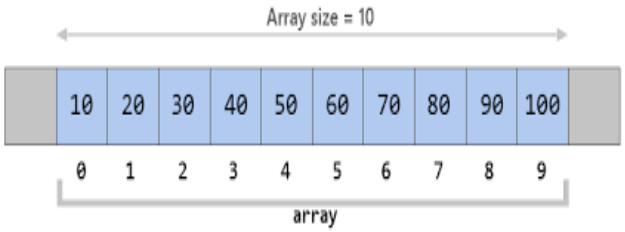
U.T. 3.- ESTRUCTURES D'EMMAGATZEMATGE



1. ARRAYS

1.1. Vectors

Els arrays són estructures de dades que agrupen, baix un mateix identificador, un conjunt de dades d'un mateix tipus. En Java, cada array és tractat com un objecte i es declara amb la següent sintaxi:

```
tipus nomArray[] = new tipus[midaArray];
```

Vegem dos exemples. En el primer anem a crear un array per guardar 10 valors numèrics double, mentre que en el segon crearem un array amb capacitat per contenir 5 objectes rectangle:

```
double nums[] = new double[5];
nums[4] = 3; // guarde un 3 en l'última posició de l'array

rectangle rs[] = new rectangle[5];
for (int i = 0; i < 5; i ++)
    rs[i] = new rectangle();
```

Dues puntualitzacions per entendre l'anterior codi:

- per treballar amb cadascun dels elements de l'array utilitzem un índex que, per a la primera posició, és 0 i, suposant N la mida de l'array, N-1 per a l'últim. Observa que, en l'exemple, l'array té 5 elements (amb index entre 0 i 4).
- si l'array és d'objectes, no d'un tipus bàsic, no n'hi ha prou amb crear l'array sinó que, a més, hem de crear-ne individualment cada objecte en la corresponent posició.

Els arrays també poden ser creats inicialitzant els seus valors en la declaració. Observa la seva sintaxi en el següent exemple:

```
int nums[] = {3,7,8,2};
```

Com pots observar, en aquest cas no usem *new* i, al seu lloc, hem indicat, entre claus, el conjunt de valors a contenir en l'array. La mida de l'array queda definida implícitament pel nombre de valors que incloem entre les claus.

Com hem dit, Java tracta els arrays com a objectes. Per exemple, tots ells contenen un atribut sencer d'ús públic, *length*, que contindrà com a valor la mida de l'array.

```
// mostrarà "L'array té 5 elements"
System.out.println("L'array té " + nums.length + " elements");
```

1.2. Matrius

Els arrays vistos fins ara són unidimensionals, també anomenats **vectors**. Però també podem crear array de 2 o més dimensions. Els de dues dimensions són anomenats **matrius**, mentre que els de 3 o més, d'ús molt poc freqüent, són coneguts com políedres. Anem a veure com treballar amb les matrius. Comencem per la sintaxi de la seva creació:

```
tipus nomMatriz[][] = new tipus[numFiles][numColumnes];
```

Per exemple:

```
rectangle matriuRs[][] = new rectangle[3][4];
```

Si la matriu és de tipus objecte, com l'anterior, hem de crear cadascun dels objectes, columna per columna, des de la primera fins a l'última fila:

```
for (fila = 0 ; fila < 3 ; fila++)
    for (columnes = 0 ; columna < 4 ; columnes++)
        matriuRs[fila][columna] = new rectangle();
```

Per a accedir a un element de la matriu ho farem indexant amb 2 índexs: primer l'índex de fila i, després, el de columna. Com pots observar en el codi anterior, l'índex de fila i de columna també comencen amb zero, corresponent a la primera fila o l'última columna. Per exemple:

```
int matriuEnters[] = new int[3][4];     // 3 files (0,1,2) i 4 columnes (0,1,2,3)
matriuEnters[1][2] = 4;
```

permetria assignar, escriure, un 4 en l'element de la matriu que ocupa la tercera columna de la segona fila.

En Java, a més, les matrius no tenen perquè ser regulars, és a dir, amb igual nombre de columnes en totes les files. També podem crear **matrius irregulars**. Per a això definim la matriu especificant el nombre de files però no el nombre de columnes, ja que aquest pot ser diferent en cada fila, i posteriorment hem de definir fila a fila el nombre de columnes en cada cas, amb la sintaxi de definició dels vectors (cada fila és un vector). Per exemple:

```
double matriu[][] = new double[3][]; // matriu de 3 files, nombre de columnes variable
matriu[0] = new double[1];  // primera fila, una columna
matriu[1] = new double[2];  // segona fila, dos columnes
matriu[2] = new double[3];  // tercera fila, tres columnes
```

Aquestes línies defineixen una matriu en forma d'escala, amb files d'1, 2 i 3 columnes respectivament. En aquest cas, podrem accedir, per exemple, a matriu[1][1] però no a matriu[1][2], atès que la segona fila de la matriu només té 2 columnes, amb índexs 0 i 1, però no 2. Això provocaria un error d'execució, una excepció anomenada *ArrayIndexOutOfBoundsException*, errada que es produeix sempre que intentem accedir a una posició inexistent en un array.

Per a les matrius, regulars o no, hem de tenir en compte que l'atribut *length* no és el nombre d'elements total, sinó el nombre de files.

1.3. Arrays com a paràmetres

Observa el següent exemple. En ell, modularment, es genera l'array, es carreguen valors fent assignacions i, finalment, mostrem a pantalla el contingut de l'array.

```
static final int TAM = 5;
public static void main(String args[])
{
        double nums[] = new double[TAM];;

        carregaValors(nums);
        mostraValors(nums);
}

public static carregaValors(double nums[])
{
        for (int i = 0 ; i < nums.length ; i++)
                nums[i] = i;
}

public static void mostraValors(double nums[])
{
        for (int i = 0 ; i < nums.length ; i++)
                System.out.print(nums[i] + "\t");
        System.out.println();
}
```

Observa com per al paràmetre actual únicament hem d'indicar el nom de l'array (quan cridem a la funció); en canvi al paràmetre formal (a la definició de la funció) hem d'expressar el tipus).

Observa, igualment, que les funcions com *carregaValors()* modifiquen el contingut de l'array i els canvis es veuen en main. Això significa que **els arrays són passats per referència**; és a dir, no es fa una còpia de l'array sinó que les funcions treballen amb l'array originari. Els tipus bàsics o primitius, en canvi, son passats per valor (es fa una còpia del paràmetre: el paràmetre formal i l'actual no són les mateixes variables).

```
public static void main(String args[])
{
        double num = 5;

        anulaValor(num);
        System.out.println(num);        // mostra 5, no 0
}

public static void anulaValor(double num)
{
        num = 0;        // modifiquem la còpia, no la dada originària
}
```

1.4. Arrays com a valor retornat

Vegem un exemple de funció que retorne un array d'enters.

```
/* funció que accepte com a paràmetre un enter i retorne un array amb tots el números
  natural entre 1 i el valor del paràmetre */
int[] arrayNaturals(int num)
{
        int array[] = new int[num];

        for (int i = 1 ; i <= num ; i++)
                array[i] = i;
        return array;
}
```

Observa que ho hem resolt creant i omplint l'array, i retornant el seu nom. La forma dús d'aquesta funció seria algo com:

```
int nums[] = arrayNaturals(8);  // retornaria un array amb els valors 1,2,3,...8
```

1.5. Arrays com a atributs

Els arrays també poden ser atributs en qualsevol classe. Vegem un exemple.

```
public class LlistaValors {
    // Atribut: array d'enters
    private double[] valors;
    private int quantitat; // Per a controlar els elements existents a l'array

    // Constructor que rep la mida màxima de l'array
    public LlistaValors(int midaMaxima) {
          this.valors = new double[midaMaxima];
          this.quantitat = 0;
```

```java
    }

    // Mètode per afegir un valor a l'array
    public void afegirValor(int nombre) {
          if (quantitat < valors.length) {
              valors[quantitat] = nombre;
              quantitat++;
              System.out.println("Valor " + nombre + " afegit.");
          } else {
              System.out.println("No hi ha espai per afegir més valors.");
          }
    }

    // Mètode per eliminar un valor per la seva posició
    public void eliminarValor(int posicio) {
          if (posicio >= 0 && posicio < quantitat) {
              for (int i = posicio; i < quantitat - 1; i++) {
                  valors[i] = valors[i + 1];
              }
              quantitat--;
              System.out.println("Valor a la posició " + posicio + " eliminat.");
          } else {
              System.out.println("Posició invàlida.");
          }
    }

    // Mètode per mostrar tots els nombres
    public void mostrarValors() {
          System.out.print("Valors: [");
          for (int i = 0; i < quantitat; i++) {
              System.out.print(valors[i]);
              if (i < quantitat - 1) {
                  System.out.print(", ");
              }
          }
          System.out.println("]");
    }

    // Mètode principal per provar la classe
    public static void main(String[] args) {
          LlistaValors llista = new LlistaValors(5);

          llista.afegirValor(10);
          llista.afegirValor(20);
          llista.afegirValor(30);

          llista.mostrarValors(); // Mostra: [10, 20, 30]

          llista.eliminarValor(1); // Elimina el 20

          llista.mostrarValors(); // Mostra: [10, 30]
    }
}
```

2. PARÀMETRES A LA FUNCIÓ PRINCIPAL

La funció principal (*main*) també pot rebre paràmetres des d'el terminal d'execució. Aquestos paràmetres són un conjunt o array de objectes String (*String args[]*). El primer d'ells serà *args[0]*, el segon *args[1]*, etc. Vegem un exemple:

```
public static void main(String args[])
{
        if (args.length > 0)
                for (int i=0 ; i< args.length ; i++)
                        System.out.println("El paràmetre " + i + " és " + args[i]);
        else
                System.out.println("S'ha executat sense paràmetres");
}
```

Observa com l'ús de l'atribut *length* permet conéixer quants paràmetres està passant l'usuari en l'execució. Un exemple d'ús del programa seria:

```
java programa paràmetre1 "segon paràmetre"
```

que mostraria en pantalla:

```
El paràmetre 0 és paràmetre1
El paràmetre 1 és segon paràmetre
```

Observa també com l'ús de les cometes dobles permeten que un paràmetre incloga espais en blanc, que per defecte actuen com a separadors dels paràmetres.

3. ALGORISMES D'ORDENACIÓ I RECERCA

3.1. Ordenació

Ens centrarem en l'ordenació i recerca d'arrays unidimensionals (vectors). Comencem pels algorismes d'**ordenació**. Per als diferents algorismes d'ordenació de vectors, podem veure una comparativa de l'eficiència d'alguns d'ells en el següent enllaç:

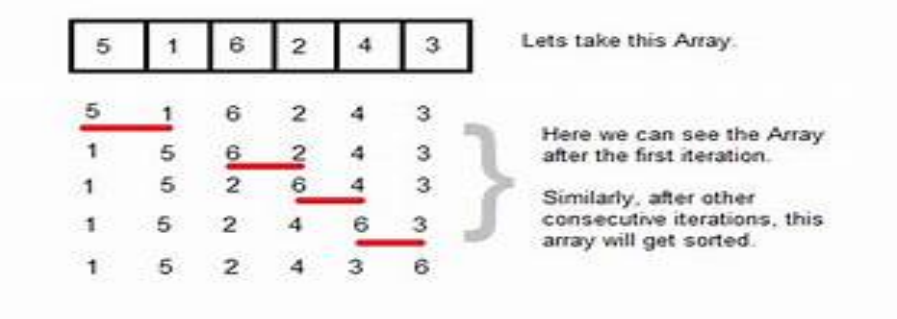
https://ed.ted.com/lessons/what-s-the-fastest-way-to-alphabetize-your-bookshelf-chand-john

Dels diferents algorismes comentats ens centrarem en l'algorisme conegut amb el nom de "mètode de la bombolla":

```
limit = TAM - 2 // penúltima posició de l'array
Mentre limit >= 0
    i = 0
    Mentre i <= limit
        Si array[i] > array[i+1]
            aux = array[i]
            array[i] = array[i + 1]
            array[i + 1] = aux
        FiSi
        i = i + 1
    FiMentre
    limit = limit - 1
FiMentre
```

Bàsicament, el que fa l'algorisme, mitjançant dos bucles niats, és:

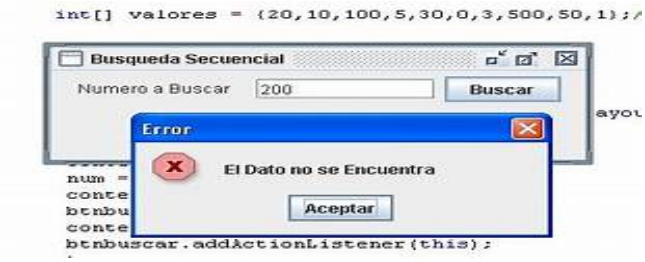
- compara cada element (el d'índex *i*) amb el seu següent (el d'índex *i+1*) i els intercanvia, utilitzant una variable auxiliar, si estan desordenats
- amb el bucle exterior es controla el límit fins on hem d'arribar a la comparació de cada valor amb el seu següent
- cada vegada que es completa una passada pel bucle exterior s'està deixant el major, de tots els valors comparats, en la posició última; per tant, amb cadascuna d'aquestes passades estem reduint el límit en una unitat. El límit és el valor últim que hem de comparar (amb el següent).
- el bucle interior sempre comença amb la primera posició i avança, d'un en un, fins a arribar al límit



Aquesta primera versió de l'algorisme és optimitzable, ja que no detecta quan la matriu es troba ja ordenada i, per tant, continuaria fent passades innecessàries pel bucle extern quan l'array hagués quedat ja perfectament ordenat.

L'optimització consistirà en afegir un booleà que anomenarem *ordenat* i iniciarem a fals al principi de l'algorisme. La condició lògica de l'bucle extern la fem composta afegint que, per seguir amb l'algorisme, siga condició necessària que aquesta variable estiga a *false* perquè en l'anterior passada, o la primera vegada, s'haja detectat algun canvi de posició. Cada vegada que iniciem una nova passada pel bucle intern tornem a deixar el booleà a *true* i només canviarà a *false* al detectar una parella de valors que hagen de intercanviar-se. Això precisament és el que forçarà a una nova passada pel bucle exterior. Només quan en alguna d'aquestes passades, des del primer element de l'array fins al límit, no intercanvie cap parell de valors aleshores el booleà *ordenat* quedarà a *true*, la qual cosa evitarà seguir ordenant:

```
limit = TAM - 2 // penúltima posició de l'array
ordenat = false
Mentre limit >= 0 AND ordenat = false
    i = 0
    ordenat = true
    Mentre i <= limit
        Si array[i] > array[i+1]
            aux = array[i]
            array[i] = array[i + 1]
            array[i + 1] = aux
            ordenat = false
        FiSi
        i = i + 1
    FiMentre
    limit = limit - 1
FiMentre
```



3.2. Recerca

Entre els diferents algorismes que podem utilitzar per trobar un valor dins d'un array, distinguirem dos tipus de cerques.

- cerques lineals. Aquelles en les que començem pel primer element de l'array, el qual comparem amb el valor buscat, i, servint-nos d'un comptador, avancem posició a posició al llarg de l'array fins al final o fins arribar a la posició on el valor buscat es trobe.
- cerques no lineals. Aquelles en que anem buscant en posicions saltejades segons algun criteri.

L'utilitzar un algorisme d'un tipus o un altre, i l'algorisme concret de la recerca, va a estar condicionat principalment pel fet que la matriu estiga desordenada o ordenada d'alguna manera (amb valors en ordre creixent o decreixent); també per la mida de l'array o pel fet de que busquem únicament la primera aparició del valor o, en canvi, totes les repetides vegades que puga aparèixer. Igualment, en algunes ocasions, no necessitem indicar les posicions on hem trobat el valor, sinó que pot ser suficient amb indicar si el valor s'ha trobat, o si no existeix en l'array. En aquest cas el resultat final serà un booleà.

Comencem amb una **cerca lineal** en la qual es retorne la posició de la primera aparició del valor buscat en una matriu desordenada (suposem l'array prèviament creat i carregat amb els seus valors):

```
Escriure "Introdueix el valor a buscar"
llegir num
posic = 0
Mentre array[posic] != num AND posic < TAM
    posic = posic + 1
FiMentre
Si array[posic] = num
    Escriure num, "trobat en la posició", posic
sinó
    Escriure num, "no es troba en l'array"
FiSi
```

En aquest cas, al haver suposat que la matriu està desordenada ens veiem obligats a arribar fins al final de l'array mentre no el trobem. Si l'array estigués ordenat creixentment, no seria necessari arribar al final: n'hi hauria prou amb trobar un valor superior al buscat per aturar la recerca, ja que la resta de valors serien tots igualment superiors. Es deixa a l'alumne la deducció d'aquesta variant de l'algorisme.

Com a exemple de cerca no lineal, veurem la recerca dicotòmica. Aquesta pressuposa l'array ordenat, en l'exemple creixentment:

```
Escriure "Introdueix el valor a buscar"
llegir num
esq = 0; dr = TAM - 1; centre = (esq + dr) / 2
Mentre array[centre] != num AND esq <= dr
     Si array[centre] <num
        esq = centre +1
     sinó
        dr = centre - 1
```

```
     FiSi
     centre = (esq + dr) / 2
FiMentre
Si array[centre] == num
     Escriure num, "trobat en la posició", centre
sinó
    Escriure num, "no es troba en l'array"
FiSi
```

En aquest algorisme es treballa amb 3 variables (esquerra, dreta i centre) que van acotant cada vegada més el marge on buscar a l'array. Inicialment abasten totes les posicions i, amb la variable centre, sempre en la mitjana de les altres dues que marquen els límits, es va restringint aquest marge modificant el límit esquerre o el dret de la recerca segons el signe de la comparació de l'alternativa doble interior al bucle .

4. CADENES DE CARACTERS

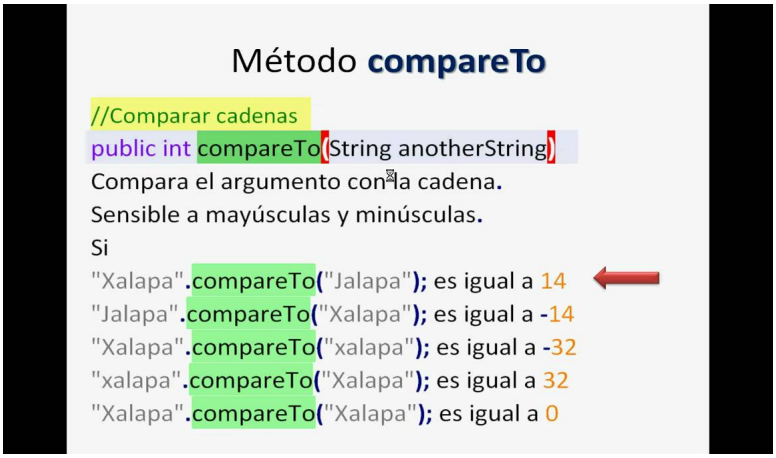
En Java per treballar amb alfanumèrics (cadenes de caracters) ho fem a través de la classe *java.lang.String*. Consulta aquesta i els mètodes que conté en la documentació del JDK. Aquesta classe, d'ús molt freqüent, té algunes particularitats. Entre elles destaquem:

- sintaxi reduïda de declaració
- els objectes *String* són immutables, quan els modifiquem realment no canvien sinó que es genera un altre objecte del mateix tipus amb el nou text i es fa que la referència apunte al nou objecte String, quedant l'anterior "desconnectat" i inservible. Això pot suposar un malbaratament important de memòria si es modifica l'*String* en nombroses ocasions.
- quan parlem, més endavant, de pas de paràmetres per referència, aquesta classe serà una excepció en el comportament que citarem per a la resta de classes. Això és a causa de la immutabilitat comentada anteriorment.

Aquesta sintaxi reduïda ens permet crear i modificar un String simplement amb l'operador d'assignació:

```
String s = "hola"; // s'observa que no tenim necessitat d'utilitzar new
s = "adéu";
```

Pots observar també que els literals String aniran sempre entre cometes dobles.



Entre els mètodes inclosos en la classe anem a ressaltar algun dels més utilitzats:

char charAt(int index)

Retorna el valor de l'caràcter en la posició indicada per l'índex.

int compareTo(String anotherString)

Compara dos objectes String lexicogràficament.

String concat(String str)

Concatena l'objecte String especificat com a paràmetre al final d'aquest String.

boolean contains(CharSequence s)

Retorna veritable si i només si aquest String conté la seqüència de caracters especificada.

boolean endsWith(String suffix)

Comprova si el String finalitza amb el sufix indicat.

boolean equals(Object anObject)

Compara aquest String amb l'objecte especificat.

boolean equalsIgnoreCase(String anotherString)

Compara aquest String amb un altre passat com a paràmetre, ignorant diferències entre majúsculas i minúscules.

int indexOf(int ch)

Retorna la posició on apareix el caràcter especificat dins de l'String (primera aparició).

boolean isEmpty()

Retorna veritable si, i només si, està buit (longitud 0).

int length()

Retorna la longitud d'aquest String.

String replace(char oldChar, char newChar)

Retorna un String resultat de reemplaçar totes les ocurrències del caràcter oldChar per newChar.

String replaceAll(String regex, String replacement)

Reemplaça cada substring d'aquest String que concorde amb l'expressió regular indicada pel reemplaçament indicat.

String replaceFirst(String regex, String replacement)

Similar a l'anterior, reemplaça només la primera aparició.

boolean startsWith(String prefix)

Comprova si aquest String comença amb el prefix especificat.

boolean startsWith(String prefix, int òfset)

Comprova si el substring a partir de la posició indicada per l'índex coincideix amb el prefix especificat com a primer paràmetre.

CharSequence subSequence(int beginIndex, int endIndex)

Retorna una seqüència de caracters que és subseqüència de la total.

String substring(int beginIndex)

Com l'anterior, indicant només l'inici.

String substring(int beginIndex, int endIndex)

Retorna un String que és part de l'inicial.

char[] toCharArray()

Converteix el String a array de caracters.

String toLowerCase()

Converteix a minúscules tots els seus caracters.

String toUpperCase()

Converteix a majúscules tots els seus caracters.

String trim()

Retorna un String d'igual contingut, però eliminant qualsevol espai en blanc inicial o final (no els intermedis).

static String valueOf(boolean b)

static String valueOf(char c)

static String valueOf(char[] data)

static String valueOf(char[] data, int òfset, int count)

static String valueOf(double d)

static String valueOf(float f)

static String valueOf(int i)

static String valueOf(long l)

Tots els valueOf retornen, el propi objecte *String*.

Com ja s'ha dit, els objectes de la classe *String* són immutables. Això, en algunes ocasions, suposa un inconvenient que podem evitar amb una classe amb una utilitat similar, però per a la qual els seus objectes sí que són mutables. És la classe **StringBuffer**. Observa en la documentació els constructors disponibles per a aquesta classe, així com la resta de mètodes entre els quals destacarem els següents.

int length ()

Retorna la longitud d'el text contingut

int capacity ()

Retorna la capacitat. Quan es crea un d'aquests objectes es fa amb una capacitat igual al nombre de caracters emmagatzemats més 16.

StringBuffer append(argument)

Afegeix l'argument a la fi de la cadena. Aquesta funció està sobrecarregada, és a dir, hi ha diverses amb diferents tipus per a l'argument: *int*, *long*, *double*, *String*, etc.

StringBuffer insert(int pos, arg)

Afegeix l'argument en la posició indicada pel primer paràmetre. També està sobrecarregada amb els diferents tipus ja indicats.

StringBuffer reverse()

Inverteix la seqüència dels caracters del text.

StringBuffer delete(int x, int i)

Elimina els caracters entre les posicions indicades pels paràmetres.

StringBuffer replace(int x, int i, String s)

Reemplaça els caracters entre les posicions indicades pel contingut de l'String passat com a tercer paràmetre.

String substring(int x, int i)

Retorna 1'*String* que conté la cadena entre *x* i *i* (o a partir de *x*, si no s'indica *i*).

String toString()

Retorna un objecte *String* amb el mateix text.

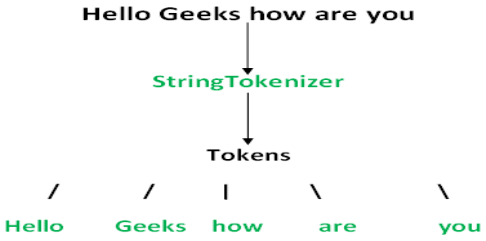
char charAt(int x)

Totalment coincident amb el *charAt* de *String*.

void setCharAt(int x, char c)

Reemplaça el caràcter existent en la posició indicada pel especificat en el segon paràmetre.

5. LA CLASSE STRINGTOKENIZER



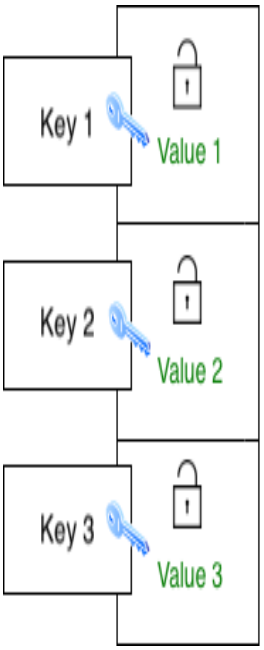
La classe *StringTokenizer*, a *java.util* (requereix clàusula import), ens permet dividir un text en parts o *tokens*. Partint d'un objecte *String* amb el text a trossejar, el procediment serà crear un objecte *StringTokenizer* (s'observa en la documentació els constructors de la classe) passant com a primer paràmetre l'*String* i, opcionalment, com a segon el/els caracters que actuen com a delimitadors. Si aquests no s'indiquen, per defecte actuen com a delimitadors l'espai en blanc, la tabulació i el canvi de línia. Un exemple:

```
StringTokenizer st = new StringTokenizer ( "La música calma a les feres", "ae");
// obtinc els tokens "L" " músic" " c" "lm" " l" "s f" "r" "s"
int num = st.countTokens ();
System.out.println ( "El text original ha quedat dividit en" + num + "elements.");
// mostrem cadascun dels elements
for (i = 0; i <num; i ++)
    System.out.println (st.nextToken ());
```

Actualment *StringTokenizer* està marcada com a obsoleta i es proposa usar *String.split()* com a alternativa.

6. TAULES HASH

Les taules *Hash* són una estructura de dades en la qual a cada clau se li fa correspondre un valor, de manera que les claus són totes diferents (no hi podrà haver dos elements amb la mateixa clau, indiferentment del valor associat), mentre que els valors sí es poden repetir per diferents elements.



Java incorpora una classe per treballar amb aquestes estructures de dades: *Hashtable*. Consulta la informació referent a aquesta classe en la documentació del JDK. Observa els diferents constructors disponibles i els mètodes inclosos en la classe. Entre aquests destacarem:

boolean contains(Object value)

Comprova si a alguna clau li correspon el valor especificat.

boolean containsKey(Object key)

Comprova si existeix aquesta clau en aquest objecte Hashtable.

boolean containsValue(Object value)

Retorna true si aquesta Hashtable conté una o més claus amb aquest valor.

Enumeration <V> elements()

Retorna un objecte Enumeration dels valors continguts en aquest Hashtable.

Enumeration <K> keys()

Retorna 1 Enumeration de les claus en aquest Hashtable.

V put(K key, V value)

Afegeix una entrada a l'Hashtable associant a aquesta clau aquest valor.

V remove(Object key)

Elimina l'entrada amb aquesta clau (i el seu corresponent valor).

boolean remove(Object key, Object value)

Elimina l'entrada per a la clau especificada només si té associat el valor indicat.

V replace(K key, V value)

Reemplaça l'entrada per al valor de clau especificat només si ja existia associada a algun altre valor.

boolean replace(K key, V oldValue, V newValue)

Igual que l'anterior però ha d'existir amb el valor indicat pel segon paràmetre, sent reemplaçat pel valor indicat com a tercer paràmetre.

int size()

Retorna el nombre d'entrades, o claus, a la taula.

Collection<V> values()

Retorna un objecte Collection amb tots els valors continguts.

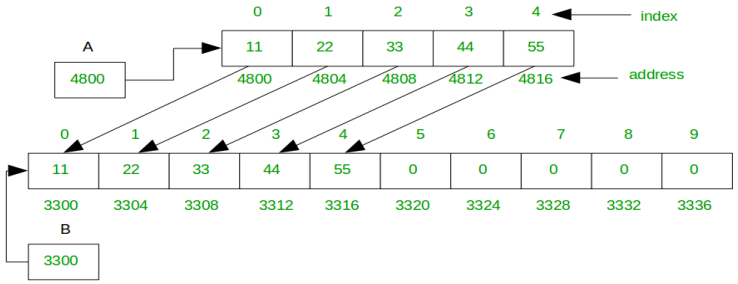
Treball per a l'alumne: estudiar les diferències entre la classe vista, *Hashtable*, i la classe *HashMap*, especialment pel que fa a eficiència i sincronicitat per a programació amb fils.

7. ARRAYS DINÀMICS I ALTRES ESTRUCTURES DE DADES

7.1. La classe Vector

Els arrays vistos fins ara eren tots estàtics, és a dir, arrays que no poden canviar de mida durant l'execució del programa. Hi haurà ocasions en que generem l'array amb una determinada mida i, posteriorment veiem que necessitem una mida més gran per emmagatzemar més valors o, per contra, vegem que hem reservat un nombre molt major al què realment necessitàvem. En ambdós casos, amb els arrays estàtics no podrem canviar aquesta mida.

Els arrays dinàmics són aquells que sí que ens van a permetre les dues operacions, tant ampliar com reduir la seva grandària durant l'execució de el programa. En altres llenguatges, com C, això s'implementa utilitzant punters i funcions de reserva directa d'memòria. En Java, utilitzarem classes ja implementades com *ArrayList* o *Vector*. Anem a veure aquesta última classe; realment les dues classes són molt similars però la primera d'elles no suporta la programació amb fils, mentre que la segona sí.



Extret de la documentació de la classe: "la classe Vector implementa una matriu d'objectes modificable en mida. Com en una matriu estàtica, conté components accessibles utilitzant un index enter. No obstant això, la mida d'un objecte Vector pot augmentar o disminuir segons es necessite afegir o eliminar elements després de la creació del Vector".

Cada objecte *Vector* permet l'optimització i gestió del seu emmagatzematge mitjançant una capacitat i una capacitat d'increment. La capacitat és sempre, al menys, igual a la mida del vector; usualment serà més gran atès que, quan es van afegint elements al *Vector*, es va ampliant aquesta en unitats iguals a la capacitat d'increment. Una aplicació pot incrementar la capacitat d'un *Vector* abans d'inserir una quantitat important de components; això pot reduir les vegades que siga necessari realojarlo en memòria.

Quan treballem amb objectes Vector hem de tenir en compte que s'ha d'indicar el tipus d'elements que guardarà. Per a això la notació a utilitzar serà algo com:

```
Vector<String> textos = new Vector<String>();
```

En l'exemple anterior estem creant un objecte Vector de tipus String. Per a contenir objectes d'altres classes utilitzarem la fórmula *Vector<E>*, on E serà el tipus corresponent.

Observa els diferents constructors disponibles i els mètodes inclosos en la classe. Entre aquests destacarem:

boolean add(E e)

Afegeix l'element especificat a la fi del Vector.

void add(int index, E element)

Insereix l'element en la posició especificada.

boolean addAll(Collection <? extends I> c)

Afegeix tots els elements continguts en l'objecte Collection al final d'aquest Vector, en l'ordre que són retornats pel Iterator de Collection.

void addElement(E obj)

Afegeix l'objecte a la fi del Vector, incrementant la seva grandària en una unitat.

int capacity()

Retorna la capacitat actual de el vector.

void clear()

Elimina tots els elements d'aquest Vector.

boolean contains(Object o)

Retorna true si aquest vector conté l'element indicat.

E elementAt(int index)

Retorna el component de la posició especificada.

Enumeration<E> elements()

Retorna una enumeració de tots els components de el vector.

E get(int index)

Igual que elementAt.

void insertElementAt(E obj, int index)

Insereix el objecte especificat en la posició indicada.

boolean isEmpty()

Comprova si el vector està buit.

Iterator <E> Iterator()

Retorna l'Iterator dels elements en el seu degut ordre.

E remove(int index)

Elimina l'element en la posició especificada.

boolean remove(Object o)

Elimina la primera aparició de l'element indicat si existeix.

boolean removeAll(Collection <?> c)

Elimina d'aquest Vector tots els elements que estigan continguts en l'objecte Collection indicat.

void removeAllElements()

Elimina tots els elements.

boolean removeElement(Object obj)

Elimina la primera aparició de l'objecte en el vector.

void removeElementAt(int index)

Elimina l'element de la posició indicada.

E set(int index, E element)

Reemplaça l'element present en la posició indicada per l'element especificat com a segon paràmetre.

void setElementAt(E obj, int index)

Igual que l'anterior invertint l'ordre dels paràmetres.

int size()

Retorna el nombre de components del vector.

La classe *Vector*, i també *ArrayList*, implementen l'interfície *Iterable*. Això vol dir que amb objectes d'aquestes classes podem fer ús del bucle *for* especial anomenat **foreach**.

Comparat amb el bucle *for* de Java, el mètode *foreach* de Java no necessita considerar l'índex o la grandària de l'array. Això evita que un índex estiga fora del rang vàlid. El bucle *foreach* utilitza internament un iterador per a obtenir successivament cada element de l'array o llista. L'iterador avança automàticament a través de l'array o colecció i acaba el bucle quan tots els elements han sigut recorreguts.

La sintaxi bàsica d'un bucle *foreach* a Java és la següent:

```
      for (tipus element : coleccio) {
              // bloc de codi
      }

*tipus: tipus de dades de l'array o col·lecció
*element: cada element de l'array o col·lecció s'assigna a esta variable en cada passada
*coleccio: nom de l'array o col·lecció
```

Dins del bucle, el bloc de codi s'executa per a cada iteració.

```
// array
String[] noms = {"Tim", "John", "Melissa"};
// també noms podria ser un objecte Vector o ArrayList, per exemple
// bucle foreach
for (String nom: noms)
       System.out.println(nom);
```

En la consola obtenim el següent resultat:

```
Tim
John
Melissa
```

7.2. La classe ArrayList

Els objectes de tipus ArrayList són estructures de tipus array redimensionable (dinàmic) que implementen, igual que *Vector* la interfície *List*. Permeten tots els elements, inclòs el valor *null* i atès que és redimensionable, disposa de mètodes per modificar la mida.

Per a l'accés concurrent amb diversos fils, cal tenir en compte que *ArrayList* no està sincronitzada entre fils i el seu accés haurà d'estar-ho de forma externa. Aquesta és la diferència entre ella i *Vector* que és igual, però ja està sincronitzada.

Operacions més comuns

- Afegir un element

```
List <String> llista = new ArrayList<String>();
String novaCadena = "Això és una cadena";

llista.add(novaCadena);
```

- Afegir un element en una posició determinada

```
int posicio = 3;
llista.add (3, novaCadena);
```

- Reemplaçar l'element que ocupa una posició determinada

```
int posicio = 3;
String altraCadena = "Això és una altra cadena";
llista.set(3, altraCadena);
```

- Eliminar un element

```
int posicio = 3;
llista.remove(posicio);
```

- Obtenir la referència a un element

```
int posicio = 3;
String unaCadena = llista.get(posicio);
```

- Conèixer el nombre d'elements

```
System.out.println("L'ArrayList té" + llista.size() + "elements");
```

- Eliminar tots els elements de la llista

```
llista.clear();
```

- Afegir tots els elements d'una altra llista

```
List<String> altraLlista = new ArrayList<>();
. . .
. . .
// Es pot passar com a paràmetre un objecte que implemente la interfície Collection
llista.addAll(altraLlista);
```

- Comprovar si aquesta buida

```
if (llista.isEmpty()) {
  System.out.println( "La llista no té elements");
}
```

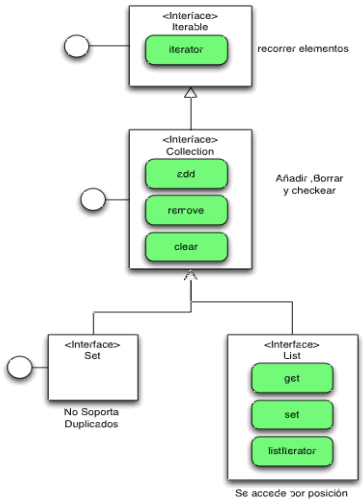
- Obtenir una matriu (estàtica) amb tots els elements de la llista

```
String[] array = llista.toArray();
```

S'ha de destacar que per a crear un objecte *List* (com *ArrayList* o *Vector*) tenim altres formes alternatives, depenent de la versió de Java amb la que treballem

```
List<String> list = Arrays.asList("a","b","c"); // per a versions de Java inferiors a la
List<String> list = List.of("a","b","c");       // per a versions de Java superiors o ig
```

7.3. Set i List



Entre les col·leccions de Java la interfície més general és *Iterable*, que definix que una col·lecció es pot recórrer retornant per a això un *Iterator*. Per la seua banda la interfície *Collection* fa referència a un conjunt d'elements recorribles. És en esta interfície on s'afigen mètodes com *add()*, *remove()* i *clear()* per a afegir o eliminar elements de la col·lecció. *Set* practicamente no afig res nou, llevat que definix que no podem tindre elements repetits. D'altra banda *List* sí que afig mètodes addicionals que ens permeten accedir per posició a cada element de la llista.

En el cas dels *Sets* existixen tres implementacions fonamentals:

- HashSet: definix el concepte de conjunt (grup d'elements no repetits) a través d'una estructura Hash. És el conjunt més habitual.
- TreeSet: definix el concepte de conjunt a través d'una estructura d'Arbre.
- LinkedHashSet: definix el concepte de conjunt afegint una llista doblement enllaçada que ens assegura que els elements sempre es recorren de la mateixa forma.

7.4. HashMap

HashMap és una taula *hash* que implementa la interfície *Map*. Permet el valor *null* tant per a valors com per claus. Al contrari que *TreeSet* no garanteix l'ordre dels elements o que aquest es mantinga.

Cal tenir en compte que l'accés a aquesta implementació no està sincronitzat, a diferència de *Hashtable*, equivalent en funcionalitat.

Operacions més comuns

- Afegir un element (parella clau-valor)

```
Map<String, Llibre> llibres = new HashMap <String,Llibre>();
. . .
Llibre llibre = new Llibre();
llibre.setTitol("Segrestat");
llibre.setAutor("Robert Louis Stevenson");
. . .
llibres.add(llibre.getTitol(), llibre);
```

- Obtenir un element

```
String titol = "Segrestat";
Llibre unLlibre = llibres.get(titol);
```

- Comprovar si eixisteix una clau

```
String titol = "Segrestat";
if (llibres.containsKey (titol)) {
  System.out.println ( "El llibre amb el títol" + titol + "existeix en la teva col·lecci
}
```

- Comprovar si està buit

```
if (llibres.isEmpty()) {
  System.out.println("La teua col·lecció de llibres està buida");
}
```

- Eliminar tots els elements

```
llibres.clear();
```

- Eliminar un element (per clau)

```
String titol = "Segrestat";
llibres.remove(titol);
```

- Obtenir un *Collection* amb tots els valors

```
Collection<Llibre> coleccioLLibres = llibres.values();
// ArrayList, TreeSet i LinkedList implementen la interfície Collection
```

- Obtenir un *Set* amb totes les claus

```
Setembre <String> titols = llibres.keySet();
// TreeSet i HashSet implementen la interfície setembre
```

- Comprovar la mida del *Map*

```
System.out.println("Tens " + llibres.size () + "llibres en la teva col·lecció");
```

- Concatenar tots els elements d'un altre *Map*

```
Map<String, Llibre> mesLLibres = new HashMap<String,Llibre>();
. . .
. . .
llibres.putAll (masLlibres);
```

7.5. TreeSet

TreeSet és una col·lecció que manté els elements ordenats de forma natural o bé a través d'un *Comparator*.

Operacions més comuns

- Afegir un element

```
Setembre <String> cadenes = new TreeSet<String>();
String unaCadena = "Això és una cadena";
cadenes.add(unaCadena);
```

- Afegir tots els elements d'una altra *Collection*

```
List <String> mesCadenes = new ArrayList<String>();
. . .
cadenes.addAll(masCadenes);
```

- Obtenir un element
- Eliminar tots els elements

```
cadenes.clear();
```

- Comprovar si hi ha un element en la llista

```
String altraCadena = "Això és una cadena";
if (cadenes.contains(altraCadena))
  System.out.println("La cadena existeix");
```

- Obtenir el primer element

```
String primeraCadena = cadenes.first();
```

- Obtenir i eliminar el primer element

```
String primeraCadena = cadenes.pollFirst();
```

- Obtenir l'últim element

```
String ultimaCadena = cadenes.last();
```

- Obtenir i eliminar l'últim element

```
String ultimaCadena = cadenes.pollLast();
```

- Comprovar si està buit

```
if (cadenes.isEmpty())
  System.out.println ("El Set de cadenes està buida");
```

- Comprovar el nombre d'elements

```
System.out.println("Té " + cadenes.size () + "cadenes");
```

- Obtenir la part del Set els elements són menors que un passat per paràmetre

```
String cadenaLimite = "Una cadena";
SortedSet <String> cadenesMenores = cadenes.headSet(cadenaLimite);
// SortedSet és la interfície que implementa TreeSet
```

- Obté la part del Set dels elements que estan entre dues determinats

```
String cadenaMenor = "Una cadena";
String cadenaMajor = "Més cadena";
SortedSet <String> cadenesMenors = cadenes.subSet(cadenaMenor, cadenaMajor);
// SortedSet és la interfície que implementa TreeSet
```

7.6. TreeMap

TreeMap és una col·lecció de parells clau-valor, com *HashMap*, però manté els elements ordenats per la seva clau de forma natural o bé a través d'un *Comparator*.

Operacions més comuns

- Afegir un element (parella clau-valor)

```
Map <String, Llibre> llibres = new TreeMap <,>();
. . .
Llibre llibre = new Llibre();
llibre.setTitol("Segrestat");
llibre.setAutor("Robert Louis Stevenson");
. . .
llibres.put(llibre.getTitulo(), llibre);
```

- Obtenir un element

```
String titolLlibre = "Segrestat";
Llibre aquestLlibre = llibres.get(títol);
```

- Comprovar si hi ha una clau

```
String titol = "Segrestat";
if (llibres.containsKey(titol))
  System.out.println ( "El llibre amb el títol" + títol + "existeix en la teva col·lecci
```

- Comprovar si està buit

```
if (llibres.isEmpty())
  System.out.println("El teu col·lecció de llibres està buida");
```

- Eliminar tots els elements

```
llibres.clear();
```

- Eliminar un element (per clau)

```
String titol = "Segrestat";
llibres.remove(titol);
```

- Obtenir una *Collection* amb tots els valors

```
Collection <Llibre> coleccioLlibres = llibres.values();
// ArrayList, TreeSet i LinkedList implementen la interfície Collection
```

- Obtenir un *Set* amb totes les claus

```
Setembre <String> titols = llibres.keySet();
// TreeSet i HashSet implementen la interfície
```

- Comprovar la mida del *Map*

```
System.out.println("Tens" + llibres.size() + "llibres en la teva col·lecció");
```

- Concatenar tots els elements d'un altre *Map*

```
Map<String, Llibre> masLlibres = new TreeMap <,>();
. . .
. . .
llibres.putAll(masLlibres);
```

7.7. LinkedList

LinkedList és una llista doblement enllaçada que implementa la interfície *List*. Convé recordar els avantatges de les llistes enllaçades, davant els arrays estàtics o dinàmics, en l'aprofitament màxim de l'espai en memòria.

Operacions més comuns

- Afegir un element

```
List <Llibre> llistaLlibres = new LinkedList<Llibre>();
. . .
Llibre llibre = new Llibre();
llibre.setTitol("Segrestat");
llibre.setAutor("Robert Louis Stevenson");
```

```
. . .
llistaLlibres.add(llibre);
```

- Inserir un element al principi

```
List <Llibre> llistaLlibres = new LinkedList<Llibre>();
. . .
Llibre llibre = new Llibre();
llibre.setTitol("Segrestat");
llibre.setAutor("Robert Louis Stevenson");
. . .
llistaLlibres.addFirst(llibre);
```

- Afegir un element al final

```
List <Llibre> llistaLlibres = new LinkedList<Llibre>();
. . .
Llibre llibre = new Llibre();
llibre.setTitol("Segrestat");
llibre.setAutor("Robert Louis Stevenson");
. . .
llistaLlibres.addLast(llibre);
```

- Afegir tota una col·lecció al final

```
List <Llibre> llistaLlibres = new LinkedList<Llibre>();
List <Llibre> altraLlistaDeLlibres = new ArrayList<Llibre>();
. . .
// Podem afegir qualsevol objecte que implemente la interfície Collection
llistaLlibres.addAll(altraLlistaDeLlibres);
```

- Obtenir un element

```
Llibre unLlibre = llistaLlibres.get(4);
```

- Obtenir el primer element

```
Llibre unLlibre = llistaLlibres.getFirst();
```

- Eliminar (i obtenir) el primer element

```
Llibre primerLlibre = llistaLlibres.removeFirst();
```

- Eliminar (i obtenir) l'últim element

```
Llibre ultimLlibre = llistaLlibres.removeLast();
```

- Eliminar (i obtenir) un element qualsevol

```
Llibre unLlibre = llistaLlibres.remove(3);
```

- Eliminar tots els elements

```
llistaLlibres.clear();
```

- Obtenir el nombre d'elements de la llista

```
System.out.println("Aquesta llista té" + llistaLlibres.size() + "llibres");
```

- Obtenir una matriu (estàtica) amb tots els elements de la llista

```
Llibre[] arrayLlibres = llistaLlibres.toArray();
```

7.8. Comparativa de la JCF (Java Collection Framework)

JCF Class	Positioning	Ordering	Duplicates	Sample Use
TreeSet	None	Ascending	Not Allowed	Keep unique list of words from a document sorted
HashSet	None	None	Not Allowed	Keeps a unique list of words from a document to be referenced frequently
ArrayList	Numerical	None	Allowed	Keeps an "in-memory" lookup table of static data
LinkedList	Numerical	Insertion Order	Allowed	Keeps a list of words that change frequently
HashMap	Object as key	None	Unique Keys	Maps words to word frequency
TreeMap	Object as key	Ascending	Unique Keys	Maps sorted words to word frequency

JCF Classes Cheat Sheet

JCF: Java Collection Framework

8. ENUMERACIONS

Les enumeracions són tipus definits per l'usuari amb una llista invariable de noms propis com a valors.

- Es compilen com classes que hereten, o estenen, *java.lang.Enum* i tenen mètodes predefinits, que podem veure a la documentació de Javadoc, encara que es poden afegir més.
- Els seus valors són atributs estàtics, objectes que es construeixen la primera vegada que s'utilitzen i que, curiosament, són del propi tipus enumerat i se'ls pot afegir atributs i mètodes propis.

Abans de Java 1.5 solien usar-se constants senceres per simular enumerats.

```
class Semafor {
    public static final int ROIG = 0;
    public static final int AMBAR = 1;
    public static final int VERD = 2;
}
```

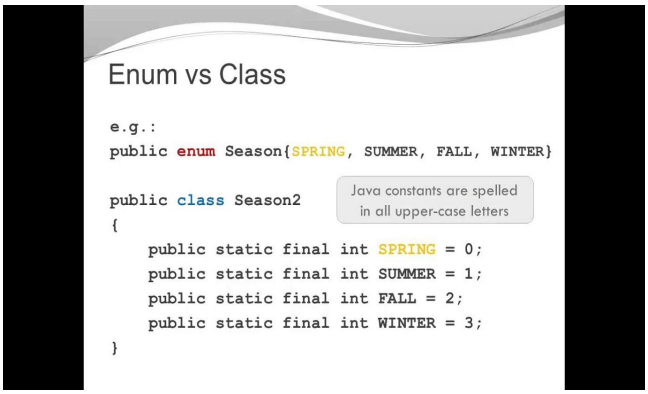
La notació actual és molt més elegant, segura i comprensible (els valors se solen seguir escrivint en majúscules).

```
enum Semafor {ROIG, AMBAR, VERD};
```

El compilador tradueix els enumerats a una classe final.

```
final class Semafor extends java.lang.Enum {
    public static final Semafor ROIG;
    public static final Semafor AMBAR;
    public static final Semafor VERD;

    public static final Semafor[] values();
    public static final Semafor valueOf(java.lang.String);
    static {};
}
```



Mètodes interessants:

int compareTo(E o)
: Els enumerats són comparables (implementen l'interfície Comparable)

boolean equals(E o)
: En els enumerats equals és equivalent a ==

String name()
: Retorna el nom del valor

int ordinal()
: Retorna la posició del valor a la llista

static E valueOf(String s)
: Retorna el valor E corresponent al nom s

Vegem un programa d'exemple que fa ús d'un parell d'enumerats.

```
import java.util.*;

enum Rang
{ DOS, TRES, QUATRE, CINC, SIS, SET, SOTA, CAVALL, REI, AS }

enum Familia
{ BASTOS, ESPASES, ORS, COPES }

// classe que representa una carta de la baralla
```

```
class Carta
{
        private final Rang rang;
        private final Familia familia;
        public Carta(Rang rang, Familia familia)
        {
                this.rang = rang;
                this.familia = familia;
        }

        public Rang getRang() { return rang; }
        public Familia getFamilia() { return familia; }
        public String toString() { return rang + " DE " + familia; }
}

// programa que fa ús de la classe anterior
public class enums
{
        private static final Vector<Carta> baralla = new Vector<Carta>();

        // inicilitzador static
        static
        {
                for (Familia familia : Familia.values())
                        for (Rang rang : Rang.values())
                                baralla.add(new Carta(rang, familia));
        }

        public static void main(String[] args)
        {
                Iterator it = baralla.iterator();
                while(it.hasNext())
                        System.out.println(it.next());


                Carta c = new Carta(Rang.QUATRE,Familia.ESPASES);
                System.out.println(c);  // toString()

                System.exit(0);
        }
}
```