe baza unui pattern. Acest pattern este un string de formatare alcatuit din specificatori de formatare. Mai jos avem un prim exemplu de astfel de formatare:

```
public class Root {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

Rezultatul este unul mult mai simplu si la fel de corect:

```
The square root of 2 is 1.414214.
```

Toti specificatorii de format incep cu % si se termina cu unul sau doua caractere de conversie ce specifica tipul de formatare. Iata cateva conversii sunt:

- d – formateaza o valoare intreaga ca decimal
- f – formateaza o valoare floating point ca decimal
- n – afiseaza terminare linie
- x – formateaza intreg ca hexazecimal
- s – formateaza orice valoare ca string
- tB formateaza un intreg ca nume de luna

Iata un exemplu mai complex despre formatarea unui numar real ca float:

```
public class Format
{
    public static void main(String[] args)
    {
        System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

Mai jos este rezultatul:

```
3.141593, +00000003.1415926536
```

In imaginea de mai jos sunt explicate aceste caractere de conversie

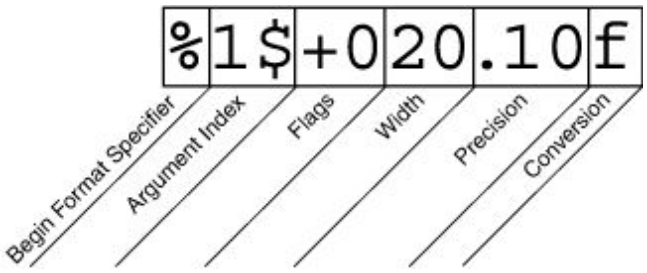


Figura 5.7 Specificatori de format

Serializarea obiectelor

Cheia pentru a scrie intr-un obiect este de a reprezenta starea lui intr-o forma serializabila, cu informatii suficiente pentru a reconstrui acel obiect la citire. De aceea citirea obiectelor si scrierea lor se numeste serializare.

Serializarea se poate folosi astfel:

1. Remote method invocation (RMI) - comunicatii intre obiecte via soketi.
2. Persitenta usoara – arhivarea unui obiect pentru invocarea ulterioare in program.

Cum scriem obiecte?

De exemplu, se preia timpul in milisecunde, se construieste un obiect de tip Date si se serializeaza obiectul:

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

`ObjectOutputStream` trebuie sa fie construit pe baza altui stream. In acest caz se construieste pe baza unui `FileOutputStream` care directioneaza catre „theTime”

Metoda writeObject serializeaza obiectul, si scrie acest stream obtinut catre destinatie, mentinand relatiile dintre obiecte.

Cum citim obiecte ?

Odata ce obiectele au fost scrise intr-un stream, acestea se pot citi pentru a reconstrui obiectele. Mai jos sunt citite obiectele care au fost scrise prin codul de mai sus:

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
 Date date = (Date)s.readObject();
```

Metoda readObject deserializeaza urmatorul obiect din stream si recursiv toate referintele lui pentru ca starea obiectului sa fie aceeasi ca la momentul serializarii.

Un obiect este serializabil doare daca clasa din care face parte, implementeaza interfata *Serializable*. Serializarea se poate personaliza, prin implementarea a doua metode din interfata Serializable: `readObject` si `writeObject`.

Pentru un control cat mai bun al relatiilor dintre clasa si superclasa, trebuie implementata si interfata *Externalizable*.

Mai jos avem un exemplu pentru a demonstra serializarea obiectelor

```
import java.io.*;
public class Serializare
{
    public static void main(String args[])
    {
        // Object serialization
        try
        {
            MyClassToSerialize objectOut = new
            MyClassToSerialize("Hello", 48, 34.658);
            System.out.println("object out: " + objectOut);
            FileOutputStream fos = new FileOutputStream("serial");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(objectOut);
            oos.flush();
            oos.close();
        }
        catch(Exception e)
        {
            System.out.println("Exception during serialization: " + e);
            System.exit(0);
        }
        // Object deserialization
        try
        {
            MyClassToSerialize objectIn;
            FileInputStream fis = new FileInputStream("serial");
            ObjectInputStream ois = new ObjectInputStream(fis);
            objectIn = (MyClassToSerialize)ois.readObject();
            ois.close();
            System.out.println("object2: " + objectIn);
        }
        catch(Exception e)
        {
            System.out.println("Exception during deserialization: " +
e);
            System.exit(0);
        }
     }
}
class MyClassToSerialize implements Serializable
{
        String s;
        int i;
        double d;
        public MyClassToSerialize(String s, int i, double d)
        {
            this.s = s;
            this.i = i;
```

```
            this.d = d;
        }
        public String toString()
        {
            return "s=" + s + "; i=" + i + "; d=" + d;
        }
}
```

Clasa care implementeaza interfata *Serializable* este `MyClassToSerialize`. Obiectele de acest tip de data pot fi serializate. In programul de mai sus serializarea se realizeaza atunci cand apelam metoda `oos.writeObject(objectOut);` , moment in care se salveaza in fisierul serial starea obiectului `objectOut`. Daca deschidem acest fisier, cu un text editor vom vedea ca nu poate fi inteles, ceea ce inseamna un plus in securitate. Seria de bytes poate fi folosita pentru deserializarea obiectului folosind metoda

```
objectIn = (MyClassToSerialize)ois.readObject();
```

Aici se citeste din fisier stream-ul si se construieste un obiect care este transformat in `MyClassToSerialize`, pentru a fi folosit ulterior.

Clasa StringBuffer

Am vazut mai sus ca sirurile in general sunt imutabile. Ceea ce inseamna ca odata declarate cu un numar de caractere, ele raman fix cu acel numar de caractere. Clasa String este implementata tot ca un sir de caractere, deci este imutabila. Un StringBuffer este asemenea unui String, dar poate fi modificat, si anume lungimea acestui string poate fi modificata prin cateva functii pe care le vom studia.

StringBuffer-ele sunt sigure in multithreading, adica pot fi folosite de mai multe fire de executie, fara a genera un comportament aleator.

String-urile pot fi concatenate si aceasta operatie reda la final un singur string format din string-urile alipite:

```
x = "a" + 4 + "c"
```

Problema aici este ca pentru a realiza aceasta operatie este nevoie de patru obiecte si anume sirurile ”a”, ”4”,”c” si *x*. Aceasta creare de obiecte, a caror referinta se pierde, duce la aparitia unui overhead (instructiuni, operatiuni auxiliare pentru indeplinirea unei operatiuni principale), ceea ce inseamna lipsa de eficienta ca timp si ca memorie.

Modul in care se lucreaza elegant cu String-uri este urmatorul:

```
x = new StringBuffer().append("a").append(4).append("c") .toString()
```

Prin aceasta operatiune se creeaza doar un obiect si anume un *StringBuffer* caruia i se alipeste la sfarsit, pe rand, string-ul ”a” apoi ”4” apoi ”c”. Pe urma acest *StringBuffer* este convertit la tipul *String*.

Iata un alt exemplu pentru functia append care suporta diverse supraincarcari pentru mai multe tipuri de data:

```
char[] c1 = new char[] {'S','i','r','d','e','c','h','a','r'};
StringBuffer sbc =  new StringBuffer("Sirul nou format : ");
sbc.append(c1);
System.out.println(sbc);
```

Alta functie este cea de stergere si anume *delete*:

```
StringBuffer sb = new StringBuffer("Hello World!");
sb.delete(0,6);
sb.delete(5,6);
System.out.println(sb);
```

Functia pentru inlocuire de string-uri este replace:

```
StringBuffer sb = new StringBuffer("Hello World!");
System.out.println("Original Text : " + sb);
sb.replace(6,11,"Lume");
sb.replace(0,5,"Buna ");
System.out.println("Replaced Text : " + sb);
```

Pe langa aceste functii, mai sunt o serie de metode pentru inserare, de cautare, redimensionare etc.