TEMA 8: PAQUETES Y CLASES DE UTILIDAD. ESTRUCTURAS DE DATOS.

Programando en ***Java*** nunca se parte de cero: siempre se parte de la infraestructura definida por el API de ***Java***, cuyo ***packages*** proporcionan una buena base para que el programador construya sus aplicaciones. En este tema veremos algunas clases que serán de utilidad.

Recordemos que los paquetes son una forma de agrupar clases e interfaces asociadas. Un *paquete* es un conjunto de clases almacenadas todas en un mismo directorio. La jerarquía de directorios es la misma que la jerarquía de paquetes. Los nombres de los paquetes deben coincidir con los nombres de los directorios. El nombre completo de una clase es el del paquete seguido por el nombre de la clase, separados por un punto.

Algunas de las clases que proporciona ***Java*** en forma de paquetes son:

- ✓ ***lang:*** para funciones del lenguaje. Contiene las clases de los elementos básicos del sistema como son *Object, System, Math, Number, String, StringBuffer,...,* además incluye las clases que encapsulan los tipos simples (*wrapper classes*). Las clases del paquete java.lang son importadas de manera automática para todos los programas, no ocurre así con el resto de paquetes.
- ✓ ***util:*** para utilidades adicionales. Entre otras clases contiene colecciones pilas, colas, tablas *hash*,...), la clase StringTokenizer, Random, ...
- ✓ ***io:*** para entrada y salida de *streams* y ficheros.
- ✓ ***text:*** para formato especializado.
- ✓ ***awt y swing:*** para diseño gráfico e interfaz de usuario.
- ✓ ***awt.event:*** para gestionar eventos.
- ✓ ***applet:*** para crear aplicaciones de red. Contiene lo necesario para que un programa Java pueda ser cargado y ejecutado por un *browser* (navegador) a partir de una referencia en una página HTML.
- ✓ ***net:*** para comunicaciones.
- ✓ ...

1. ARRAYS

Un array o tabla (arreglo en algunas tradiciones) es una estructura de datos interna, con un número fijo de elementos del mismo tipo. Los elementos de los arrays son almacenados en posiciones consecutivas de memoria, y, en general, aunque forman un conjunto compacto, pueden ser tratados como variables independientes cuando se referencian de modo individual.

También se puede definir un array como un conjunto fijo, finito y ordenado de elementos, todos del mismo tipo y bajo un nombre común. Se denominan **componentes** a los elementos de un array.

Según el número de dimensiones los arrays pueden ser:

- ❑ Unidimensionales, conocidos como vectores.
- ❑ Bidimensionales, conocidos como matrices.
- ❑ Multidimensionales, conocidos como poliedros.

Se llaman **índices** a los indicadores que permiten determinar la posición de un elemento dentro de un array (en la dimensión correspondiente). Se necesitan tantos índices como dimensiones tenga un array.

En la definición o especificación de un tipo Array intervienen otros dos tipos:

- ✓ El tipo Base T: es el tipo de los elementos que constituyen los arrays.
- ✓ El tipo Indice I (ordinal de cardinalidad finita): a cada componente del array se le asocia un valor de tipo I denominado índice. El primer elemento de I se asocia al primer elemento del array (de tipo T), el segundo de I al segundo del array, y así sucesivamente. A través de estos índices es como se accede de forma directa a una determinada componente del array. La cardinalidad de este tipo I determina la longitud de los arrays.

La **longitud** o **tamaño** de un array es el número de componentes que contiene.

Los componentes de un array se utilizan de la misma forma que cualquier otra variable de un programa, pudiendo por tanto intervenir en instrucciones de asignación, entrada/salida, etc.

Al declarar un array, se reserva una cantidad fija de memoria inalterable durante toda la ejecución del programa, ocupando cada componente un espacio igual al del tipo de dato que aparece en la declaración, y estando todas ellas en posiciones contiguas de memoria.

Para ver la utilidad de los arrays consideremos el siguiente ejemplo:

Una casa comercial tiene en plantilla 20 agentes de ventas (identificados por números del 1 al 20) que cobran comisión sobre la parte de sus operaciones comerciales que excede los 2/3 del promedio de ventas del grupo.

Se necesita un algoritmo que lea el valor de las operaciones comerciales de cada agente e imprima el número de identificación de aquellos que deban percibir comisión así como el valor correspondiente a sus ventas.

Este problema tiene 2 aspectos que juntos lo hacen difícil de programar con los mecanismos vistos hasta ahora:

- ✓ Procesamiento similar sobre los datos de cada agente: leer ventas, calcular promedio de ventas, comparar niveles y decidir si se debe o no recibir comisión.
- ✓ Se necesita almacenar durante la ejecución del programa los valores de las ventas de cada agente; para el cálculo del promedio y para la comparación de niveles.

Esto implica que necesitamos 20 variables para retener los valores de las ventas. Un posible algoritmo sería:

```
Constantes
     porcion =  2.0 / 3.0
Variables
     real ventas1, ventas2, ..., ventas20, suma, umbral
Inicio
     suma = 0.0
     lee(ventas1)
     suma = suma + ventas1
     lee(ventas2)
     suma = suma + ventas2
     ...
     lee(ventas20)
     suma = suma + ventas20
     umbral = porcion * (suma / 20.0)
     si ventas1 > umbral entonces
          escribir('1',ventas1)
     fin-si
     si venta2 > umbral entonces
          escribir('2',ventas2)
     fin-si
     ...
     si ventas20 > umbral entonces
          escribir('20',ventas20)
     fin-si
Fin
```

La escritura de este programa es, como se ve, bastante tediosa. El problema se agravaría más si en lugar de 20 hubiera 200 agentes de ventas. Una mejor solución consiste en considerar estas variables de ventas como componentes de una estructura de datos array. Cada componente tendrá como índice asignado el número de identificación del correspondiente agente de ventas.

Los ***arrays*** de ***Java*** se tratan como objetos de una clase predefinida. Los ***arrays*** son ***objetos***, pero con algunas características propias. Los ***arrays*** pueden ser asignados a objetos de la clase ***Object*** y los métodos de ***Object*** pueden ser utilizados con ***arrays.***

Algunas de las características más importantes de los ***arrays*** son las siguientes:

1. Los ***arrays*** se crean con el operador ***new*** seguido del tipo y número de elementos.
2. Se puede acceder al número de elementos de un array con la variable miembro implícita ***length*** (por ejemplo, ***vect.length***).
3. Se accede a los elementos de un ***array*** con los ***corchetes []*** y un ***índice*** que varía de 0 a ***length-1***.
4. Se pueden crear ***arrays*** de objetos de cualquier tipo. En principio un ***array*** de objetos es un ***array de referencias*** que hay que completar llamando al operador ***new***.
5. Los elementos de un ***array*** se inicializan al valor por defecto del tipo correspondiente (cero para valores numéricos, el carácter nulo para ***char***, ***false*** para ***boolean***, ***null*** para ***Strings*** y para referencias).
6. Como todos los objetos, los ***arrays*** se pasan como argumentos a los métodos ***por referencia***.
7. Se pueden crear ***arrays anónimos*** (por ejemplo, crear un nuevo array como argumento actual en la llamada a un método).

Inicialización de ***arrays***:

1. Los ***arrays*** se pueden inicializar con valores entre llaves {...} separados por comas.
2. También los ***arrays de objetos*** se pueden inicializar con varias llamadas a ***new*** dentro de unas llaves {...}.
3. Si se igualan dos referencias a un array no se copia el array, sino que se tiene un array con dos nombres, apuntando al mismo y único objeto.
4. Creación de una ***referencia*** a un array. Son posibles dos formas:

```
double [] x; // preferible
double x [];
```

5. Creación del ***array*** con el operador ***new***:

```
x = new double[100];
```

6. Las dos etapas 4 y 5 se pueden unir en una sola:

```
double[] x = new double[100];
```

A continuación veremos algunos ejemplos de creación de arrays:

```
// crea un array de 10 enteros, que por defecto se inicializan a cero
int v[] = new int[10];

// otra forma de hacer esto mismo sería:
int [] v;              // define un array de enteros
v=new int[10];         // reserva memoria para 10 elementos
System.out.println("Longitud: " + v.length);  // Muestra la longitud del array
```

```
// array de 5 objetos
Punto listaObj[] = new Punto[5]; // de momento hay 5 referencias a null
for (int i = 0 ; i < 5;i++)
     listaObj[i] = new Punto(i,2);
System.out.print("Distancia entre el punto 2 y el 5: ");
System.out.println(listaObj[1].distancia(listaObj[4]));

// crear arrays inicializando con determinados valores
int [] v = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
String [] dias = {"lunes", "martes", "miercoles", "jueves",
                  "viernes", "sabado", "domingo"};
for (i=0; i<v.length; i++)   System.out.print("  " + v[i]);
for (i=0; i<dias.length; i++) System.out.print("  " + dias[i]);

// array anónimo
obj.metodo(new String[]={"uno", "dos", "tres"});
```

Veremos ahora el ejemplo anterior sobre agentes de ventas y comisiones, pero esta vez usando vectores y la sintaxis de Java.

```
double porcion =  2.0 / 3.0;

double [] ventas;
double suma=0.0, umbral;

ventas = new double[20];

for (int i=0; i<=19; i++) {
      ventas[i]=Leer.datoDouble();
      suma = suma + ventas[i];
}

umbral = porcion * (suma / 20.0);

for (int i=0; i<=19; i++)
  if (ventas[i]  > umbral)
      System.out.println(i+ " "+ ventas[i]);
```

En este último ejemplo si es posible que el número de agentes vaya a variar en algún momento de la vida de la empresa, es preferible definir una constante `NumeroAgentes=20` y esta se pone en los lugares del algoritmo en los que hemos puesto 20 o 19 (NumeroAgentes – 1). Con esto, al cambiar el número de agentes sólo hay que reflejarlo en la declaración de la constante y no en todos los sitios en que aparezca, como ocurrirá con el planteamiento anterior.

Veamos por último otro ejemplo que realice las siguientes operaciones:

1.- Lea la temperatura, al medio día, de cada día del mes, controlando los días de ese mes.
2.- Nos visualice por pantalla la temperatura media mensual
3.- Visualice por pantalla el día más frío y el más caluroso.

```
public static void main(String[] args) {
        // TODO Auto-generated method stub
        int temp[], dias, min = 50, max = 0, dfrio=0, dcalor=0;
        float media = 0;

        // Pedir memoria para el array
        temp = new int[31];

        System.out.print("Introduzca nº de días del mes: ");
        dias = Leer.datoInt();

        for (int cont = 0; cont < dias; cont++)
        {
           System.out.print("Introduzca temperatura del día " + (cont+1) + ": ");
           temp[cont]=Leer.datoInt();
           media = media + temp[cont];
           if (temp[cont] < min) {
                 min = temp[cont];
                 dfrio = cont;
              }
           if (temp[cont] > max){
                  max = temp[cont];
                  dcalor = cont;
              }
        } // end for

        System.out.println("La temperatura media es: " + media/dias);
        System.out.println("El día más caluroso fue: " + (dcalor + 1));
        System.out.println("El día más frío fue: " + (dfrio + 1));
 }
```

¿Crees que es una buena solución el planteamiento anterior? ¿Consideras correcto inicializar las variables min a 50 y max a 0? ¿Qué ocurriría si nos vamos a tomar las temperaturas al polo norte en pleno invierno? ¿Funcionaría en este último caso nuestro programa? ¿Cómo podríamos arreglarlo?

1.1. Algoritmos de búsqueda y ordenación

Existen multitud de algoritmos para manejar arrays, de entre ellos destacan por su importancia y frecuencia de utilización los de búsqueda y ordenación. Estos algoritmos son clásicos y su conocimiento es fundamental para cualquier programador.

A lo largo de la historia de la programación se han estudiado y propuesto diversos métodos. De todos ellos veremos los más importantes y también los más sencillos. La elección final de uno u otro método dependerá de las características particulares de un problema concreto y de sus datos.

1.1.1. Algoritmos de búsqueda

La localización y búsqueda de un elemento tiene como objetivo determinar si un elemento se encuentra en un conjunto de datos, en nuestro caso, un vector; en el caso de que se encuentre, el algoritmo dará como resultado la posición que ocupa.

Los algoritmos de búsqueda en arrays ordenados más utilizados son el de búsqueda lineal y el de búsqueda dicotómica. Si el array está desordenado, no hay más método de localización del elemento que la búsqueda secuencial o lineal del mismo.

Búsqueda lineal en un vector desordenado

Consiste en hacer un recorrido secuencial por el vector, comparando cada componente del vector con el elemento que se quiere encontrar. Hay dos condiciones que interrumpen la búsqueda:

- ✓ Se ha localizado el elemento que se buscaba.
- ✓ Se recorre todo el vector y no se ha encontrado el elemento.

Veamos como quedaría el algoritmo de búsqueda suponiendo que tenemos un array de k elementos llamado *vector* y el elemento que buscamos es *n*.

```
Procedimiento busqueda_lineal (entero vector[k], n)
Variables
     entero cont
Inicio
        cont = -1
        repite
            cont = cont + 1
        hasta ((vector[cont] == n) O (cont == k-1))


        si (vector[cont]==n) entonces
            escribe (“El elemento ocupa la posición: “ + cont)
        si-no
            escribe (“El elemento no existe”)
        fin-si
Fin
```

Ejercicio: Crear un vector con 20 números comprendidos entre 1 y 100 y escribir un método que nos busque un elemento dentro de dicho vector, utilizando búsqueda lineal.

Búsqueda lineal en un vector ordenado

Cuando el vector de búsqueda está ordenado se consigue un algoritmo más eficiente sin más que modificar la codificación de terminación en el algoritmo anterior. La ventaja que se obtiene es en el caso de orden ascendente, una vez sobrepasado el valor buscado, no es necesario recorrer el resto del vector para saber que el valor no existe.

El único cambio que habría que hacer al algoritmo anterior sería la condición de salida del bucle, que en este caso sería:

```
hasta (vector[cont] >= n) o (cont = k-1)
```

Búsqueda dicotómica o binaria

Este algoritmo es válido exclusivamente para vectores ordenados y consiste en comparar en primer lugar con la componente central[1] del vector, y si no es igual al valor buscado se reduce el intervalo de búsqueda a la mitad derecha o izquierda según donde pueda encontrarse el valor a buscar. El algoritmo termina si se encuentra el valor buscado o si el tamaño del intervalo de búsqueda queda anulado.

En los casos en que existan repeticiones en el vector, del valor buscado, este algoritmo obtendrá uno de ellos aleatoriamente según los lugares que ocupen, los cuales necesariamente son consecutivos.

El algoritmo de búsqueda binaria podría ser similar al que se muestra a continuación, suponiendo que tenemos un array de k elementos llamado *vector* y el elemento que buscamos es *n*.

[1] Existen algunas versiones, de este algoritmo de búsqueda, que comienzan por un componente elegido al azar, pero son menos eficientes.

```
Procedimiento busqueda_dicotomica(entero vector[k], n)
Variables
        entero mitad, primero, ultimo
        logico encontrado
Inicio
        encontrado = Falso
        primero = 0
        ultimo = k-1
        repite mientras ((encontrado == Falso) Y (primero <= ultimo))
            mitad = div((primero + ultimo), 2)
            si (vector[mitad]==n) entonces
                encontrado = Verdadero
            si-no
                si (vector[mitad] < n) entonces
                    primero = mitad + 1
                si-no
                    ultimo = mitad – 1
                fin-si
            fin-si
        fin-repite
        si (encontrado = Falso) entonces
            escribe (“El nº no existe”)
        si-no
         escribe (“La posición es: “, mitad)
        fin-si
Fin
```

1.1.2. Algoritmos de ordenación

La ordenación y clasificación de los elementos de un conjunto de datos depende del tipo de elementos que lo constituyen, de modo que si se trata de elementos literales, la ordenación podrá ser alfabética directa (de la A a la Z) o alfabética inversa (de la Z a la A). Si se trata de elementos numéricos, el orden puede ser creciente (los mayores valores en las posiciones de índice mayor, los últimos) o decreciente (los mayores al principio).

Los algoritmos de ordenación de elementos son numerosos y aumentan su complejidad a medida que aumenta el número de elementos, la velocidad de proceso necesaria y su eficacia.

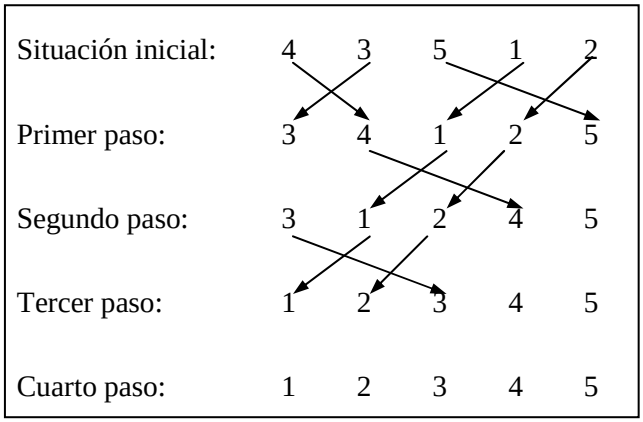
Los algoritmos que vamos a estudiar se aplican sobre vectores y al igual que los algoritmos de búsqueda, pueden generalizarse fácilmente para tratar con otras estructuras de datos. Nos centraremos en la ordenación ascendente.

Ordenación por intercambio (método de la burbuja)

El método de ordenación por intercambio es, probablemente, uno de los más empleados y se conoce también como **método de la burbuja**. Mediante este método la ordenación se obtiene tras el siguiente proceso:

1. Se debe recorrer el vector desde el primer elemento hasta el último (de izquierda a derecha), comparando elementos consecutivos entre sí e intercambiándolos en caso de encontrarse desordenados. El recorrido puede hacerse también a la inversa (de derecha a izquierda).
2. En cada recorrido se deja ordenado el mayor o el menor de los elementos tratados (según sea la ordenación). Si en el recorrido se ha realizado algún cambio, debe repetirse el proceso anterior.
3. Si no se ha realizado ningún cambio en el recorrido anterior, el proceso ha concluido.

El proceso descrito se muestra gráficamente en la siguiente figura:



Ordenación de un vector por el método de la burbuja

Con el método de la burbuja, para un vector de N elementos, el primer paso necesita recorrer todos los elementos, el segundo todos menos el último, el tercero todos menos el penúltimo, etc., lo que permite perfeccionar, ajustar y optimizar el algoritmo, si ello fuera necesario.

A continuación se muestra una posible versión del algoritmo de ordenación por el método de la burbuja, para un vector de K elementos:

```
Procedimiento burbuja (entero vector[K])
Variables
        entero i, j, temp
Inicio
        repite-para i=1, K-1
            repite-para j=K-1, i, -1
                si (vector[j] < vector[j-1]) entonces
                    temp = vector[j]
                    vector[j] = vector[j-1]
                    vector[j-1] = temp
                fin-si
            fin-repite
        fin-repite
Fin
```

La versión anterior se puede mejorar controlando que el algoritmo finalice en el momento en que detecte que en una vuelta no se ha intercambiado ningún valor, pudiendo quedar el algoritmo de la siguiente forma:

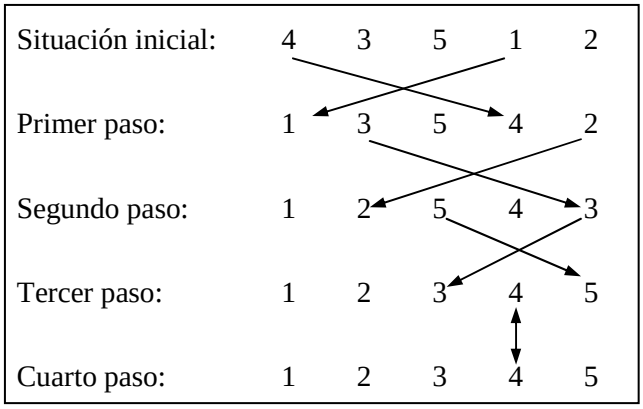
```
Procedimiento burbuja (entero vector[K])
Variables
        entero i, j, temp
        logico ordenado
Inicio
        ordenado=Falso
        i=1
        repite-mientras ((i<K) Y  (ordenado==Falso))
            ordenado = Verdadero
            repite-para j=K-1, i, -1
                si (vector[j] < vector[j-1]) entonces
                    temp = vector[j]
                    vector[j] = vector[j-1]
                    vector[j-1] = temp
                    ordenado = Falso
                fin-si
            fin-repite
            i=i+1
        fin-repite
Fin
```

Ordenación por selección

El método de ordenación de vectores por selección directa, o método de obtención sucesiva de menores, cumple el siguiente proceso:

1. Se debe recorrer la tabla tantas veces como elementos haya menos uno.
2. Se toma como origen de cada pasada el primer elemento no ordenado (en la primera pasada el 1º, en la segunda el 2º, etc.) y así se repetirá hasta el penúltimo inclusive.
3. Se busca el menor de entre todos los desordenados y, una vez localizado, se intercambio éste con el elemento origen de este recorrido.

El proceso descrito se muestra gráficamente a continuación:



Ordenación de un vector por el método de selección

A continuación se muestra una posible versión del algoritmo de ordenación por el método de selección, para un vector de K elementos:

```
Procedimiento selección (entero vector[K])
Variables
        entero i, j, pos_menor, menor
Inicio
        repite para i = 0, K-2
            pos_menor = i
            menor = vector[i]
            repite para j = i+1, K-1
                si (vector[j] < menor) entonces
                    pos_menor = j
                    menor = vector[j]
                fin_si
            fin-repite
            vector[pos_menor] = vector[i]
            vector[i] = menor
        fin_repite
Fin
```

Ordenación por inserción

El método de ordenación por inserción, también es conocido como método de la baraja, requiere un proceso como el siguiente:

Se debe recorrer la tabla tantas veces como elementos tiene menos uno.

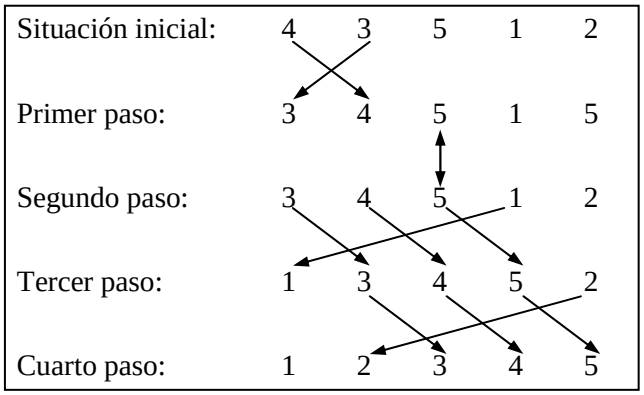
En la primera pasada se toma el segundo elemento y en las siguientes se toma como referencia los siguientes de forma sucesiva.

En cada pasada se saca el elemento seleccionado, se compara con todos los anteriores a él y, en caso de ser mayores, se van desplazando una posición.

El elemento tomado se inserta en el hueco que le corresponda según su valor.

Dicho en otras palabras, lo que hace es considerar que el primer elemento está ordenado y ordena el segundo respecto al primero, es decir, lo coloca delante o lo deja donde está según proceda. Luego ordena el tercer elemento respecto de los dos anteriores. Y así sucesivamente hasta el último elemento, que lo debe colocar en el lugar que le corresponda respecto a todos los demás.

El proceso descrito se muestra gráficamente a continuación:



Ordenación de un vector por el método de inserción

El algoritmo para ordenar un array de K elementos usando este método puede quedar de la siguiente forma:

```
Procedimiento inserción (entero vector[K])
Variables
        entero i, j, aux
Inicio
        repite para i=1, K-1
            aux = vector[i]
            j = i-1
            repite mientras ((j >= 0) Y (vector[j] > aux))
                  vector[j+1] = vector[j]
                  j = j-1
            fin-repite
            vector[j+1] = aux
        fin-repite
Fin
```

Ordenación por el método Shell

Está basado en la ordenación por inserción. Se ordenan los elementos que están separados por una distancia de comparación que en un principio puede ser la mitad de la longitud del array. En cada etapa esta distancia se va reduciendo hasta que sea menor que la unidad, en cuyo caso la lista ya estará ordenada. Si el número de elementos es impar, calcularemos la parte entera de su mitad. La comparación de los elementos que se encuentran a una distancia determinada debe estar realizándose mientras que haya cambios, es decir, no están ordenados los elementos respecto a ese salto.

La secuencia de salto 64, 31, 16, 8, 4, 2 y 1 es una secuencia muy mala, ya que los elementos en posiciones pares e impares no son comparados hasta el último momento. Una secuencia de salto bastante buena es: 121, 40, 13, 4 y 1.

Veamos como quedaría el algoritmo si tomamos como secuencia de salto inicialmente la mitad de la longitud del vector y que se va reduciendo a la mitad en cada repetición hasta que dicha distancia sea 1:

```
Procedimiento SHELL (entero vector[K])
Variables
        entero sw, i, salto, aux
Inicio
        n = K-1
        salto = n
        repite (mientras salto != 1)
           sw = 1
           salto = div(salto,2)
           repite mientras (sw!=0)
             i = 0
             sw = 0
             repite mientras (i <= (n-salto))
                si (vector[i] > vector[i+salto]) entonces
                    aux = vector[i+salto]
                    vector[i+salto] = vector[i]
                    vector[i]= aux
                    sw = 1
                fin-si
                i = i + 1
             fin-repite
           fin-repite
        fin-repite
Fin
```

1.2. Arrays bidimensionales o matrices.

No existen como tal en Java. En ***Java*** una ***matriz*** es un ***vector*** de ***vectores fila***, o más en concreto un vector de referencias a los vectores fila. Con este esquema, cada fila podría tener un número de elementos diferente.

Una matriz se puede crear directamente en la forma,

```
int [][] mat = new int[3][4];
```

o bien se puede crear de modo dinámico dando los siguientes pasos:

1. Crear la ***referencia*** indicando con un doble corchete que es una ***referencia a matriz***,

   ```
   int[][] mat;
   ```

2. Crear el vector de referencias a las filas,

   ```
   mat = new int[nfilas][];
   ```

3. Reservar memoria para los vectores correspondientes a las filas,

   ```
   for (int i=0; i<nfilas; i++)
       mat[i] = new int[ncols];
   ```

A continuación se presentan algunos ejemplos de creación de arrays bidimensionales:

```
// crear una matriz 3x3 se inicializan a cero
double mat[][] = new double[3][3];

// crear una matriz de 2x3
int [][] b = { {1, 2, 3},
             {4, 5, 6},   // esta coma es permitida
             };
```

En el caso de una matriz ***b***, ***b.length*** es el número de filas y ***b[0].length*** es el número de columnas (de la fila 0). Por supuesto, los arrays bidimensionales pueden contener tipos primitivos de cualquier tipo u objetos de cualquier clase.

```
// Crear una matriz de 3 filas y longitud diferente para cada columna
int [][] c = new int[3][];  // se crea el array de referencias a arrays
c[0] = new int[5];  // 1ª fila de 5 columnas
c[1] = new int[4];  // 2ª fila de 4 columnas
c[2] = new int[8];  // 3ª fila de 8 columnas
c[0][0]=1;
c[2][6]=9;
c[1][3]=5;
for (int i=0; i<c.length; i++)
    {
        for (int j=0; j<c[i].length; j++)
              System.out.print(c[i][j] + "  ");
        System.out.println(" ");
    }
```

En este último ejemplo mostrará por pantalla al ejecutarlo algo similar a:

---------- Interprete ----------
1 0 0 0 0
0 0 0 5
0 0 0 0 0 0 9 0
Normal Termination
Output completed (2 sec consumed).

1.3. La clase Arrays

En el paquete java.utils se encuentra una clase estática llamada Arrays. Una clase estática permite ser utilizada como si fuera un objeto (como ocurre con Math), es decir que para utilizar sus métodos hay que utilizar simplemente esta sintaxis:

Arrays.método(argumentos);

Algunos métodos interesantes de esta clase son:

- ***fill:*** Permite rellenar todo un array unidimensional con un determinado valor. Sus argumentos son el array a rellenar y el valor deseado:

```
int valores[]=new int[23];
Arrays.fill(valores,-1);//Todo el array vale -1
```

También permite decidir desde que índice hasta qué índice rellenamos:

```
Arrays.fill(valores,5,8,-1);//Del elemento 5 al 7 valdrán -1
```

- ***equals:*** compara dos arrays y devuelve true si son iguales. Se consideran iguales si son del mismo tipo, tamaño y contienen los mismos valores. A diferencia del operador de igualdad (==), este operador sí compara el contenido.

Ejemplo (comparación entre el operador de igualdad y el método equals):

```
int a[]= {2,3,4,5,6};
int b[]= {2,3,4,5,6};
int c[]=a;
System.out.println(a==b);      //false
System.out.println(Arrays.equals(a,b));  //true
System.out.println(a==c);      //true
System.out.println(Arrays.equals(a,c));  //true
```

- ***sort:*** Permite ordenar un array en orden ascendente. Se pueden ordenar todo el array o bien desde un elemento a otro:

```
int x[]={4,5,2,3,7,8,2,3,9,5};
Arrays.sort(x,2,5);  //El array queda {4 5 2 3 7 8 2 3 9 5}
Arrays.sort(x);   //Estará completamente ordenado
```

- ***binarySearch:*** Permite buscar un elemento de forma ultrarrápida en un array ordenado (en un array desordenado sus resultados son impredecibles). Devuelve el índice en el que está colocado el elemento. Ejemplo:

```
int x[]={1,2,3,4,5,6,7,8,9,10,11,12};
Arrays.sort(x);
System.out.println(Arrays.binarySearch(x,8));  //Escribe: 7
```

- ***copyOf:*** Disponible desde la versión 1.6 del SDK, obtiene una copia de un array. Recibe dos parámetros: el primero es el array a copiar y el segundo el tamaño que tendrá el array resultante. De modo que si el tamaño es menor que el del array original, sólo obtiene copia de los primeros elementos (tantos como indique el tamaño); si el tamaño es mayor que el original, devuelve un array en el que los elementos que superan al original se rellenan con ceros o con datos de tipo null (dependiendo del tipo de datos del array.

```
int a[] = {1,2,3,4,5,6,7,8,9};
int b[]=Arrays.copyOf(a, a.length);//b es {1,2,3,4,5,6,7,8,9}
int c[]=Arrays.copyOf(a, 12);      //c es {1,2,3,4,5,6,7,8,9,0,0,0}
int d[]=Arrays.copyOf(a, 3);       //d es {1,2,3}
```

- ***copyOfRange:*** Funciona como la anterior (también está disponible desde la versión 1.6), sólo que indica con dos números de qué elemento a qué elemento se hace la copia:

```
int a[] = {1,2,3,4,5,6,7,8,9};
int b[]=Arrays.copyOfRange(a, 3,6); //b vale {4,5,6}
```

1.4. El método System.arraysCopy

La clase System también posee un método relacionado con los arrays, dicho método permite copiar un array en otro. Recibe cinco argumentos: el array que se copia, el índice desde que se empieza a copia en el origen, el array destino de la copia, el índice desde el que se copia en el destino, y el tamaño de la copia (número de elementos de la copia).

```
int uno[]={1,1,2};
int dos[]={3,3,3,3,3,3,3,3,3};
System.arraycopy(uno, 0, dos, 0, uno.length);
for (int i=0;i<=8;i++){
    System.out.print(dos[i]+"  ");
} //Sale 1  1 2 3 3 3 3 3 3
```

La ventaja sobre el método copyOf de la clase Arrays está en que esté método funciona en cualquier versión de Java.

2. EL PAQUETE "java.lang"

Algunas de las clases que contiene este paquete son: *Object, System, Math, Character, String, StringBuffer* y las clases envoltorios de tipos básicos. Es el paquete por defecto, por tanto no es necesario importarlo para poderlo utilizar.

2.1. La clase Object

Es la clase superior de toda la jerarquía de clases de Java: si una definición de clase no extiende a otra, entonces extiende (hereda) a *Object*. Algunos de los métodos más importantes de esta clase son:

- *boolean equals(Object obj):* permite verificar si dos referencias se refieren a un mismo objeto.
- *String toString( ):* devuelve un String que almacena el nombre de la clase del objeto que recibe el mensaje *toString*.
- *void finalize( ):* este método es invocado por el *recolector de basura* cuando Java determina que no hay más referencias a un objeto. No ejecuta ninguna acción en especial; simplemente retorna normalmente. Las subclases de Object deberán sobreescribir la definición de este método sólo cuando se necesite ejecutar alguna operación de finalización especial.
- *Object clone( ):* permite duplicar un objeto.
- *wait, notify y notifyAll* soportan el control de hilos.

2.2. La clase System

Maneja particularidades del sistema. Dispone de tres variables de clase públicas:

- *System.**in**:* Referencia a la entrada estándar del sistema, que normalmente coincide con el teclado. Se utiliza para leer datos introducidos por el usuario
- *System.**out**:* Referencia a la salida estándar del sistema, que normalmente es el monitor. Se utiliza para mostrar datos al usuario.
- *System.**err**:* Referencia a la salida estándar de error del sistema, que normalmente es el monitor. Se utiliza para mostrar mensajes de error al usuario.

y de varios métodos de clase públicos, de entre los cuales destacan:

- *void exit(int).* Finaliza la ejecución actual de la JVM.
- *void gc( ).* Permite forzar una completa *recolección de basura*.
- *void runFinalizersOnExit(boolean).* Este método es inherentemente inseguro. Puede producir la finalización de objetos que están siendo manipulados por otros hilos, produciendo conductas erróneas o bloqueo.

2.3. La clase Math

La clase ***java.lang.Math*** deriva de ***Object***. Proporciona *métodos estáticos* para realizar las operaciones matemáticas habituales. Proporciona además las constantes: **E** y **PI**, cuyo significado no requiere muchas explicaciones.

Para utilizar cualquiera de estos métodos no hay que importar la clase *Math* puesto que pertenece al paquete *java.lang* (paquete que se carga por defecto):

Ejemplo:
```
System.out.println("Raíz cuadrada de " + x + "= " + Math.sqrt(x));
System.out.println(Math.abs(-5) * Math.PI);
```

La siguiente tabla muestra los métodos soportados por esta clase.

Métodos	Significado
double **ceil**(double x)	Redondea x al entero mayor siguiente: ♦ Math.ceil(2.8) vale 3 ♦ Math.ceil(2.4) vale 3 ♦ Math.ceil(-2.8) vale -2
double **floor**(double x)	Redondea x al entero menor siguiente: ♦ Math.floor(2.8) vale 2 ♦ Math. floor (2.4) vale 2 ♦ Math. floor (-2.8) vale -3
int **round**(double x)	Redondea x de forma clásica: ♦ Math.round(2.8) vale 3 ♦ Math. round (2.4) vale 2 ♦ Math. round (-2.8) vale -3
double **rint**(double x)	Idéntico al anterior, sólo que éste método da como resultado un número double mientras que round da como resultado un entero tipo int
double **random**()	Número aleatorio decimal situado entre el 0 y el 1
tiponúmero **abs**(tiponúmero x)	Devuelve el valor absoluto de x.
tiponúmero **min**(tiponúmero x, tiponúmero y)	Devuelve el menor valor de x o y
tiponúmero **max**(tiponúmero x, tiponúmero y)	Devuelve el mayor valor de x o y
double **sqrt**(double x)	Calcula la raíz cuadrada de x
double **pow**(double x, double y)	Calcula x^y
double **exp**(double x)	Calcula e^x
double **log**(double x)	Calcula el logaritmo neperiano de x
double **acos**(double x)	Calcula el arco coseno de x
double **asin**(double x)	Calcula el arco seno de x
double **atan**(double x)	Calcula el arco tangente de x
double **sin**(double x)	Calcula el seno de x, entre - PI/2 y PI/2
double **cos**(double x)	Calcula el coseno de x
double **tan**(double x)	Calcula la tangente de x
double **atan2**(double y, double x)	Arcotangente entre -PI y PI
double **toDegrees**(double anguloEnRadianes)	Convierte de radianes a grados
double **toRadians**(double anguloEnGrados)	Convierte de grados a radianes
double **signum**(double n)	Devuelve el valor del signo del número n. Si n vale cero, la función devuelve cero; si es positivo devuelve 1.0 y si es negativo -1.0. Esta función apareció en la versión 1.5 de Java.
double **hypot**(double x, double y)	Suponiendo que x e y son los dos catetos de un triángulo rectángulo, la función devuelve la hipotenusa correspondiente según el teorema de Pitágoras. Disponible desde la versión 1.5
double **nextAfter**(double valor, double dir)	Devuelve el siguiente número representable desde el valor indicado hacia la dirección que indique el valor del parámetro dir. Por ejemplo Math.nextAfter(34.7, 90) devolvería 34.7000000001 Función añadida en la versión Java 1.6
IEEEremainder(double,double)	Calcula el resto de la división

Una de las aplicaciones más interesantes de Math es la posibilidad de crear números aleatorios. Para ello se utiliza el método **random** que devuelve un número double entre cero y uno.

Para conseguir un número decimal por ejemplo entre cero y diez bastaría utilizar la expresión:

```
Math.random()*10
```

Y si queremos que sea un número entero entre 1 y 10, la expresión correcta es:

```
(int) Math.floor(Math.random()*10+1)
```

Entre 10 y 30 (inclusive) sería:

```
(int) Math.floor(Math.random()*21+10)
```

2.4. Las clases String y StringBuffer

Sirven para manipular cadenas de caracteres. La clase ***String*** está orientada a manejar cadenas de caracteres constantes, es decir, que no pueden cambiar. La clase ***StringBuffer*** permite que el programador cambie la cadena insertando, borrando, etc. La primera es más eficiente, mientras que la segunda permite más posibilidades.

Ambas clases pertenecen al package ***java.lang***, y por lo tanto no hay que importarlas. Hay que indicar que el ***operador de concatenación*** (+) entre objetos de tipo ***String*** utiliza internamente objetos de la clase ***StringBuffer*** y el método ***append()***.

Los métodos ***String*** se pueden utilizar directamente sobre *literales* (cadenas entre comillas), como por ejemplo: “Hola”.length();

2.4.1. Métodos de la clase String

Los objetos de la clase ***String*** se pueden crear a partir de cadenas constantes o *literales*, definidas entre dobles comillas, como por ejemplo: "Hola". ***Java*** crea siempre un objeto ***String*** al encontrar una cadena entre comillas. A continuación se describen dos formas de crear objetos de la clase ***String***,

```
String str1 = "Hola";              // el sistema más eficaz de crear Strings
String str2 = new String("Hola");  // también se pueden crear con un constructor
```

El primero de los métodos expuestos es el más eficiente, porque como al encontrar un texto entre comillas se crea automáticamente un objeto ***String***, en la práctica utilizando ***new*** se llama al constructor dos veces. También se pueden crear objetos de la clase ***String*** llamando a otros constructores de la clase, a partir de objetos ***StringBuffer***, y de arrays de ***bytes*** o de ***chars***.

Las cadenas pueden ocupar varias líneas utilizando el operador de concatenación “+”.

```
String texto2 = ”Este es un texto que ocupa “ +
                “varias líneas, no obstante se puede “+
                “perfectamente encadenar”;
```

También se pueden crear objetos String sin utilizar constantes entrecomilladas, usando otros constructores. Por ejemplo podemos utilizar un array de caracteres, como por ejemplo:

```
char[] palabra = {‘P’,’a’,’l’,'a','b’,’r’,’a’};//palabra es un array de caracteres
                                               //no es lo mismo que un String
```

a partir de ese array de caracteres podemos crear un String:

```
String cadena = new String(palabra); // mediante el operador new podemos convertir
                                     // el array de caracteres en un String
```

De forma similar hay posibilidad de convertir un array de bytes en un String. En este caso se tomará el array de bytes como si contuviera códigos de caracteres, que serán traducidos por su carácter correspondiente al ser convertido a String. Hay que indicar la tabla de códigos que se utilizará (si no se indica se entiende que utilizamos el código ASCII:

```
byte[] datos = {97,98,99};
String codificada = new String (datos, “8859_1”);
```

En el último ejemplo la cadena codificada se crea desde un array de tipo byte que contiene números que serán interpretados como códigos Unicode. Al asignar, el valor 8859_1 indica la tabla de códigos a utilizar.

La siguiente tabla muestra los métodos más importantes de la clase ***String***.

Métodos de String	**Función que realizan**
`String(...)`	Constructores para crear Strings a partir de arrays de bytes o de caracteres (ver documentación on-line)
`String(String str)` y `String(StringBuffer sb)`	Costructores a partir de un objeto String o StringBuffer
`charAt(int)`	Devuelve el carácter en la posición especificada
`concat()`	Concatena cadenas, aunque es más fácil usar el operador ‘+’.
`endsWitch(String)`	Indica si un String termina con otro String o no
`getChars(int, int, char[], int)`	Copia los caracteres indicados en la posición indicada de un array de caracteres
`indexOf(String, [int])`	Devuelve la posición en la que aparece por primera vez un String en otro String, a partir de una posición dada (opcional)
`lastIndexOf(String, [int])`	Devuelve la última vez que un String aparece en otro empezando en una posición y hacia el principio
`length()`	Devuelve el número de caracteres de la cadena
`matches(String)`	Examina la expresión regular que recibe como parámetro (en forma de String) y devuelve verdadero si el texto que examina cumple la expresión regular. (*Ver más información a continuación*).
`replace(char, char)`	Sustituye un carácter por otro en un String
`startsWith(String)`	Indica si un String comienza con otro String o no
`substring(int, int)`	Devuelve un String extraído de otro
`toCharArray()`	Consigue un array de caracteres a partir de una cadena. De esa forma podemos utilizar las características de los arrays para manipular el texto.
`toLowerCase()`	Convierte en minúsculas (puede tener en cuenta el locale)
`toUpperCase()`	Convierte en mayúsculas (puede tener en cuenta el locale)
`trim()`	Elimina los espacios en blanco al comienzo y final de la cadena
`valueOf()`	Devuelve la representación como String de sus argumentos. Admite Object, arrays de caracteres y los tipos primitivos

Un punto importante a tener en cuenta es que hay métodos, tales como ***System.out.println()***, que exigen que su argumento sea un objeto de la clase ***String***. Si no lo es, habrá que utilizar algún método que lo convierta en ***String***.

El ***locale*** citado en la tabla anterior, es la forma que java tiene para adaptarse a las peculiaridades de los idiomas distintos del inglés: acentos, caracteres especiales, forma de escribir las fechas y las horas, unidades monetarias, etc.

El método matches

El método ***matches***, es un método muy interesante disponible desde la versión 1.4. Examina la expresión regular que recibe como parámetro (en forma de String) y devuelve verdadero si el texto que examina cumple la expresión regular.

Una expresión regular es una expresión textual que utiliza símbolos especiales para hacer búsquedas avanzadas. Las expresiones regulares pueden contener:

- Caracteres. Como a, s, ñ,… y les interpreta tal cual. Si una expresión regular contuviera sólo un carácter, matches devolvería verdadero si el texto contiene sólo ese carácter. Si se ponen varios, obliga a que el texto tenga exactamente esos caracteres.
- Caracteres de control (\n,\\,….)
- Opciones de caracteres. Se ponen entre corchetes. Por ejemplo [abc] significa a, b ó c.
- Negación de caracteres. Funciona al revés impide que aparezcan los caracteres indicados. Se pone con corchetes dentro de los cuales se pone el carácter circunflejo (^). [^abc] significa ni a ni b ni c.
- Rangos. Se ponen con guiones. Por ejemplo [a-z] significa: cualquier carácter de la a a la z.
- Intersección. Usa &&. Por ejemplo [a-x&&r-z] significa de la r a la x (intersección de ambas expresiones.
- Substracción. Ejemplo [a-x&&[^cde]] significa de la a a la x excepto la c, d ó e.
- Cualquier carácter. Se hace con el símbolo punto (.)
- Opcional. El símbolo ? sirve para indicar que la expresión que le antecede puede aparecer una o ninguna veces. Por ejemplo a? indica que puede aparecer la letra a o no.
- Repetición. Se usa con el asterisco (*). Indica que la expresión puede repetirse varias veces o incluso no aparecer.
- Repetición obligada. Lo hace el signo +. La expresión se repite una o más veces (pero al menos una).
- Repetición un número exacto de veces. Un número entre llaves indica las veces que se repite la expresión. Por ejemplo \d{7} significa que el texto tiene que llevar siete números (siete cifras del 0 al 9). Con una coma significa al menos, es decir \d{7,} significa al menos siete veces (podría repetirse más veces). Si aparece un segundo número indica un máximo número de veces \d{7,10} significa de siete a diez veces.

Ejemplo, se trata de comprobar si el texto que se lee mediante teclado empieza con dos guiones, le siguen tres números y luego una o más letras mayúsculas. De no ser así, el programa vuelve a pedir escribir el texto:

```
public static void main(String[] args) {

    String respuesta;
    do {
      System.out.println("Escriba el texto: ");
      respuesta=Leer.dato();
      if(respuesta.matches("--[0-9]{3}[A-Z]+")==false) {
         System.out.println("La expresión no encaja con el patrón");
      }
    }while(respuesta.matches("--[0-9]{3}[A-Z]+")==false);
    System.out.println("Expresión correcta!" );

}
```

Comparación entre objetos String

Los objetos String (como ya ocurría con los arrays) no pueden compararse directamente con los operadores de comparación. En su lugar se deben utilizar estas expresiones:

- **cadena1.equals(cadena2)**. El resultado es true si la cadena1 es igual a la cadena2. Ambas cadenas son variables de tipo String.
- **cadena1.equalsIgnoreCase(cadena2)**. Como la anterior, pero en este caso no se tienen en cuenta, mayúsculas y minúsculas. Por cierto en cuestiones de mayúsculas y minúsculas, los hispanohablantes (y el resto de personas del planeta que utilice otro idioma distinto al inglés) podemos utilizar esta función sin temer que no sea capaz de manipular correctamente caracteres como la eñe, las tildes,… Funciona correctamente con cualquier símbolo alfabético de cualquier lengua presente en el código Unicode.
- **s1.compareTo(s2)**. Compara ambas cadenas, considerando el orden alfabético. Si la primera cadena es mayor en orden alfabético que la segunda devuelve 1, si son iguales devuelve 0 y si es la segunda la mayor devuelve -1. Hay que tener en cuenta que el orden no es el del alfabeto español, sino que usa la tabla ASCII, en esa tabla la letra ñ es mucho mayor que la o.
- **s1.compareToIgnoreCase(s2).** Igual que la anterior, sólo que además ignora las mayúsculas (disponible desde Java 1.2)

2.4.2. La clase StringBuffer

Sirve para manipular cadenas de caracteres permitiendo su actualización. La clase ***StringBuffer*** se utiliza prácticamente siempre que se desee modificar una cadena de caracteres. Completa los métodos de la clase ***String*** ya que éstos realizan sólo operaciones sobre el texto que no conllevan un aumento o disminución del número de letras del ***String***.

Recordemos que hay muchos métodos cuyos argumentos deben ser objetos String, que antes de pasar esos argumentos habrá que realizar la conversión correspondiente.

La siguiente tabla muestra los métodos más importantes de la clase ***StringBuffer***.

Métodos de StringBuffer	Función que realizan
`StringBuffer(), StringBuffer(int), StringBuffer(String)`	Constructores
`append(...)`	Tiene muchas definiciones diferentes para añadir un String o una variable (int, long, double, etc.) a su objeto
`capacity()`	Devuelve el espacio libre del StringBuffer
`charAt(int)`	Devuelve el carácter en la posición especificada
`getChars(int, int, char[], int)`	Copia los caracteres indicados en la posición indicada de un array de caracteres
`insert(int, )`	Inserta un String o un valor (int, long, double, ...) en la posición especificada de un StringBuffer
`length()`	Devuelve el número de caracteres de la cadena
`reverse()`	Cambia el orden de los caracteres
`setCharAt(int, char)`	Cambia el carácter en la posición indicada
`setLength(int)`	Cambia el tamaño de un StringBuffer
`toString()`	Convierte en objeto de tipo String

2.5. Las clases envoltorios (WRAPPERS).

Los ***Wrappers*** (*envoltorios*) son clases diseñadas para ser un ***complemento*** de los ***tipos primitivos***. En efecto, los tipos primitivos son los únicos elementos de ***Java*** que no son objetos. Esto tiene algunas ventajas desde el punto de vista de la ***eficiencia***, pero algunos inconvenientes desde el punto de vista de la ***funcionalidad***. Por ejemplo, los ***tipos primitivos*** siempre se pasan como argumento a los métodos ***por valor***, mientras que los ***objetos*** se pasan ***por referencia***. No hay forma de modificar en un método un argumento de tipo primitivo y que esa modificación se trasmita al entorno que hizo la llamada. Una forma de conseguir esto es utilizar un ***Wrapper***, esto es un objeto cuya variable miembro es el tipo primitivo que se quiere modificar. Las clases ***Wrapper*** también proporcionan métodos para realizar otras tareas con los tipos primitivos, tales como conversión con cadenas de caracteres en uno y otro sentido.

Existe una clase ***Wrapper*** para cada uno de los tipos primitivos numéricos, esto es, existen las clases ***Byte***, ***Short***, ***Integer***, ***Long***, ***Float*** y ***Double*** (obsérvese que los nombres empiezan por mayúscula, siguiendo la nomenclatura típica de ***Java***). A continuación se van a ver dos de estas clases: ***Double*** e ***Integer***. Las otras cuatro son similares y sus características pueden consultarse en la documentación on-line.

2.5.1 Clase Double

La clase ***java.lang.Double*** deriva de ***Number***, que a su vez deriva de ***Object***. Esta clase contiene un valor primitivo de tipo ***double***. La siguiente tabla muestra algunos métodos y constantes predefinidas de la clase ***Double***.

Métodos	Función que desempeñan
`Double(double) y Double(String)`	Los constructores de esta clase
`doubleValue(), floatValue(), longValue(), intValue(),shortValue(), byteValue()`	Métodos para obtener el valor del tipo primitivo
`String toString(), Double valueOf(String)`	Conversores con la clase String
`isInfinite(), isNaN()`	Métodos de chequear condiciones
`equals(Object)`	Compara con otro objeto
`MAX_DOUBLE,MIN_DOUBLE, POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN, TYPE`	Constantes predefinidas. TYPE es el objeto Class representando esta clase

El *Wrapper* ***Float*** es similar al *Wrapper* ***Double***.

2.5.2. Clase Integer

La clase ***java.lang.Integer*** tiene como variable miembro un valor de tipo ***int***. La siguiente tabla muestra los métodos y constantes de la clase ***Integer***.

Métodos	Función que desempeñan
`Integer(int) y Integer(String)`	Constructores de la clase
`doubleValue(), floatValue(), longValue(),intValue(), shortValue(), byteValue()`	Conversores con otros tipos primitivos
`Integer decode(String), Integer parseInt(String), String toString(), Integer valueOf(String)`	Conversores con String

`String toBinaryString(int),`
`String toHexString(int),`
`String toOctalString(int)`
	Conversores a cadenas representando enteros en otros sistemas de numeración

`Integer getInteger(String)`	Determina el valor de una propiedad del sistema a partir del nombre de dicha propiedad
`MAX_VALUE, MIN_VALUE, TYPE`	Constantes predefinidas

Los *Wrappers* ***Byte***, ***Short*** y ***Long*** son similares a ***Integer***.

Los *Wrappers* son utilizados para convertir cadenas de caracteres (texto) en números. Esto es útil cuando se leen valores desde el teclado, desde un fichero de texto, etc. Los ejemplos siguientes muestran algunas conversiones:

```
String numDecimalString = "8.978";
float numFloat=Float.valueOf(numDecimalString).floatValue();  // numFloat=8.979
double numDouble=Double.valueOf(numDecimalString).doubleValue(); //numDouble=8.979

String numIntString = "1001";
int numInt=Integer.valueOf(numIntString).intValue();   //numInt=1001
```

En el caso de que el texto no se pueda convertir directamente al tipo especificado se lanza una excepción de tipo ***NumberFormatException***, por ejemplo si se intenta convertir directamente el texto "4.897" a un número entero. El proceso que habrá que seguir será convertirlo en primer lugar a un número ***float*** y posteriormente a número entero.

3. EL PAQUETE "java.util"

Contiene clases de utilidad, como son: las colecciones, la clase StringTokenizer, la clase Random, clases para el manejo de fechas y horas (Calendar, Date, ...).

3.1. Colecciones

Java dispone también de clases e interfaces para trabajar con colecciones de objetos. Entre otras tenemos las clases ***Vector, Stack*** y ***Hashtable***, así como la interface ***Enumeration***.

3.1.1. *Clase Vector*

La clase ***java.util.Vector*** deriva de ***Object***, implementa ***Cloneable*** (para poder sacar copias con el método ***clone( )***) y ***Serializable*** (para poder ser convertida en cadena de caracteres).

Como su mismo nombre sugiere, ***Vector*** representa un ***array de objectos*** (referencias a objetos de tipo ***Object***) que puede crecer y reducirse, según el número de elementos de forma dinámica. Además permite acceder a los elementos con un ***índice***, aunque no permite utilizar los corchetes [].

El método ***capacity()*** devuelve el tamaño o número de elementos que puede tener el vector. El método ***size()*** devuelve el número de elementos que realmente contiene, mientras que ***capacityIncrement*** es una variable que indica el salto que se dará en el tamaño cuando se necesite crecer. La siguiente tabla muestra los métodos más importantes de la clase ***Vector***. Puede verse que el gran número de métodos que existen proporciona una notable flexibilidad en la utilización de esta clase.

Además de ***capacityIncrement***, existen otras dos variables miembro: ***elementCount***, que representa el número de componentes válidos del vector, y ***elementData[]*** que es el array de ***Objects*** donde realmente se guardan los elementos del objeto ***Vector*** (***capacity*** es el tamaño de este array). Las tres variables citadas son ***protected***.

Métodos	Función que realizan
`Vector(), Vector(int), Vector(int, int)`	Constructores que crean un vector vacío, un vector de la capacidad indicada y un vector de la capacidad e incremento indicados
`void add(Object obj)`	Añade un objeto al final
`boolean removeElement(Object obj)`	Elimina el primer objeto que encuentra como su argumento y desplaza los restantes. Si no lo encuentra devuelve false
`void removeAllElements()`	Elimina todos los elementos
`Object clone()`	Devuelve una copia del vector
`void copyInto(Object anArray[])`	Copia un vector en un array
`void trimToSize()`	Ajusta el tamaño a los elementos que tiene
`void setSize(int newSize)`	Establece un nuevo tamaño
`int capacity()`	Devuelve el tamaño (capacidad) del vector
`int size()`	Devuelve el número de elementos
`boolean isEmpty()`	Devuelve true si no tiene elementos
`Enumeration elements()`	Devuelve una Enumeración con los elementos
`boolean contains(Object elem)`	Indica si contiene o no un objeto
`int indexOf(Object elem, int index)`	Devuelve la posición de la primera vez que aparece un objeto a partir de una posición dada
`int lastIndexOf(Object elem, int index)`	Devuelve la posición de la última vez que aparece un objeto a partir de una posición, hacia atrás
`Object elementAt(int index)`	Devuelve el objeto en una determinada posición
`Object firstElement()`	Devuelve el primer elemento
`Object lastElement()`	Devuelve el último elemento
`void setElementAt(Object obj, int index)`	Cambia el elemento que está en una determinada posición
`void removeElementAt(int index)`	Elimina el elemento que está en una determinada posición
`void insertElementAt(Object obj, int index)`	Inserta un elemento por delante de una determinada posición

3.1.2. *Interface Enumeration.*

La interface ***java.util.Enumeration*** define métodos útiles para recorrer una colección de objetos. Puede haber distintas clases que implementen esta interface y todas tendrán un comportamiento similar.

La interface ***Enumeration*** declara dos métodos:

1. **`public boolean hasMoreElements()`**. Indica si hay más elementos en la colección o si se ha llegado ya al final.
2. **`public Object nextElement()`**. Devuelve el siguiente objeto de la colección. Lanza una *NoSuchElementException* si se llama y ya no hay más elementos.

Todas las colecciones implementan el método ***Enumeration elements()*** que devuelve una Enumeración con los elementos de la colección correspondiente.

Ejemplo: Para imprimir los elementos de un vector ***vec*** se pueden utilizar las siguientes sentencias:

```
for (Enumeration e = vec.elements(); e.hasMoreElements(); ) {
     System.out.println(e.nextElement());
}
```

donde, como puede verse en la tabla de los métodos de la clase Vector, el método ***elements( )*** devuelve precisamente una referencia de tipo ***Enumeration***. Con los métodos ***hasMoreElements( )*** y ***nextElement( )*** y un bucle ***for*** se pueden ir imprimiendo los distintos elementos del objeto ***Vector***.

3.1.3. *Clase Hashtable.*

La clase ***java.util.Hashtable*** extiende ***Dictionary*** (***abstract***) e implementa ***Cloneable*** y ***Serializable***. Una ***hashtable*** es una tabla que relaciona una ***clave*** con un ***valor***. Cualquier objeto distinto de ***null*** puede ser tanto ***clave*** como ***valor***.

La clase a la que pertenecen las ***claves*** debe implementar los métodos ***hashCode()*** y ***equals()***, con objeto de hacer búsquedas y comparaciones. El método ***hashCode()*** devuelve un entero único y distinto para cada clave, que es siempre el mismo en una ejecución del programa pero que puede cambiar de una ejecución a otra. Además, para dos claves que resultan iguales según el método ***equals()***, el método ***hashCode()*** devuelve el mismo entero.

Cada objeto de ***hashtable*** tiene dos variables: ***capacity*** y ***load factor*** (entre 0.0 y 1.0). Cuando el número de elementos excede el producto de estas variables, la ***hashtable*** crece llamando al método ***rehash()***. Un ***load factor*** más grande apura más la memoria, pero será menos eficiente en las búsquedas. Es conveniente partir de una ***hashtable*** suficientemente grande para no estar ampliando continuamente.

Ejemplo de definición de ***hashtable***:

```
class PruebaArray
{
           static public void main(String args[])
             {
               Hashtable<String,String> h = new Hashtable<String,String>();
               String s;
               h.put("casa", "home");
               h.put("copa", "cup");
               h.put("lápiz", "pen");
               s=(String) h.get("copa");
               System.out.println(h.get("copa"));
               System.out.println(h);
             }
}

Mostrará la salida siguiente:

---------- Interprete ----------
cup
{copa=cup, lápiz=pen, casa=home}
Normal Termination
Output completed (7 sec consumed).
```

donde se ha hecho uso del método ***put()***. La tabla siguiente muestra los métodos de la clase ***Hashtable***.

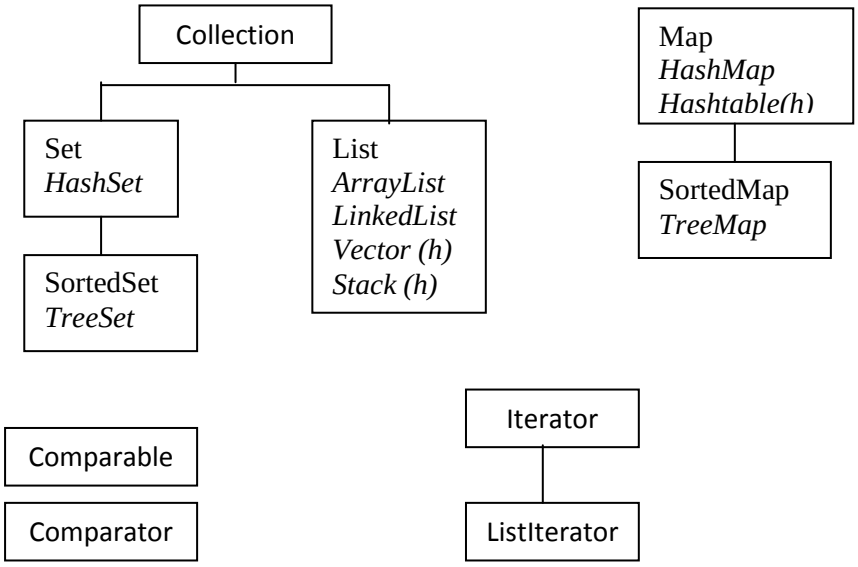
Métodos	Función que realizan
`Hashtable(), Hashtable(int nElements), Hashtable(int nElements, float loadFactor)`	Constructores
`int size()`	Devuelve el tamaño de la tabla
`Boolean isEmpty()`	Indica si la tabla está vacía
`Enumeration keys()`	Devuelve una Enumeration con las claves
`Enumeration elements()`	Devuelve una Enumeration con los valores
`Boolean contains(Object value)`	Indica si hay alguna clave que se corresponde con el

	valor
`Boolean containsKey(Object key)`	Indica si existe esa clave en la tabla
`Object get(Object key)`	Devuelve un valor dada la clave
`void rehash()`	Amplía la capacidad de la tabla
`Object put(Object key, Object value)`	Establece una relación clave-valor
`Object remove(Object key)`	Elimina un valor por la clave
`void clear()`	Limpia la tabla
`Object clone()`	Hace una copia de la tabla
`String toString()`	Devuelve un string representando la tabla

3.1.4. *El Collections Framework*

A partir de la versión 1.2 del JDK se introdujo el ***Java Framework Collections*** o "estructura de colecciones de Java" (en adelante JCF). Se trata de un conjunto de clases e interfaces que mejoran notablemente las capacidades del lenguaje respecto a estructuras de datos. Además, constituyen un excelente ejemplo de aplicación de los conceptos propios de la *programación orientada a objetos*. Dada la amplitud de ***Java*** en éste y en otros aspectos se va a optar por insistir en la descripción general, dejando al lector la tarea de buscar las características concretas de los distintos métodos en la documentación de ***Java***.

La siguiente figura muestra la jerarquía de interfaces de la ***Java Collection Framework*** (JCF). En letra cursiva se indican las clases que implementan las correspondientes interfaces. Por ejemplo, hay dos clases que implementan la interface ***Map***: ***HashMap*** y ***Hashtable***.



Interfaces de la Collection Framework

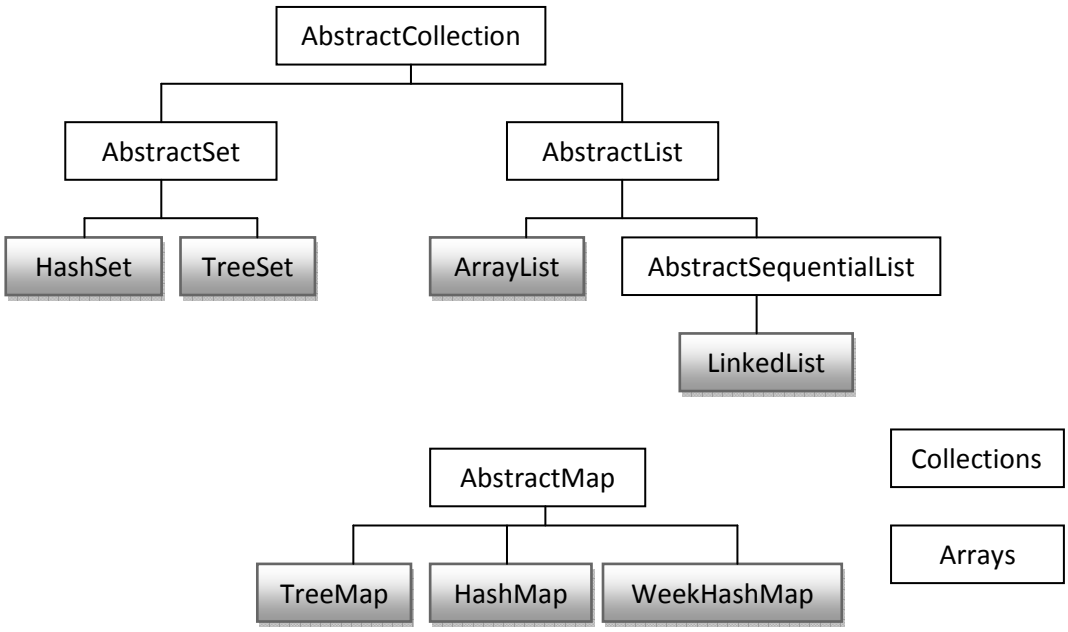
Las clases vistas en los apartados anteriores son clases "históricas", es decir, clases que existían antes de la versión JDK 1.2. Dichas clases se denotan en la figura anterior con la letra "h" entre paréntesis. Aunque dichas clases se han mantenido por motivos de compatibilidad, sus métodos no siguen las reglas del diseño general del JCF; en la medida de lo posible se recomienda utilizar las nuevas clases.

En el diseño de la JCF las ***interfaces*** son muy importantes porque son ellas las que determinan las capacidades de las clases que las implementan. Dos clases que implementan la misma interface se pueden utilizar exactamente de la misma forma. Por ejemplo, las clases ***ArrayList*** y ***LinkedList*** disponen exactamente de los mismos métodos y se pueden utilizar de la misma forma. La diferencia está en la implementación: mientras que ***ArrayList*** almacena los objetos en un array, la clase ***LinkedList*** los almacena

en una lista vinculada. La primera será más eficiente para acceder a un elemento arbitrario, mientras que la segunda será más flexible si se desea borrar e insertar elementos.

En la siguiente página se muestra la figura con la jerarquía de clases de la JCF. En este caso, la jerarquía de clases es menos importante desde el punto de vista del usuario que la jerarquía de interfaces. En dicha figura se muestran con fondo blanco las clases abstractas, y con fondo gris claro las clases de las que se pueden crear objetos.

Las clases ***Collections*** y ***Arrays*** son un poco especiales: no son ***abstract***, pero no tienen constructores públicos con los que se puedan crear objetos. Fundamentalmente contienen métodos ***static*** para realizar ciertas operaciones de utilidad: ordenar, buscar, introducir ciertas características en objetos de otras clases, etc.



Jerarquía de clases de la Collection Framework

3.1.4.1. Elementos del Java Collections Framework

Interfaces de la JCF: Constituyen el elemento central de la JCF.

- ***Collection***: define métodos para tratar una colección genérica de elementos
- ***Set***: colección que no admite elementos repetidos
- ***SortedSet***: *set* cuyos elementos se mantienen ordenados según el criterio establecido
- ***List***: admite elementos repetidos y mantiene un orden inicial
- ***Map***: conjunto de pares clave/valor, sin repetición de claves
- ***SortedMap***: *map* cuyos elementos se mantienen ordenados según el criterio establecido

Interfaces de soporte:

- ***Iterator***: sustituye a la interface ***Enumeration***. Dispone de métodos para recorrer una colección y para borrar elementos.
- ***ListIterator***: deriva de ***Iterator*** y permite recorrer *lists* en ambos sentidos.
- ***Comparable***: declara el método ***compareTo()*** que permite ordenar las distintas colecciones según un orden natural (***String***, ***Date***, ***Integer***, ***Double***, …).
- ***Comparator***: declara el método ***compare()*** y se utiliza en lugar de ***Comparable*** cuando se desea ordenar objetos no estándar o sustituir a dicha interface.

Clases de propósito general: Son las implementaciones de las interfaces de la JFC.

- ***HashSet***: Interface ***Set*** implementada mediante una hash table.
- ***TreeSet***: Interface ***SortedSet*** implementada mediante un árbol binario ordenado.
- ***ArrayList***: Interface ***List*** implementada mediante un array.
- ***LinkedList***: Interface ***List*** implementada mediante una lista vinculada.
- ***HashMap***: Interface ***Map*** implementada mediante una hash table.
- ***WeakHashMap***: Interface ***Map*** implementada de modo que la memoria de los pares clave/valor pueda ser liberada cuando las claves no tengan referencia desde el exterior de la ***WeakHashMap***.
- ***TreeMap***: Interface ***SortedMap*** implementada mediante un árbol binario

Clases Wrapper: Colecciones con características adicionales, como no poder ser modificadas o estar sincronizadas. No se accede a ellas mediante constructores, sino mediante métodos "factory" de la clase ***Collections***.

Clases de utilidad: Son mini-implementaciones que permiten obtener ***sets*** especializados, como por ejemplo ***sets*** constantes de un sólo elemento (***singleton***) o ***lists*** con ***n*** copias del mismo elemento (***nCopies***). Definen las constantes EMPTY_SET y EMPTY_LIST. Se accede a través de la clase ***Collections***.

Clases históricas: Son las clases ***Vector*** y ***Hashtable*** presentes desde las primeras versiones de ***Java***. En las versiones actuales, implementan respectivamente las interfaces ***List*** y ***Map***, aunque conservan también los métodos anteriores.

Clases abstractas: Tienen total o parcialmente implementados los métodos de la interface correspondiente. Sirven para que los usuarios deriven de ellas sus propias clases con un mínimo de esfuerzo de programación.

Algoritmos: La clase ***Collections*** dispone de métodos ***static*** para ordenar, desordenar, invertir orden, realizar búsquedas, llenar, copiar, hallar el mínimo y hallar el máximo.

Clase Arrays: Es una clase de utilidad introducida a partir del JDK 1.2 que contiene métodos ***static*** para ordenar, llenar, realizar búsquedas y comparar los arrays clásicos del lenguaje. Permite también ver los ***arrays*** como ***lists***.

Después de esta visión general de la *Java Collections Framework*, se verán algunos detalles de las clases e interfaces más importantes.

3.1.4.2. Interface Collection

La interface ***Collection*** es implementada por los ***conjuntos*** (*sets*) y las ***listas*** (*lists*). Esta interface declara una serie de métodos generales utilizables con ***Sets*** y ***Lists***. La declaración o header de dichos métodos se puede ver ejecutando el comando > ***javap java.util.Collection*** en una ventana de MS-DOS. El resultado se muestra a continuación:

```
public interface java.util.Collection
{
    public abstract boolean add(java.lang.Object); // opcional
    public abstract boolean addAll(java.util.Collection); // opcional
    public abstract void clear(); // opcional
    public abstract boolean contains(java.lang.Object);
    public abstract boolean containsAll(java.util.Collection);
    public abstract boolean equals(java.lang.Object);
    public abstract int hashCode();
    public abstract boolean isEmpty();
    public abstract java.util.Iterator iterator();
    public abstract boolean remove(java.lang.Object); // opcional
    public abstract boolean removeAll(java.util.Collection); // opcional
    public abstract boolean retainAll(java.util.Collection); // opcional
    public abstract int size();
    public abstract java.lang.Object toArray()[];
```

```
        public abstract java.lang.Object toArray(java.lang.Object[])[];
    }
```

A partir del nombre, de los argumentos y del valor de retorno, la mayor parte de estos métodos resultan autoexplicativos. A continuación se introducen algunos comentarios sobre los aspectos que pueden resultar más novedosos de estos métodos. Los detalles se pueden consultar en la documentación de ***Java***.

Los métodos indicados como "// opcional" pueden no estar disponibles en algunas implementaciones, como por ejemplo en las clases que no permiten modificar sus objetos. Por supuesto dichos métodos deben ser definidos, pero lo que hacen al ser llamados es lanzar una ***UnsupportedOperationException***.

El método ***add()*** trata de añadir un objeto a una colección, pero puede que no lo consiga si la colección es un ***set*** que ya tiene ese elemento. Devuelve ***true*** si el método ha llegado a modificar la colección. Lo mismo sucede con ***addAll()***. El método ***remove()*** elimina un único elemento (si lo encuentra), y devuelve ***true*** si la colección ha sido modificada.

El método ***iterator()*** devuelve una referencia ***Iterator*** que permite recorrer una colección con los métodos ***next()*** y ***hasNext()***. Permite también borrar el elemento actual con ***remove()***.

Los dos métodos ***toArray()*** permiten convertir una colección en un array.

3.1.4.3. Interfaces Iterator y ListIterator

La interface ***Iterator*** sustituye a ***Enumeration***, utilizada en versiones anteriores del JDK. Dispone de los métodos siguientes:

```
Compiled from Iterator.java
public interface java.util.Iterator
{
    public abstract boolean hasNext();
    public abstract java.lang.Object next();
    public abstract void remove();
}
```

El método ***remove()*** permite borrar el último elemento accedido con ***next()***. Es la única forma segura de eliminar un elemento mientras se está recorriendo una colección.

Los métodos de la interface ***ListIterator*** son los siguientes:

```
Compiled from ListIterator.java
public interface java.util.ListIterator extends java.util.Iterator
{
    public abstract void add(java.lang.Object);
    public abstract boolean hasNext();
    public abstract boolean hasPrevious();
    public abstract java.lang.Object next();
    public abstract int nextIndex();
    public abstract java.lang.Object previous();
    public abstract int previousIndex();
    public abstract void remove();
    public abstract void set(java.lang.Object);
}
```

La interface ***ListIterator*** permite recorrer una lista en ambas direcciones, y hacer algunas modificaciones mientras se recorre. Los elementos se numeran desde 0 a *n-1*, pero los valores válidos para el índice son de 0 a *n*. Puede suponerse que el índice *i* está en la frontera entre los elementos *i-1* e *i*; en ese caso ***previousIndex***() devolvería *i-1* y ***nextIndex()*** devolvería *i*. Si el índice es 0, ***previousIndex()*** devuelve –1 y si el índice es *n* ***nextIndex()*** devuelve el resultado de ***size()***.

3.1.4.4. Interfaces Comparable y Comparator

Estas interfaces están orientadas a mantener ordenadas las ***listas***, y también los ***sets*** y ***maps*** que deben mantener un orden. Para ello se dispone de las interfaces ***java.lang.Comparable*** y ***java.util.Comparator*** (obsérvese que pertenecen a packages diferentes).

La interface ***Comparable*** declara el método ***compareTo()*** de la siguiente forma:

```
public int compareTo(Object obj)
```

que compara su argumento implícito con el que se le pasa por ventana. Este método devuelve un entero ***negativo***, ***cero*** o ***positivo*** según el argumento implícito (***this***) sea ***anterior***, ***igual*** o ***posterior*** al objeto ***obj***. Las listas de objetos de clases que implementan esta interface tienen un ***orden natural***. Esta interface está implementada -entre otras- por las clases ***String***, ***Character***, ***Date***, ***File***, ***BigDecimal***, ***BigInteger***, ***Byte***, ***Short***, ***Integer***, ***Long***, ***Float*** y ***Double***. Téngase en cuenta que la implementación estándar de estas clases no asegura un orden alfabético correcto con mayúsculas y minúsculas, y tampoco en idiomas distintos del inglés.

Si se redefine, el método ***compareTo()*** debe ser programado con cuidado: es muy conveniente que sea coherente con el método ***equals()*** y que cumpla la ***propiedad transitiva***. Para más información, consultar la documentación.

Las ***listas*** y los ***arrays*** cuyos elementos implementan ***Comparable*** pueden ser ordenadas con los métodos static ***Collections.sort()*** y ***Arrays.sort()***.

La interface ***Comparator*** permite ordenar listas y colecciones cuyos objetos pertenecen a clases de tipo cualquiera. Esta interface permitiría por ejemplo ordenar figuras geométricas planas por el área o el perímetro. Su papel es similar al de la interface ***Comparable***, pero el usuario debe siempre proporcionar una implementación de esta interface. Sus dos métodos se declaran en la forma:

```
public int compare(Object o1, Object o2)
public boolean equals(Object obj)
```

El objetivo del método ***equals()*** es comparar ***Comparators***.

El método ***compare()*** devuelve un entero ***negativo***, ***cero*** o ***positivo*** según su primer argumento sea ***anterior***, ***igual*** o ***posterior*** al segundo. Los objetos que implementan ***Comparator*** pueden pasarse como argumentos al método ***Collections.sort()*** o a algunos ***constructores*** de las clases ***TreeSet*** y ***TreeMap***, con la idea de que las mantengan ordenadas de acuerdo con dicho ***Comparator***. Es muy importante que ***compare()*** sea compatible con el método ***equals()*** de los objetos que hay que mantener ordenados. Su implementación debe cumplir unas condiciones similares a las de ***compareTo()***.

Java dispone de clases capaces de ordenar cadenas de texto en diferentes lenguajes. Para ello se puede consultar la documentación sobre las clases ***CollationKey***, ***Collator*** y sus clases derivadas, en el package ***java.text***.

3.1.4.5. Sets y SortedSets

La interface ***Set*** sirve para acceder a una colección sin elementos repetidos. La colección puede estar o no ordenada (con un orden natural o definido por el usuario, se entiende). La interface ***Set*** no declara ningún método adicional a los de ***Collection***.

Como un ***Set*** no admite elementos repetidos es importante saber cuándo dos objetos son considerados iguales (por ejemplo, el usuario puede o no desear que las palabras ***Mesa*** y ***mesa*** sean consideradas iguales). Para ello se dispone de los métodos ***equals()*** y ***hashcode()***, que el usuario puede redefinir si lo desea.

Utilizando los métodos de ***Collection***, los ***Sets*** permiten realizar operaciones algebraicas de ***unión***, ***intersección*** y ***diferencia***. Por ejemplo, ***s1.containsAll(s2)*** permite saber si ***s2*** está contenido en ***s1***; ***s1.addAll(s2)*** permite convertir ***s1*** en la unión de los dos conjuntos; ***s1.retainAll(s2)*** permite convertir ***s1*** en la intersección de ***s1*** y ***s2***; finalmente, ***s1.removeAll(s2)*** convierte ***s1*** en la diferencia entre ***s1*** y ***s2***.

La interface ***SortedSet*** extiende la interface ***Set*** y añade los siguientes métodos:

```
Compiled from SortedSet.java
public interface java.util.SortedSet extends java.util.Set
{
    public abstract java.util.Comparator comparator();
    public abstract java.lang.Object first();
    public abstract java.util.SortedSet headSet(java.lang.Object);
    public abstract java.lang.Object last();
    public abstract java.util.SortedSet subSet(java.lang.Object, java.lang.Object);
    public abstract java.util.SortedSet tailSet(java.lang.Object);
}
```

que están orientados a trabajar con el "orden". El método ***comparator()*** permite obtener el objeto pasado al constructor para establecer el orden. Si se ha utilizado el orden natural definido por la interface ***Comparable***, este método devuelve ***null***. Los métodos ***first()*** y ***last()*** devuelven el primer y último elemento del conjunto. Los métodos ***headSet()***, ***subSet()*** y ***tailSet()*** sirven para obtener subconjuntos al principio, en medio y al final del conjunto original (los dos primeros no incluyen el límite superior especificado).

Existes dos implementaciones de conjuntos: la clase ***HashSet*** implementa la interface ***Set***, mientras que la clase ***TreeSet*** implementa ***SortedSet***. La primera está basada en una hash table y la segunda en un ***TreeMap***.

Los elementos de un ***HashSet*** no mantienen el orden natural, ni el orden de introducción. Los elementos de un ***TreeSet*** mantienen el orden natural o el especificado por la interface ***Comparator***. Ambas clases definen constructores que admiten como argumento un objeto ***Collection***, lo cual permite convertir un ***HashSet*** en un ***TreeSet*** y viceversa.

3.1.4.6. Listas

La interface ***List*** define métodos para operar con colecciones ordenadas y que pueden tener elementos repetidos. Por ello, dicha interface declara métodos adicionales que tienen que ver con el orden y con el acceso a elementos o rangos de elementos. Además de los métodos de ***Collection***, la interface ***List*** declara los métodos siguientes:

```
Compiled from List.java
public interface java.util.List extends java.util.Collection
{
    public abstract void add(int, java.lang.Object);
    public abstract boolean addAll(int, java.util.Collection);
    public abstract java.lang.Object get(int);
    public abstract int indexOf(java.lang.Object);
    public abstract int lastIndexOf(java.lang.Object);
    public abstract java.util.ListIterator listIterator();
    public abstract java.util.ListIterator listIterator(int);
    public abstract java.lang.Object remove(int);
    public abstract java.lang.Object set(int, java.lang.Object);
    public abstract java.util.List subList(int, int);
}
```

Los nuevos métodos ***add()*** y ***addAll()*** tienen un argumento adicional para insertar elementos en una posición determinada, desplazando el elemento que estaba en esa posición y los siguientes. Los métodos ***get()*** y ***set()*** permiten obtener y cambiar el elemento en una posición dada. Los métodos ***indexOf()*** y ***lastIndexOf()*** permiten saber la posición de la primera o la última vez que un elemento aparece en la lista; si el elemento no se encuentra se devuelve –1.

El método ***subList(int fromIndex, toIndex)*** devuelve una "vista" de la lista, desde el elemento ***fromIndex*** inclusive hasta el ***toIndex*** exclusive. Un cambio en esta "vista" se refleja en la lista original, aunque no conviene hacer cambios simultáneamente en ambas. Lo mejor es eliminar la "vista" cuando ya no se necesita.

Existen dos implementaciones de la interface ***List***, que son las clases ***ArrayList*** y ***LinkedList***. La diferencia está en que la primera almacena los elementos de la colección en un ***array*** de ***Objects***, mientras

que la segunda los almacena en una ***lista vinculada***. Los ***arrays*** proporcionan una forma de acceder a los elementos mucho más eficiente que las listas vinculadas. Sin embargo tienen dificultades para crecer (hay que reservar memoria nueva, copiar los elementos del array antiguo y liberar la memoria) y para insertar y/o borrar elementos (hay que desplazar en un sentido u en otro los elementos que están detrás del elemento borrado o insertado). Las ***listas vinculadas*** sólo permiten acceso secuencial, pero tienen una gran flexibilidad para crecer, para borrar y para insertar elementos. El optar por una implementación u otra depende del caso concreto de que se trate.

3.1.4.7. Maps y SortedMaps

Un ***Map*** es una estructura de datos agrupados en parejas ***clave/valor***. Pueden ser considerados como una tabla de dos columnas. La ***clave*** debe ser única y se utiliza para acceder al ***valor***.

Aunque la interface ***Map*** no deriva de ***Collection***, es posible ver los ***Maps*** como colecciones de ***claves***, de ***valores*** o de parejas ***clave/valor***. A continuación se muestran los métodos de la interface ***Map*** (comando > ***javap java.util.Map***):

```
Compiled from Map.java
public interface java.util.Map
{
    public abstract void clear();
    public abstract boolean containsKey(java.lang.Object);
    public abstract boolean containsValue(java.lang.Object);
    public abstract java.util.Set entrySet();
    public abstract boolean equals(java.lang.Object);
    public abstract java.lang.Object get(java.lang.Object);
    public abstract int hashCode();
    public abstract boolean isEmpty();
    public abstract java.util.Set keySet();
    public abstract java.lang.Object put(java.lang.Object, java.lang.Object);
    public abstract void putAll(java.util.Map);
    public abstract java.lang.Object remove(java.lang.Object);
    public abstract int size();
    public abstract java.util.Collection values();
    public static interface java.util.Map.Entry
    {
         public abstract boolean equals(java.lang.Object);
         public abstract java.lang.Object getKey();
         public abstract java.lang.Object getValue();
         public abstract int hashCode();
         public abstract java.lang.Object setValue(java.lang.Object);
    }
}
```

Muchos de estos métodos tienen un significado evidente, pero otros no tanto. El método ***entrySet()*** devuelve una "vista" del ***Map*** como ***Set***. Los elementos de este ***Set*** son referencias de la interface ***Map.Entry***, que es una ***interface interna*** de ***Map***. Esta "vista" del ***Map*** como ***Set*** permite modificar y eliminar elementos del ***Map***, pero no añadir nuevos elementos.

El método ***get(key)*** permite obtener el valor a partir de la clave. El método ***keySet()*** devuelve una "vista" de las claves como ***Set***. El método ***values()*** devuelve una "vista" de los valores del ***Ma***p como ***Collection*** (porque puede haber elementos repetidos). El método ***put()*** permite añadir una pareja clave/valor, mientras que ***putAll()*** vuelca todos los elementos de un ***Map*** en otro ***Map*** (los pares con clave nueva se añaden; en los pares con clave ya existente los valores nuevos sustituyen a los antiguos). El método ***remove()*** elimina una pareja clave/valor a partir de la clave.

La interface ***SortedMap*** añade los siguientes métodos, similares a los de ***SortedSet***:

```
Compiled from SortedMap.java
public interface java.util.SortedMap extends java.util.Map
{
    public abstract java.util.Comparator comparator();
    public abstract java.lang.Object firstKey();
    public abstract java.util.SortedMap headMap(java.lang.Object);
    public abstract java.lang.Object lastKey();
    public abstract java.util.SortedMap subMap(java.lang.Object, java.lang.Object);
    public abstract java.util.SortedMap tailMap(java.lang.Object);
```

```
}
```

La clase ***HashMap*** implementa la interface ***Map*** y está basada en una hash table, mientras que ***TreeMap*** implementa ***SortedMap*** y está basada en un árbol binario.

3.1.4.8. Algoritmos y otras características especiales: Clases Collections y Arrays

La clase ***Collections*** (no confundir con la interface ***Collection***, en singular) es una clase que define un buen número de métodos static con diversas finalidades. No se detallan o enumeran aquí porque exceden del espacio disponible. Los más interesantes son los siguientes:

- *Métodos que definen algoritmos:*

  Ordenación mediante el método mergesort
  ```
  public static void sort(java.util.List);
  public static void sort(java.util.List, java.util.Comparator);
  ```

  Eliminación del orden de modo aleatorio
  ```
  public static void shuffle(java.util.List);
  public static void shuffle(java.util.List, java.util.Random);
  ```

  Inversión del orden establecido
  ```
  public static void reverse(java.util.List);
  ```

  Búsqueda en una lista
  ```
  public static int binarySearch(java.util.List, java.lang.Object);
  public static int binarySearch(java.util.List, java.lang.Object,
  java.util.Comparator);
  ```

  Copiar una lista o reemplazar todos los elementos con el elemento especificado
  ```
  public static void copy(java.util.List, java.util.List);
  public static void fill(java.util.List, java.lang.Object);
  ```

  Cálculo de máximos y mínimos
  ```
  public static java.lang.Object max(java.util.Collection);
  public static java.lang.Object max(java.util.Collection, java.util.Comparator);
  public static java.lang.Object min(java.util.Collection);
  public static java.lang.Object min(java.util.Collection, java.util.Comparator);
  ```

- *Métodos de utilidad:*

  Set inmutable de un único elemento
  ```
  public static java.util.Set singleton(java.lang.Object);
  ```

  Lista inmutable con n copias de un objeto
  ```
  public static java.util.List nCopies(int, java.lang.Object);
  ```

  Constantes para representar el conjunto y la lista vacía
  ```
  public static final java.util.Set EMPTY_SET;
  public static final java.util.List EMPTY_LIST;
  ```

Además, la clase ***Collections*** dispone de dos conjuntos de métodos "factory" que pueden ser utilizados para convertir objetos de distintas colecciones en objetos "***read only***" y para convertir distintas colecciones en objetos "***synchronized***" (por defecto las clases vistas anteriormente no están sincronizadas), lo cual quiere decir que se puede acceder a la colección desde distintas ***threads*** sin que se produzcan problemas. Los métodos correspondientes son los siguientes:

```
public static java.util.Collection synchronizedCollection(java.util.Collection);
public static java.util.List synchronizedList(java.util.List);
public static java.util.Map synchronizedMap(java.util.Map);
public static java.util.Set synchronizedSet(java.util.Set);
public static java.util.SortedMap synchronizedSortedMap(java.util.SortedMap);
public static java.util.SortedSet synchronizedSortedSet(java.util.SortedSet);
public static java.util.Collection unmodifiableCollection(java.util.Collection);
public static java.util.List unmodifiableList(java.util.List);
```

```
public static java.util.Map unmodifiableMap(java.util.Map);
public static java.util.Set unmodifiableSet(java.util.Set);
public static java.util.SortedMap unmodifiableSortedMap(java.util.SortedMap);
public static java.util.SortedSet unmodifiableSortedSet(java.util.SortedSet);
```

Estos métodos se utilizan de una forma muy sencilla: se les pasa como argumento una referencia a un objeto que no cumple la característica deseada y se obtiene como valor de retorno una referencia a un objeto que sí la cumple.

3.1.4.9. Desarrollo de clases por el usuario: Clases abstract

Las clases ***abstract*** indicadas en la figura de la página 13, pueden servir como base para que los programadores, con necesidades no cubiertas por las clases vistas anteriormente, desarrollen sus propias clases.

3.1.4.10. Interfaces Cloneable y Serializable

Las clases ***HashSet***, ***TreeSet***, ***ArrayList***, ***LinkedList***, ***HashMap*** y ***TreeMap*** (al igual que ***Vector*** y ***Hashtable***) implementan las interfaces ***Cloneable*** y ***Serializable***, lo cual quiere decir que es correcto sacar copias bit a bit de sus objetos con el método ***Object.clone()***, y que se pueden convertir en cadenas o flujos (***streams***) de caracteres.

Una de las ventajas de implementar la interface ***Serializable*** es que los objetos de estas clases pueden ser impresos con los métodos ***System.Out.print()*** y ***System.Out.println()***.

3.2. Novedades a partir de la versión 1.5 en las Colecciones

- ***Tipos de datos parametrizados (*generics*)***

Esta mejora permite tener colecciones de tipos concretos de datos, lo que permite asegurar que los datos que se van a almacenar van a ser compatibles con un determinado tipo o tipos. Por ejemplo, podemos crear un *Vector* que sólo almacene *Strings*, o una *HashMap* que tome como claves *Integers* y como valores *Vectors*. Además, con esto nos ahorramos las conversiones *cast* al tipo que deseemos, puesto que la colección ya se asume que será de dicho tipo.

Ejemplo:

```
// Vector de cadenas
// Antes
Vector v = new Vector();  // No se especifica el tipo de los datos que va a contener

// Ahora
Vector<String> v = new Vector<String>();
v.add("Hola");
String s = v.elementAt(0);
v.add(new Integer(20));      // Daría error!!, no coincide el tipo


// HashMap con claves enteras y valores de vectores
// Antes
HashMap hm = new HashMap();

// Ahora
HashMap<Integer, Vector> hm = new HashMap<Integer, Vector>();
hm.put(1, v);
Vector v2 = hm.get(1);
```

- ***Autoboxing***

Esta nueva característica evita al programador tener que establecer correspondencias manuales entre los tipos simples (*int*, *double*, etc) y sus correspondientes *wrappers* o tipos complejos (*Integer*, *Double*, etc). Podremos utilizar un *int* donde se espere un objeto complejo (*Integer*), y viceversa.

Ejemplo:

```
Vector<Integer> v = new Vector<Integer>();
v.add(30);   // Le añade un tipo simple en lugar del objeto
Integer n = v.elementAt(0);
n = n+1;
```

- ***Mejoras en bucles***

Se mejoran las posibilidades de recorrer colecciones y arrays, previniendo índices fuera de rango, y pudiendo recorrer colecciones sin necesidad de acceder a sus iteradores (*Iterator*).

Ejemplo:

```
// Recorre e imprime todos los elementos de un array
int[] arrayInt = {1, 20, 30, 2, 3, 5};
for(int elemento: arrayInt)
    System.out.println (elemento);

// Recorre e imprime todos los elementos de un Vector
Vector<String> v = new Vector<String>();
for(String cadena: v)
    System.out.println (cadena);
```

3.3. La clase StringTokenizer.

Esta clase divide una cadena en una serie de cadenas menores llamadas “tokens”. Por ejemplo al dividir la cadena "Tecnológico de Monterrey Campus Querétaro" por palabras tendríamos cinco tokens: "Tecnológico", "de", "Monterrey", "Campus" y "Queretaro". Fíjate que el StringTokenizer por omisión no devuelve más que las palabras divididas por el separador, sin devolver el separador (en este caso serían los espacios).

Una cadena se divide en tokens al aplicar algún tipo de carácter o caracteres como delimitador (por defecto es el espacio en blanco). Por ejemplo, el texto “12/05/52”, podría dividirse en tres tokens, 12, 05 y 52, usando el carácter barra (“/”) como delimitador.

En el constructor se proporcionan la cadena y opcionalmente, los delimitadores. Luego se maneja a través de los métodos.

- `boolean hasMoreTokens()`
- `String nextToken()`

Ejemplo:

```
import java.util.StringTokenizer;
class ST {
     static public void main (String arg [])
      {
        StringTokenizer st=new StringTokenizer(“Esto es una prueba de StringTokenizer”);
        while ( st.hasMoreTokens())
            System.out.println(st.nextToken());
      }
}
```

La salida del anterior ejemplo será:

```
---------- Interprete ----------
Esto
es
```

```
una
prueba
de
StringTokenizer
Normal Termination
Output completed (7 sec consumed).
```

3.4. Clase Random

Permite generar números aleatorios de diversas formas, para ello dispone de los métodos ***nextFloat, nextDouble, nextGaussian, nextInt***, etc. Para mayor información consúltese la documentación de ***Java***.

El siguiente ejemplo, genera números aleatorios entre 0 y 4.

```
import java.util.Random;
class Aleatorio
{
   public static void main(String[] args)
   {
        Random r = new Random();
        int i = r.nextInt(5);  // Genera numeros aleatorios entre 0 y 4
        System.out.println("Valor: " + i);
   }
}
```

3.5.- Clases para manejo de fechas y horas

Ver archivo “Tema_08_Anexo_fechas_y_horas”.