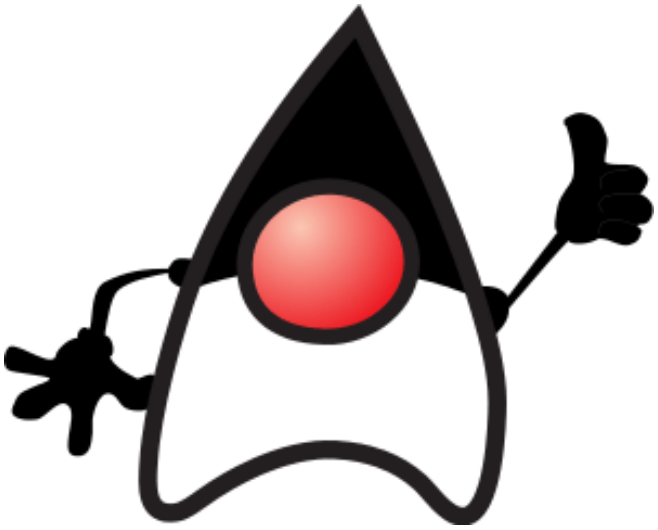
ESTRUCTURAS DE ALMACENAMIENTO



3

Contenido

1.- Introducción.

Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables. Pero, ¿qué ocurre cuando en una aplicación tenemos que gestionar varios pedidos a la vez?. Para poder realizar ciertas aplicaciones se necesita poder manejar datos que van más allá de meros **datos simples** (números y letras). A veces, los datos que tiene que manejar la aplicación son **datos compuestos**, es decir, datos que están compuestos a su vez de varios datos más simples. Por ejemplo, un pedido está compuesto por varios datos, los datos podrían ser el cliente que hace el pedido, la dirección de entrega, la fecha requerida de entrega y los artículos del pedido.

Los datos compuestos son un tipo de estructura de datos. **Las clases son un ejemplo de estructuras de datos que permiten almacenar datos compuestos**, y el objeto en sí, la instancia de una clase, sería el dato compuesto. Pero, a veces, los datos tienen estructuras aún más complejas, y son necesarias soluciones adicionales. Esas soluciones consisten básicamente en la capacidad de poder manejar varios datos del mismo o diferente tipo de forma dinámica y flexible.

¿Cómo almacenarías en memoria un listado de números del que tienes que extraer el valor máximo? Seguro que te resultaría fácil. Pero, ¿y si el listado de números no tiene un tamaño fijo, sino que puede variar en tamaño de forma dinámica? Entonces la cosa se complica. Un listado de números que aumenta o decrece en tamaño es una de las cosas que aprenderás a utilizar aquí, utilizando estructuras de datos.

Pasaremos por alto de momento las clases y los objetos, pero debes saber que las clases en sí mismas son la evolución de un tipo de estructuras de datos conocidas como datos compuestos (también llamadas *registros*). Las clases, además de aportar la ventaja de agrupar datos relacionados entre sí en una misma estructura (característica aportada por los datos compuestos), permiten agregar métodos que manejen dichos datos, ofreciendo una herramienta de programación sin igual.

Las estructuras de almacenamiento, en general, **se pueden clasificar de varias formas**. Por ejemplo, atendiendo a si pueden almacenar datos de diferente tipo, o si solo pueden almacenar datos de un solo tipo, se pueden distinguir:

- **Estructuras con capacidad de almacenar varios datos del mismo tipo**: varios números, varios caracteres, etc. Ejemplos de estas estructuras son los ***arrays***, las ***cadenas de caracteres***, las ***listas*** y los conjuntos.
- **Estructuras con capacidad de almacenar varios datos de distinto tipo**: números, fechas, cadenas de caracteres, etc., todo junto dentro de una misma estructura. Ejemplos de este tipo de estructuras son las ***clases***.

Otra forma de clasificar las estructuras de almacenamiento va en función de si pueden o no cambiar de tamaño de forma dinámica:

- **Estructuras cuyo tamaño se establece en el momento de la creación** o definición y su tamaño no puede variar después. Ejemplos de estas estructuras son los ***arrays*** y las ***matrices*** (arrays multimensionales).
- **Estructuras cuyo tamaño es variable (conocidas como estructuras dinámicas)**. Su tamaño crece o decrece según las necesidades de forma dinámica. Es el caso de las **listas**, ***árboles***, conjuntos y, como veremos también, el caso de algunos tipos de cadenas de caracteres.

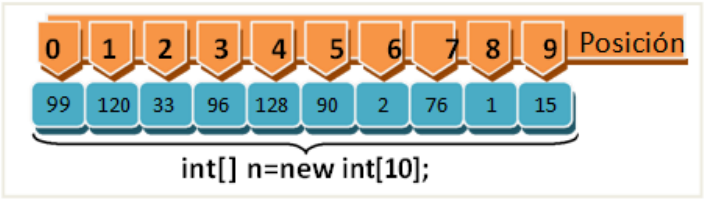
Por último, atendiendo a la forma en la que los datos se ordenan dentro de la estructura, podemos diferenciar varios tipos de estructuras:

- **Estructuras que no se ordenan de por sí**, y debe ser el programador el encargado de ordenar los datos si fuera necesario. Un ejemplo de estas estructuras son los ***arrays***.
- **Estructuras ordenadas**. Se trata de estructuras que al incorporar un dato nuevo a todos los datos

existentes, este se almacena en una posición concreta que irá en función del orden. El orden establecido en la estructura puede variar dependiendo de las necesidades del programa: alfabético, orden numérico de mayor a menor, momento de inserción, etc.

2.- Creación de arrays.

Un **array** es un grupo de variables (llamadas elementos o componentes) que contienen valores todos del mismo tipo. Cada uno de los elementos del array tiene asignado un índice numérico según su posición, siendo 0 el índice del primero y el tamaño del array menos 1 el del último. Los arrays son objetos, por lo que se consideran como tipos de referencia.



2.1.- Declaración.

Un array debe declararse utilizando la expresión:

tipo[ ] variable_array;

o

tipo variable_array[ ];

Como se puede apreciar, los corchetes pueden estar situados delante de la variable o detrás. No se puede especificar el número de elementos en los corchetes de la declaración.

Los siguientes son ejemplos de declaraciones de array:

```
int[ ] k;
String[ ] p;
char cads[ ];
```

Los arrays pueden declararse en los mismos lugares que las variables estándar: como atributos de una clase o locales en el interior de un método. Como ocurre con cualquier otro tipo de variable de instancia (atributo o campo), cuando un array se declara como atributo se inicializa implícitamente y en este caso al valor *null*.

2.2.- Dimensionado de un array.

Para asignar un tamaño al array se utiliza la expresión:

variable_array = new tipo[tamaño];

También se puede asignar tamaño al array en la misma línea de declaración de la variable. Los siguientes son ejemplos de dimensionado de un array:

```
k = new   int[5];
cads = new  char[10];
p = new String[5];
String[ ]   noms = new  String[10];
int[ ] a ={3, 5, 1, 7};
```

Cuando un array se dimensiona, todos sus elementos son inicializados implícitamente al valor por defecto del tipo correspondiente, independientemente de que la variable que contiene al array sea atributo o local.

Una vez dimensionado el array, éste no puede cambiar de tamaño.

Existe una forma de declarar, dimensionar e inicializar un array en una misma sentencia. La siguiente instrucción crea un array de cuatro enteros y los inicializa a los valores indicados entre llaves:

```
int[ ] nums = {10, 20, 30, 40};
```

2.3.- Acceso a los elementos de un array.

El acceso a los elementos de un array se realiza utilizando la expresión:

variable_array [índice]

Donde *índice* representa la posición a la que se quiere tener acceso, y cuyo valor puede estar comprendido entre 0 y tamaño-1. Un índice debe ser un valor *int* o un valor de un tipo que pueda promoverse a un *int*, pero no puede ser *long*.

Java no permite acceder fuera de los límites de un array. Cuando se ejecuta el programa, la JVM comprueba los índices del array para asegurarse de que sean válidos, es decir mayores a 0 y menores que el tamaño del array. Así, si en un programa se utiliza un índice inválido, Java genera una excepción para indicar que se produjo un error en el programa en tiempo de ejecución.

Todos los objetos array tienen un atributo público, llamado **length**, que permite conocer el tamaño al que ha sido dimensionado un array. Este atributo resulta especialmente útil en aquellos métodos que necesitan recorrer todos los elementos de un array, independientemente de su tamaño.

El siguiente bloque de código utiliza *length* para recorrer un array y rellenarlo con números enteros pares consecutivos, empezando por el 0.

```
int[ ]   nums = new   int[10];
for (int i = 0 ; i < nums.length ; i++)
{
        nums[i] = i*2;
}
```

El programa que se muestra a continuación calcula la suma de todos los números almacenados en un array de enteros, mostrando además el mayor y el menor de los números contenidos:

```
public class Principal{
        public static void main (String[ ] args){
                int mayor, menor, suma;
                int[ ] nums = {3,4,8,2};
                suma = 0;
                menor = nums[0] ;
                mayor = nums[0];
                for (int i = 0 ; i < nums.length ; i++){
                        if (nums[i] > mayor)
                                mayor = nums[i];
                        else
                                if (nums[i] < menor)
                                        menor = nums[i];
                        suma+= nums[i];
                }
                System.out.println ("El mayor es "+ mayor);
                System.out.println ("El menor es "+ menor);
                System.out.println ("La suma es "+ suma);
        }
}
```

Vamos a ver una nueva versión del programa anterior utilizando bucles *for mejorados* sin índices en vez de *for* tradicionales:

```
public class Principal{
        public static void main (String[ ] args){
                int mayor, menor, suma;
                int[ ] nums = {3,4,8,2};
                //Llamada a los métodos
                suma = sumar(nums);
                menor = calculoMenor(nums);
                mayor = calculoMayor(nums);
                System.out.println ("El mayor es "+ mayor);
                System.out.println ("El menor es "+ menor);
                System.out.println ("La suma es "+ suma);
        }
```

```
        static  int  sumar(int  numeros[ ]){
                int s = 0;
                for (int n:numeros){
                        s+=n;
                }
                return  s;
        }

        static  int  calculoMayor(int  numeros[ ]){
                int  may = numeros[0];
                for (int n:numeros){
                        if (n > may)
                                may = n;
                }
                return  may;
        }

        static  int  calculoMenor(int  numeros[ ]){
                int  men = numeros[0];
                for (int  n:numeros){
                        if (n < men)
                                men = n;
                }
                return  men;
        }
}
```

2.4.- Paso de un array como argumento de llamada a un método.

Los arrays también pueden utilizarse como argumentos de llamada a métodos. Para declarar un método que reciba como parámetro un array se emplea la sintaxis:

```
tipo  método(tipo  variable_array[ ])  {
        //código del método
}
```

El siguiente método recibe como parámetro un array de enteros:

```
void metodoEjemplo( int m[ ] ){
…
}
```

En cuanto a la llamada, se utilizará la expresión:

método(variable_array);

Nota importante: Si dentro del método se cambia algún valor del array enviado como parámetro, éste permanece modificado cuando se regresa del método. Ejemplo:

```
public class Principal{
        public static void main (String [ ] args){
                int mayor, menor, suma;
                int[ ] nums = {3,4,8,2};
                //Llamada a los métodos
                suma = sumar(nums);
                menor = calculoMenor(nums);
                mayor = calculoMayor(nums);
                System.out.println ("El mayor es "+ mayor);
                System.out.println ("El menor es "+ menor);
                System.out.println ("La suma es "+ suma);
                System.out.println ("primer elemento cambiado a "+ nums[0]);
        }

        static int sumar(int numeros[ ]){
                int s = 0;
                for (int i = 0;i < numeros.length;i++)
                        s+= numeros[i] ;
                numeros[0] = 0;
                return s;
        }

        static int calculoMayor(int numeros[ ]){
                int may = numeros[0];
                for (int i = 0; i < numeros.length;i++)
                {
                        if (numeros[i] > may)
                                may = numeros[i];
                }
                return may;
        }
        static int calculoMenor(int numeros[ ]){
                int men = numeros[0];
                for (int i = 0;i < numeros.length;i ++){
                        if (numeros[i] < men)
                                men = numeros[i];
                        }
                return men;
        }
}
```

Dando como resultado la siguiente visualización en pantalla:

El mayor es 8

El menor es 0

La suma es 17

primer elemento cambiado a 0

2.5.- Array como tipo de devolución de un método.

También un array puede ser devuelto por un método tras la ejecución del mismo. El formato de un método de esas características será el siguiente:

```
tipo[ ] método(parámetros){
        …
        return variable_array;
}
```

El siguiente método devuelve un array con los cinco números enteros siguientes al número recibido como parámetro:

```
int[ ] getNumeros(int n){
        int[ ] nums = new int[5];
        for (int i = 0 ; i < nums . length; i++)
                nums[i] = n+i+1;
        return nums;
}
```

3.- Búsqueda en arrays.

3.1.- Búsqueda secuencial o lineal.

En un array desordenado sería el único método de búsqueda, consiste en examinar el elemento de índice 0 y los elementos sucesivos hasta que se encuentra el valor buscado o no quedan más elementos que examinar.

En un array ordenado se optimiza la búsqueda finalizando en el momento en que se encuentra un valor mayor al buscado, lo que indicaría que dicho valor ya no existe.

```
public static int busquedaSecuencial(int [ ] arreglo, int dato){
        int posicion = -1;
        for(int i = 0 ; i < arreglo.length ; i++){      // recorremos el array
                if(arreglo[i] == dato){             // comparamos el elemento con el buscado
                        posicion = i;               // si es verdadero guardamos la posición
                        break;                      // salimos del bucle
                }
        }
        return posicion;
}
```

3.2.- Búsqueda binaria o dicotómica.

Sólo se aplica a arrays ordenados, consiste en dividir el array en sección inferior y superior, calculando el índice central del array. Si el dato se encuentra en ese elemento, la búsqueda binaria termina. Si el dato es numéricamente menor que el dato del elemento central, la búsqueda binaria calcula el índice central de la mitad inferior del array, ignorando la sección superior y repite el proceso. La búsqueda continua hasta que se encuentre el dato o se exceda el límite de la sección (lo que indica que el dato no existe en el array).

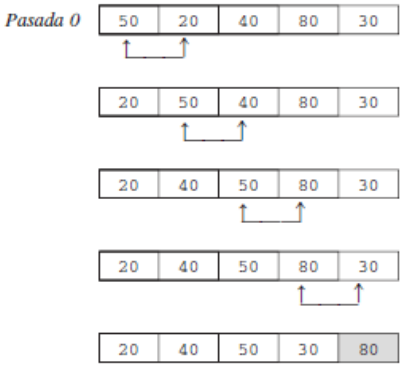
```
public static int busquedaBinaria(int vector[ ], int dato){
        int n = vector.length;
        int centro, inf = 0, sup = n-1;
                while(inf <= sup){
                        centro = (sup + inf) / 2;
                        if(vector[centro] == dato)
                                return centro;
                        else
                                if(dato < vector [centro] ){
                                        sup = centro-1;
                                }
                                else {
                                    inf = centro+1;
                                }
                }
        return -1;
}
```

4.- Ordenamiento de arrays.

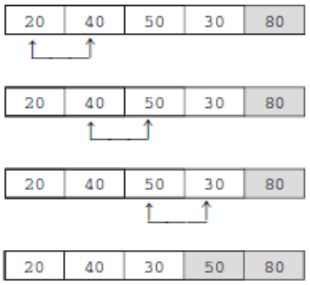
Haciendo un resumen de los métodos de ordenación de arrays (sin entrar en detalles) podemos decir que se clasifican en 3 tipos: métodos de **Inserción**, métodos de **Selección** y métodos de **Intercambio**. Dentro de los métodos de inserción se encuentran los métodos de inserción directa e inserción binaria o dicotómica. Entre los métodos de selección se encuentra el método de selección directa y entre los métodos de intercambio nos encontramos con los métodos de la *burbuja*, la sacudida y *quickSort*. De los métodos citados el peor método que hay con diferencia para ordenar arrays es el de la burbuja (el más conocido) y el mejor el quickSort.

4.1.- Burbuja.

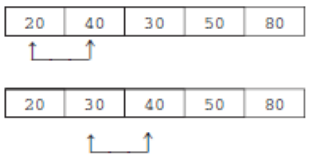
Consiste en comparar el primer elemento con el segundo, si el segundo es menor que el primero se intercambian los valores. Después el segundo con el tercero y así sucesivamente, cuando no haya ningún intercambio, el array estará ordenado:



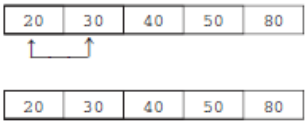
En la pasada 1:



En la pasada 2, sólo se hacen dos comparacione



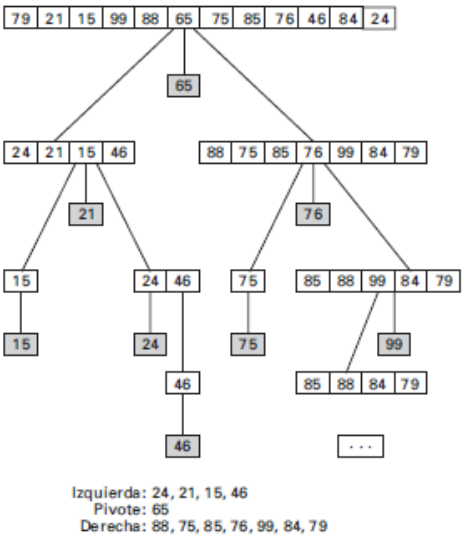
En la pasada 3, se hace una única comparación de 20 y 30, y no se produce intercambio:



```
public  static  void   burbuja (int lista[ ]){
      int   cuentaintercambios = 0;
      for (boolean  ordenado = false ; !ordenado ;   ){            // saldrá cuando esté ordenado
         for (int  i =0 ; i<lista.length-1 ; i++){
            if (lista[i] > lista[i+1]){
               //Intercambiamos valores
               int  variableauxiliar = lista[i];
               lista[i] = lista[i+1];
               lista[i+1] = variableauxiliar;
                           cuentaintercambios++;                  //indicamos que hay un cambio
            }
         }
         //Si no hay intercambios, es que está ordenado
         if (cuentaintercambios == 0){
               ordenado = true;
         }
         cuentaintercambios = 0;                                  //inicializamos de nuevo  a 0
      }
   }
```

4.2.- Quicksort.

Consiste en ordenar un array mediante un pivote, que es un punto intermedio en el array, es como si se ordenaran pequeños trozos del array, haciendo que a la izquierda estén los menores a ese pivote y a la derecha los mayores a éste, después se vuelve a calcular el pivote de trozos de listas. Usa *recursividad* (tema que trataremos más adelante). Le pasamos el array, su posición inicial y su posición final como parámetros.



Izquierda: 24, 21, 15, 46
Pivote: 65
Derecha: 88, 75, 85, 76, 99, 84, 79

```
public  static  void  quicksort (int lista1[ ],  int izq,  int der){
   int  i = izq;
   int  j = der;
   int  pivote = lista1[(i+j)/2];
   do {
      while (lista1[i] < pivote){
         i++;
      }
      while (lista1[j] > pivote){
         j--;
      }
      if (i <= j){
         int  aux = lista1[i];
         lista1[i] = lista1[j];
         lista1[j] = aux;
         i++;
         j--;
      }
   }while(i <= j);
   if (izq < j){
      quicksort(lista1, izq, j);
   }
   if (i < der){
      quicksort(lista1, i, der);
   }
}
```

5.- Arrays multidimensionales.

Java no soporta los arrays multidimensionales directamente, pero permite al programador especificar arrays unidimensionales cuyos elementos sean también arrays unidimensionales, con lo cual se obtiene el mismo efecto.

Un array de dos dimensiones se declararía:

tipo[ ][ ] nombre_array;

ó

tipo nombre_array[ ][ ];

En esta Figura el recuadro sombreado corresponde al elemento m[1][4].

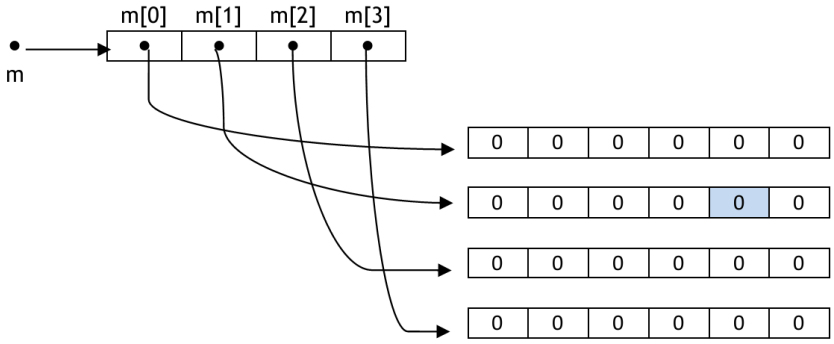


Figura 4. Representación esquemática del array bidimensional m.

Al igual que con los arrays de una sola dimensión, los arrays multidimensionales deben declararse y crearse. Podremos hacer arrays multidimensionales de todas las dimensiones que queramos y de cualquier tipo, todos los elementos del array serán del mismo tipo como en el caso de los arrays de una sola dimensión. La declaración comenzará especificando el tipo o la clase de los elementos que forman el array, después pondremos **tantos pares de corchetes como dimensiones tenga el array** y por último el nombre del array.

La creación se hace usando el operador *new*, seguido del tipo y los corchetes, en los cuales se especifica el tamaño de cada dimensión:

```
int[ ][ ] k = new   int[3][5];
```

Igualmente, **para acceder a cada una de las posiciones del array se utilizaría un índice por dimensión**:

```
k[1][3] = 28;
```

Al igual que los arrays de una dimensión, los arrays multidimensionales **pueden inicializarse en la propia declaración agrupándolos por filas entre llaves**. La siguiente instrucción crea un array de cuatro enteros (2*2) y los inicializa a los valores indicados entre llaves:

```
int[ ][ ] a = {{5, 4},{2,3}};
```

5.1.- Recorrido de un array multidimensional.

Para recorrer un array multidimensional podemos utilizar la instrucción *for tradicional*, empleando para ello tantas variables de control como dimensiones tenga el array. La siguiente instrucción almacenaría en cada una de las posiciones del array *k*, definido anteriormente, la suma de sus índices:

```
for(int   f = 0 ; f < 3 ; f++)                  //recorre la primera dimensión
        for(int   c = 0 ; c < 5 ; c++)          //recorre la segunda dimensión
                k[f][c] = f + c;
```

Igualmente, se puede utilizar también una instrucción *for mejorado* para recuperar el contenido de un array multidimensional. El siguiente ejemplo mostraría en pantalla los valores almacenados en el array *k* anterior:

```
for (int[ ]  n:k)      // → Ojo!
        for (int  p:n)
                System.out.println("valor:    " + p) ;
```

5.2.- Arrays multidimensionales irregulares.

La forma en que se representan los arrays multidimensionales los hace bastante flexibles. Los programas tambien pueden utilizar variables para especificar las dimensiones de los arrays, ya que *new* crea arrays en tiempo de ejecución, no en tiempo de compilación.

Es posible definir un array multidimensional en el que el número de elementos de la segunda y sucesivas dimensiones sea variable, indicando inicialmente sólo el tamaño de la primera dimensión.

Así, un array multidimensional en el que cada fila tiene un número distinto de columnas, puede crearse de la siguiente manera:

```
int [ ][ ]  irreg = new int [2][ ];        //crea 2 filas
irreg[0] = new  int[4];                    //crea 4 columnas para la fila 0
irreg[1] = new  int[6];                    //crea 6 columnas para la fila 1
```

ó también así,

```
int [ ][ ]  irreg2 ={{1, 2}, {3}, {4, 5, 6}};
```

La clase **Arrays** del paquete *java.util* soporta utilidades para los arrays tales como ordenación, comparación entre arrays y relleno con datos. Esta clase proporciona múltiples métodos para manipular arrays sobre múltiples tipos de datos primitivos (enteros, reales, caracteres y booleanos) y objetos, entre ellos:

- **static void fill(tipo[ ] array, tipo valor):** Llena el array con el *valor* indicado, todos los elementos iguales.
- **static boolean equals (tipo[ ] a1, tipo[ ] a2):** Chequea si los arrays son idénticos, comprobando valores primitivos (==) y referencias (con equals).
- **static boolean deepEquals(Object[ ] a1, Object[ ] a2):** Chequea si los arrays son idénticos, comprobando valores primitivos (==) y referencias (con *equals*). Además, si el array es multidimensional, profundiza en las sucesivas dimensiones.
- **static String toString(tipo[ ] datos):** Genera una cadena a partir del array.
- **static String deepToString(tipo[ ] datos):** Genera una cadena incluso si se trata de un array multidimensional.
- **static tipo[ ] copyOf(tipo[ ] datos, int n):** Genera una copia de los datos. Si *n* es mayor que *datos.length*, rellena con *null*. Si *n* es menor que *datos.length*, se ignora el exceso (es decir, trunca).

- **static tipo[ ] copyOfRange(tipo[ ] datos, int desde, int hasta):** Copia un segmento de los datos, el indicado con los valores *desde* y *hasta*.
- **static int binarySearch(tipo[ ] datos, tipo clave):** Busca en qué posición del array *datos* se encuentra la *clave* dada. El array debe estar ordenado ascendentemente.
- **static void sort(tipo[ ] datos):** Ordena ascendentemente el array.
- **static void sort(tipo[ ] datos, int desde, int hasta):** Ordena ascendentemente el array entre las posiciones indicadas.

6.- Introducción a las Colecciones.

Una problemática que surge frecuentemente en la programación, es organizar una colección de objetos. La primera solución que nos surge es utilizar un array.

```
String[ ] nombres = new String[10];
```

Sin embargo, a diferencia de los arrays, las colecciones **son dinámicas**, en el sentido de que no tienen un tamaño fijo y permiten añadir y eliminar objetos en tiempo de ejecución.

Java incluye en el paquete *java.util* un amplio conjunto de clases e interfaces para la creación y tratamiento de colecciones. Todas ellas proporcionan una serie de métodos para realizar las operaciones básicas sobre una colección, como son:

- Añadir objetos a la colección.
- Eliminar objetos de la colección.
- Obtener un objeto de la colección.
- Localizar un objeto en la colección.
- Iterar a través de una colección.

A continuación, vamos a estudiar algunas de las clases de colección más significativas. Para más información ver el documento *Guía de colecciones en Java*.

6.1.- Collection e Iterator.

La interfaz más importante es **Collection<E>**. La interfaz Collection, que define un tipo que podemos denominar Colección, está definida en el paquete *java.util*. Una colección es un tipo muy general, que agrupa objetos (elementos) de un mismo tipo; su comportamiento específico viene determinado por sus subinterfaces, que pueden admitir elementos duplicados o no, y cuyos elementos pueden estar ordenados o no según determinado criterio. No existe una implementación específica de la interfaz Collection; sí la tienen sus subinterfaces. El tipo Collection se utiliza para declarar variables o parámetros donde se quiere la máxima generalidad posible, esto es, que puedan ser utilizados tanto por listas como por conjuntos (Collection tiene otros subtipos que se tratarán más adelante). La interfaz Collection es genérica (se puede definir sobre cualquier tipo de objeto) y hereda de *Iterable*, lo que implica que las colecciones, así como las listas y los conjuntos, subtipos suyos, son iterables y se puede utilizar con ellas el *for extendido*. **Los elementos que puede contener tienen que ser obligatoriamente objetos**. Esto quiere decir que no se puede definir sobre los tipos *int* o *double* por ejemplo y se ha de recurrir a los tipos envoltorios *Integer* o *Double*. Las operaciones del tipo Collection se especifican a continuación (puedes obtener más información de la interfaz Collection en la documentación de la API de Java). En general, el valor devuelto por los métodos que devuelven boolean será *true* si la colección queda modificada por la aplicación del método y *false* en caso contrario.

boolean	**add**(**E** e) Añade un elemento a la colección, devuelve *false* si no se añade.
boolean	**addAll**(**Collection<? extends E>** c) Añade todos los elementos de c a la colección que invoca. Es el operador unión. Devuelve *true* si la colección original se modifica.
void	**clear**() Borra todos los elementos de la colección.
boolean	**contains**(**Object** o) Devuelve *true* si o está en la colección invocante.
boolean	**containsAll**(**Collection<?>** c) Devuelve *true* si la colección que invoca contiene todos los elementos de c.

boolean	**isEmpty**() Devuelve *true* si la colección no tiene elementos.
boolean	**remove**(**Object** o) Borra el objeto o de la colección que invoca; si no estuviera se devuelve *false*.
boolean	**removeAll**(**Collection**<?> c) Borra todos los objetos de la colección que invoca que estén en c. Devuelve *true* si la colección original se modifica.
boolean	**retainAll**(**Collection**<?> c) En la colección que invoca sólo se quedarán aquellos objetos que están en c. Por tanto, es la intersección entre ambas colecciones. Devuelve *true* si la colección original se modifica.
int	**size**() Devuelve el número de elementos.

Disponemos la clase *java.util.**Collections*** que trabaja sobre los objetos que implementen la interface Collection. Todos sus métodos son estáticos, por listar algunos *binarySearch*, *addAll*, *reverse*, *reverseOrder* ...

Tenemos **dos formas de recorrer una colección**. Una es con la sentencia *for each*, esta es una buena opción si solo queremos realizar operaciones de lectura ya que no podemos recorrer y modificar la colección a la vez. La segunda mediante un **Iterator**, resulta mejor cuando queramos realizar modificaciones de la colección. Las operaciones de un *Iterator* son:

- **boolean hasNext()**. Indica si hay más elementos por recorrer en la colección. Cuando el objeto Iterator esté apuntando al último elemento, la llamada a este método devolverá *false*.

- **Object next()**. La llamada a este método sobre un objeto Iterator provoca que éste pase a apuntar al siguiente objeto de la colección, devolviendo el nuevo objeto apuntado. Hay que tener en cuenta que, inicialmente, un Iterator se encuentra apuntando a la posición que está antes del primer objeto de la colección, por lo que la primera llamada a *next* () devolverá el primer objeto. Si no existe siguiente elemento, lanzará una excepción (NoSuchElementException para ser exactos), con lo que conviene chequear primero si el siguiente elemento existe.

- **remove**().Elimina de la colección el último elemento retornado en la última invocación de *next*() (no es necesario pasarselo por parámetro). Cuidado, si *next()* no ha sido invocado todavía, saltará una incómoda excepción.

¿Cómo recorreríamos una colección con estos métodos? Pues de una forma muy sencilla, un simple *bucle mientras* (*while*) con la condición *hasNext*() nos permite hacerlo:

```
Iterator <Integer>  it  = lista.iterator();

while (it.hasNext())                // Mientras que haya un siguiente elemento, seguiremos en el bucle.
{
        Integer t = it.next();      // Escogemos el siguiente elemento.
        if (t%2 == 0)
                it.remove();
        }
```

Existen un conjunto de interfaces que heredan de Collection, que nos aportan más prestaciones:

- **List**. Es una colección donde se mantiene la posición de los elementos. No está ordenada y puede haber elementos repetidos.
- **Set**. Es una colección en la que no existen elementos repetidos.
- **Map**. Asocian un valor a una clave.
- **Queue**. Es una colección que permite crear colas FIFO o LIFO.

6.2.- Listas.

Conceptualmente, **las listas** representan colecciones de elementos en las que accedemos a cada elemento indicando su posición en la lista. Cada elemento está referenciado mediante un índice; el índice del primer elemento es el 0. Las listas pueden contener elementos duplicados. La inicialización de *List* se hace con el operador *new* y el constructor de una clase.

Las clases **ArrayList** y **LinkedList** implementan *List*. La elección de una u otra implementación dependerá del tipo de operaciones que realicemos sobre ellas. Si se va a acceder preferentemente a los elementos mediante índice o a realizar búsquedas, debe usarse *ArrayList*. Si preferentemente se van a realizar operaciones de inserción o borrado al principio o al final debe usarse *LinkedList*.

List es una lista, no ordenada, en que se mantiene el orden de los elementos, pudiendo acceder a su contenido según la posición. Esta lista crece según las necesidades, con lo que nos podemos olvidar de los tamaños. Esta interface hereda de *Collection*, y aporta varias operaciones específicas, que no aparecen en *Collection*:

`void`	**`add`**`(int index, E element)` Inserta el elemento especificado en la posición especificada. El que estaba previamente en la posición `index` pasará a la posición `index + 1`, el que estaba en la posición `index + 1` pasará a la posición `index + 2`, etc. Los valores lícitos para `index` son `0 ≤ index ≤ size()`
`boolean`	**`addAll`**`(int index, Collection<? extends E> c)` Inserta todos los elementos de c en la posición especificada, desplazando el que estaba previamente en la posición `index` a la posición `index + c.size()`,etc. Los valores lícitos para `index` son `0 ≤ index ≤ size()`
`E`	**`get`**`(int index)` Devuelve el elemento de la lista en la posición especificada. Los valores lícitos para `index` son `0 ≤ index < size()`.
`int`	**`indexOf`**`(Object o)` Devuelve el índice donde se encuentra por primera vez el elemento o (si no está devuelve `−1`).

int	**lastIndexOf**(**Object** o) Devuelve el índice donde se encuentra por última vez el elemento o (si no estuviera devuelve −1).
E	**remove**(int index) Borra el elemento de la posición especificada. Los valores lícitos para index son 0 ≤ index < size()
E	**set**(int index, **E** element) Remplaza el elemento de la posición indicada por el que se da como argumento. Los valores lícitos para index son 0 ≤ index < size()
List<E>	**subList**(int fromIndex, int toIndex) Devuelve una **vista** de la porción de la lista entre fromIndex, inclusive, and toIndex, sin incluir. Los valores lícitos de fromIndex y toIndex son 0 ≤ fromIndex ≤ toIndex ≤ size()

El método *subList*() devuelve una vista de la lista original. Debes saber que las operaciones aplicadas a una sublista repercuten sobre la lista original. Por ejemplo, si ejecutamos el método *clear*() sobre una sublista, se borrarán todos los elementos de la sublista, pero también se borrarán dichos elementos de la lista original y lo mismo ocurre al añadir un elemento, se añade en la sublista y en la lista original.

6.2.1.- ArrayList.

Un **ArrayList** representa una colección basada en índices, en la que cada objeto de la misma tiene asociado un número (índice) según la posición que ocupa dentro de la colección, siendo 0 la posición del primer elemento. Tiene la ventaja de realizar lecturas muy rápidas. El problema está en borrar elementos intermedios, ya que tiene que mover el resto del contenido.

Antes de la versión Java 5, la forma de declarar una colección de objetos con la clase ArrayList era la siguiente,

ArrayList variable_objeto = new ArrayList();

En la que no había que declarar de qué tipo van a ser los objetos de la lista, esto obligaba a que en la recuperación de dichos objetos había que hacer una transformación de tipos. Permitiendo incluso que en la lista existieran objetos de diferente tipo, con el consiguiente problema de gestión e incluso de estabilidad de la lista.

A partir de la versión 5, es obligatorio declarar de qué tipo van a ser los objetos de la lista, tal como se recoge en la sentencia siguiente para una lista:

ArrayList <tipo_objeto> variable_objeto = new ArrayList <tipo_objeto> ();

Los principales métodos expuestos por esta clase son:

- **boolean add(Object o)**. Añade un nuevo objeto a la colección (su referencia) y lo sitúa al final de la misma, devolviendo el valor *true*.

```
ArrayList <Integer> v = new ArrayList <Integer>();
v.add(5);          //añade el número en la primera posición del ArrayList
```

- **void add(int indice, Object o)**. Añade un elemento al ArrayList en la posición especificada por índice, desplazando hacia delante el resto de los elementos de la colección.
- **Object get(int indice)**. Devuelve el elemento que ocupa la posición indicada.

```
ArrayList <String> v = new  ArrayList <String> ();
v.add("texto") ;
String   s = v.get(0);
```

```
ArrayList <Integer>  v = new  ArrayList <Integer> ();
v.add(6);
int   i =  v.get (0);                    // Recupera  el  elemento
System.out.println(i ) ;                 // y  lo muestra  en  número
```

- **Object remove(int indice)**. Elimina de la colección el elemento que ocupa la posición indicada, desplazando hacia atrás los elementos de las posiciones siguientes. Devuelve el elemento eliminado.
- **void clear()**. Elimina todos los elementos de la colección.

  int indexOf(Object o). Localiza en el ArrayList el elemento indicado como parámetro, devolviendo su posición. En caso de que el elemento no se encuentre en la colección, la llamada al método devolverá como resultado el valor -1.
- **int size()**. Devuelve el número de elementos almacenados en la colección. Utilizando este método conjuntamente con *get*(), se puede recorrer la colección completa.

El siguiente método realiza el recorrido de un ArrayList de cadenas para mostrar su contenido en pantalla:

```
public  void  muestra(ArrayList <String> v){
        for(int   i = 0 ; i < v.size() ; i++){
                System.out.println(v.get(i) ) ;
        }
}
```

Como sabemos, a partir de la versión Java 5 se puede utilizar la variante *for-each* para simplificar el recorrido de colecciones, así el método anterior quedaría:

```
public  void  muestra(ArrayList <String> v){
        for(String   ob:v)
                System.out.println(ob);
}
```

A modo de resumen sobre el funcionamiento de la colección ArrayList, se presenta el siguiente programa para la gestión de temperaturas:

Inicialmente, se presentará un menú en pantalla para elegir la opción deseada (1. Añadir temperatura, 2. Mostrar temperatura media. 3. Mostrar temperaturas extremas), menú que volverá a presentarse de nuevo en la pantalla tras completar cualquiera de las tres primeras opciones. A medida que lo analicéis ir mejorando el código expuesto:

```
import java.io.*;
import java.util.*;
public class Gestion {
 public static void main(String[ ] args) throws IOException{
  ArrayList temperaturas = new ArrayList();
  String opcion;
  BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
  do{
        System.out.println("Elegir opción:\n");
        System.out.println("1. Añadir temperatura");
        System.out.println("2. Mostrar temperatura media");
        System.out.println("3. Mostrar temperaturas extremas");
        System.out.println("4. Salir");
        opcion = bf.readLine();
        switch(Integer.parseInt(opción)){
              case 1:
                    double temp;
                    System.out.println("Introduce la temperatura: ") ;
                    temp = Double.parseDouble(bf.readLine() ) ;
                    almacenaTemperatura(temp,temperaturas);
                    break;
              case 2:
                    muestraMedia(temperaturas);
                    break;
              case 3:
                    muestraExtremas(temperaturas);
        }
  }while(!opción.eguals("4"));
}
        static void almacenaTemperatura(double d, ArrayList temperaturas){
        //Necesita convertir el número a objeto para poderlo añadir al ArrayList
              temperaturas.add(new Double(d);
        }
```

```
        static  void  muestraMedia(ArrayList  temperaturas){
                double  media = 0.0;
                for(Object  tp:temperaturas){
                        //Las  temperatura  se  convierten
                        //explícitamente  al  tipo  de  objeto  original
                        //para  después  extraer  el  valor  numérico
                        media += ((Double)tp).doubleValue();
                }
                media /= temperaturas.size()  ;
                System.out.println("La  temperatura  media  es:  "+ media);
        }
        static  void  muestraExtremas(ArrayList  temperaturas){
        //Se  inicializan  las  variables  extremo  con
        //el  valor  de  la  primera  temperatura
                double  máxima;
                máxima=((Double)temperaturas.get(0)).doubleValue();
                double  mínima = máxima;
                for(Object  tp:temperaturas){
                        double  aux;
                        aux = ((Double)tp).doubleValue();
                        if(aux > máxima){
                                máxima = aux;
                        }
                        if(aux < mínima){
                                mínima = aux;
                        }
                }
                System.out.println("La  temperatura  máxima  es  "+ máxima);
                System.out.println("La  temperatura  mínima  es  " + mínima);
        }
}
```

Esta clase no está preparada para trabajar con varios hilos; para ello tenemos la clase **Vector**.

6.2.2.- LinkedList.

Son listas doblemente enlazadas, es decir son listas que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos. Los nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento. Tener un doble enlace significa que en cada nodo se almacena la información de cuál es el siguiente nodo y además, de cuál es el nodo anterior. Si un nodo no tiene nodo siguiente o nodo anterior, se almacena *null* o nulo para ambos casos.

No es el caso de los ArrayList. Estos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente a nosotros, no nos enteramos cuando se produce, pero eso redunda en una diferencia de rendimiento notable dependiendo del uso. Los ArrayList son más rápidos en cuanto a acceso a los elementos, acceder a un elemento según su posición es más rápido en un array que en una lista doblemente enlazada (hay que recorrer la lista). En cambio, eliminar un elemento implica muchas más operaciones en un ArrayList que en una lista enlazada de cualquier tipo.

¿Y esto que quiere decir? Que si se van a realizar muchas operaciones de eliminación de elementos sobre la lista, conviene usar una lista enlazada LinkedList, pero si no se van a realizar muchas eliminaciones, sino que solamente se van a insertar y consultar elementos por posición, conviene usar una lista basada en arrays redimensionados ArrayList.

El siguiente ejemplo muestra cómo utilizarla:

```
import java.util.*;
public class VerLista{
	public static void main(String args[ ]){
		LinkedList<String>  lista = new  LinkedList<String>();
		lista.add("Martes");
		lista.add("Miercoles");
		lista.add("Jueves");
		lista.add("Viernes");
		lista.addFirst("Domingo");
		lista.add("Sabado");
		lista.addLast("Sabado");
		System.out.println("\nLa semana \"ideal\"");
		ListIterator i = lista.listIterator();
		while(i.hasNext())
			System.out.print(i.next().toString()+" ");
	}
}
```

El resultado de la ejecución sería:

La semana "ideal"

Domingo Martes Miercoles Jueves Viernes Sabado Sabado

6.2.3.- Vector.

Es muy parecida al anterior, pero con la diferencia de estar preparada para trabajar con hilos. De esta clase hereda *Stack*.

La clase **Stack** permite crear objetos conocidos como pilas, es decir colecciones de objetos cuya característica principal es su forma de almacenamiento, en la cual el último en llegar a la pila es el primero en salir.

Una pila (stack en inglés) es una estructura de datos de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos. Se aplica en multitud programas debido a su simplicidad y ordenación implícita en la propia estructura.

Para el manejo de los datos se cuenta con dos operaciones básicas: apilar (*push*), que coloca un objeto en la pila, y su operación inversa, retirar (o desapilar, *pop*), que retira el último elemento apilado

Consta sólo de cinco métodos:

- **E push(E item)**. Permite introducir un elemento en la pila.
- **E pop()**. Permite sacar un elemento de la pila.
- **boolean empty()**. Permite saber si la pila está vacía.
- **E peek()**. Permite obtener una copia del último elemento que ha entrado en la pila, sin sacarlo de ésta.
- **int search(Object o)**. Busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella.

Vamos a ver un ejemplo de utilización de una pila con la clase Stack:

```
import java.util.*;
public class VerPila {
	public static void main(String args[ ]){
		Stack<String> pila = new Stack<String>();
		if(pila.empty())
			System.out.println("La pila está vacía");
		pila.push("primero");
		pila.push("segundo");
		pila.push("tercero");
		pila.push("cuarto");
		pila.push("quinto");
		System.out.print("La pila tiene ");
		System.out.println(pila.size()+ " elementos");
		System.out.print("El primero en salir es: ");
		System.out.println(pila.peek().toString());
		while(!pila.empty())
			System.out.print(pila.pop().toString()+" ");
	}
}
```

Resultado de ejecutar VerPila.java:

La pila está vacía

La pila tiene 5 elementos

El primero en salir es: quinto

quinto cuarto tercero segundo primero

6.3.- Set.

Es una colección en la que no contiene elementos duplicados. Hereda del interface Collection, apenas añade métodos nuevos. Podemos destacar la implementación **HashSet**.

6.3.1.- HashSet.

Esta implementación se basa en una tabla *hash*. Nos referimos a hash, como una función que genera claves, garantizando que el mismo elemento genera siempre la misma clave. Para obtener la clave, se aplica la función *hash* al valor devuelto por el método *hashcode*() de Object. Utilizando esta clave como índice de un array se puede obtener accesos muy rápidos a los objetos. Es posible que dos objetos diferentes generen la misma clave, en ese caso se dice que ha habido una **colisión**. Este aspecto nos obliga a tener asociado a cada clave más de un elemento. Normalmente, si la capacidad está bien elegida, las colisiones no son muy probables. Necesitan bastante memoria y no almacenan los objetos de forma ordenada (al contrario pueden aparecer completamente desordenados). Pero ofrece un tiempo constante en la ejecución de las operaciones básicas: add, remove, contains y size. La clase **HashSet** permite construir listas en las cuales no puede haber repeticiones de datos.

El siguiente ejemplo muestra cómo utilizar esta clase, en el que se intenta añadir dos veces el día *sábado* pero no cuela, y la semana desordenada sólo cuenta con siete días:

```
import java.util.*;
public class VerConjunto{
        public static void main(String args[ ]){
                HashSet<String>  lista = new  HashSet<String>();
                lista.add("Lunes");
                lista.add("Martes");
                lista.add("Miércoles");
                lista.add("Jueves");
                lista.add("Viernes");
                lista.add("Sábado");
                lista.add("Sábado");
                lista.add("Domingo");
                System.out.print("La semana \"desordenada ");
                System.out.println("y sin repeticiones\"");
                Iterator i = lista.iterator();
                while(i.hasNext())
                        System.out.print(i.next().toString()+" ");
        }
}
```

Resultado:

La semana "desordenada y sin repeticiones"

Jueves Martes Sábado Lunes Viernes Domingo Miércoles

6.4.- Map.

Un **Map** es una colección de duplas de clave-valor, también conocido como diccionario. Las claves no pueden estar repetidas, si dos elementos tienen la misma clave se considera que es el mismo. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.

En Java existe la interface java.util.Map que define los métodos que deben tener los mapas, y existen tres implementaciones principales de dicha interface: java.util.**HashMap**, java.util.TreeMap y java.util.LinkedHashMap .

Los mapas utilizan clases genéricas para dar extensibilidad y flexibilidad, y permiten definir un tipo base para la clave, y otro tipo diferente para el valor. Veamos un ejemplo de cómo crear un mapa, que es extensible a los otros dos tipos de mapas:

HashMap<String, Integer> t = new HashMap<String, Integer>();

El mapa anterior permite usar cadenas como llaves y almacenar de forma asociada a cada llave, un número entero.

Veamos los métodos principales de la interface Map:

Método.	Descripción.
V put(K key, V value);	Inserta un par de objetos llave (*key*) y valor (*value*) en el mapa. Si la llave ya existe en el mapa, entonces retornará el valor asociado que tenía antes, si la llave no existía, entonces retornará *null*.
V get(Object key);	Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, retornará *null*.
V remove(Object key);	Elimina la llave y el valor asociado. Retorna el valor asociado a la llave, por si lo queremos utilizar para algo, o *null*, si la llave no existe.
boolean containsKey(Object key);	Retornará *true* si el mapa tiene almacenada la llave pasada por parámetro, *false* en cualquier otro caso.
boolean containsValue(Object value);	Retornará *true* si el mapa tiene almacenado el valor pasado por parámetro, *false* en cualquier otro caso.
int size();	Retornará el número de pares llave y valor almacenado en el mapa.
boolean isEmpty();	Retornará *true* si el mapa está vacío, *false* en cualquier otro caso.
void clear();	Vacía el mapa.

6.4.1.- HashMap.

La utilización de colecciones basadas en claves resulta útil en aquellas aplicaciones en las que se requiera realizar búsquedas de objetos a partir de un dato que lo identifica. Por ejemplo, si se va a gestionar una colección de objetos de tipo "Empleado", puede resultar más práctico almacenarlos en un HashMap, asociándoles como clave el "dni", que guardarlos en un ArrayList en el que a cada empleado se le asigna un índice según el orden de almacenamiento.

La creación de un objeto *HashMap* se realiza utilizando el constructor sin parámetros de la clase:

HashMap <tipo_clave, tipo_elemento> variable = new HashMap <tipo_clave, tipo_elemento> ();

Los principales métodos expuestos por la clase *HashMap* para manipular la colección son los siguientes:

- **Object put(Object key, Object valor)**. Añade a la colección el elemento *valor*, asignándole la clave especificada por *key*. En caso de que exista esa clave en la colección, el objeto que tenía asignada esa clave se sustituye por el nuevo objeto *valor*, devolviendo el objeto sustituido. Por ejemplo, el siguiente código:

```
HashMap<String, String>  hm = new  HashMap<String, String> ();
hm.put("a21","pepito");
System.out.println("Antes  se  llamaba  "+hm.put("a21","luis"));
```

  Mostrará en pantalla: *Antes se llamaba pepito*

- **boolean containsKey(Object key)**. Indica si la clave especificada existe o no en la colección.
- **Object get(Object key)**. Devuelve el valor que tiene asociado la clave que se indica en el parámetro. En caso de que no exista ningún objeto con esa clave asociada devolverá *null*.
- **Object remove(Object key)**. Elimina de la colección el valor cuya clave se especifica en el parámetro. En caso de que no exista ningún objeto con esa clave, no hará nada y devolverá *null*, si existe, eliminará el objeto y y el mismo será devuelto por el método.
- **int size()**. Devuelve el número de objetos almacenados en la colección.
- **Collection values()**. Devuelve una *Collection* con los valores del conjunto. La colección es resultado de eliminar las claves en el *HashMap*.
- **Set keySet()**. Se obtienen las claves y mediante un *iterador* se recorre la lista de claves. De esta forma si queremos saber también el valor de cada elemento tenemos que usar el método *get(clave)*.

Para recorrer los mapas con iteradores, hay que hacer un pequeño truco. Usamos el método ***keySet***() para generar un conjunto con las llaves existentes en el mapa. Veamos cómo sería:

```
HashMap<Integer, Integer>   mapa = new HashMap<Integer,Integer> test();
for (int i = 0 ; i<10 ; i++)
        mapa.put(i, i);                    // Insertamos datos de prueba en el mapa.
for (Integer llave:mapa.keySet())          // Recorremos el conjunto generado por keySet que tendrá las llaves.
        Integer valor = mapa.get(llave);   //Para cada llave, accedemos a su valor si es necesario.
```

Hay que tener en cuenta que el conjunto generado por keySet no tendrá obviamente el método *add* para añadir elementos al mismo, dado que eso tendrás que hacerlo a través del mapa.

Si se usan iteradores, y se quieren eliminar elementos de la colección (e incluso de un mapa), se debe usar el método ***remove()*** del iterador y no el de la colección. Si se eliminan los elementos utilizando el método *remove* de la colección, mientras se está dentro de un bucle de iteración, o dentro de un bucle for-each, los fallos que pueden producirse en el programa son impredecibles. Los problemas son debidos a que el método *remove* del iterador elimina el elemento de dos sitios, de la colección y del iterador en sí (que mantiene interiormente información del orden de los elementos). Si usas el método *remove* de la colección, la información solo se elimina de un lugar, de la colección.

El siguiente método recibe como parámetro un objeto *HashMap* en el que se han almacenado objetos *String* con claves asociadas de tipo *String*, su misión consiste en mostrar en pantalla todos los valores almacenados. Tendríamos que escribir el siguiente código:

```
public void muestraDatos(HashMap<String, String> tb){
        String valor, clave;
        Iterator i = tb.iterator();
        while (i.hasNext()) {
                clave = (String)i.next ();
                valor = (String)tb.get(clave);
                System.out.println(valor);
        }
}
```

Con el fin de aclarar el funcionamiento de esta clase, se presenta a continuación un programa para la gestión de una lista de nombres:

El programa presentará inicialmente un menú en pantalla para elegir la opción deseada (1. Añadir nombre, 2. Eliminar nombre, 3. Mostrar todos, 4. Salir), menú que volverá a presentarse de nuevo en la pantalla tras completar cualquiera de las tres primeras opciones. Cada nombre llevará asociado como clave un DNI:

```
import java.io.*;
import java.util.*;
public class GestionNombres {
   public static void main(String[ ] args) throws IOException {
      HashMap nombres = new HashMap();
      String opcion;
      BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
      do {
         System.out.println("Elegir opción:\n");
         System.out.println("1.  Añadir nombre");
         System.out.println("2.  Eliminar nombre");
         System.out.println("3.  Mostrar todos los nombres");
         System.out.println("4.  Salir");
         opcion = bf.readLine();
         switch (Integer.parseInt(opcion)) {
            case 1:
               String nom,dni;
               System.out.println("Introduce  Nombre:   ");
               nom = bf.readLine();
               System.out.println("DNI:   ");
               dni = bf.readLine();
               almacenaNombre(nom, dni, nombres);
               break;
            case 2:
               String d;
               System.out.println("Introduzca  el  dni: ");
               d = bf.readLine();
               eliminaNombre(d, nombres);
               break;
            case 3:
               mostrarTodos(nombres);
               break;
         }
      } while (!opcion.equals("4"));
   }

static void almacenaNombre(String n, String k, HashMap lista) {
      if (!lista.containsKey(k))
         lista.put(k, n);
   }

   static void eliminaNombre(String k, HashMap lista) {
      if (lista.containsKey(k))
         lista.remove(k);
      }

   static void mostrarTodos(HashMap lista) {
      System.out.println("Los  nombres  son:   ");
      Iterator claves = lista.keySet().iterator();
      while (claves.hasNext()) {
         String k = (String) claves.next();
         System.out.println(k + "  -  " + lista.get(k));
      }
   }
}
```

7.- Definición de tipos genéricos.

Para que sea posible especificar en la creación del objeto de colección los tipos elementos que se pueden añadir, es necesario definir estas clases con una sintaxis especial. Si acudimos a la documentación del API del J2SE 7 para obtener información sobre la clase ArrayList, observamos cómo dicha clase aparece declarada de la siguiente manera:

class ArrayList <E>

A esta forma de definir una clase se la conoce como definición con **tipo parametrizado** o definición de **tipo genérico**. La anterior declaración se lee "clase ArrayList de E", **donde E**, llamado también parámetro de tipo, es la letra utilizada para referirse al tipo de elementos que se pueden añadir y **representa a cualquier tipo de objeto Java**.

Como hemos visto, es en la declaración de una variable de la clase ArrayList y en la creación de un objeto de la misma cuando se tiene que especificar el tipo concreto de objetos que se van a tratar en esa colección, sustituyendo la letra E por el nombre de clase correspondiente.

A diferencia de la versión 1.4 en donde el método *add*() se declaraba como *boolean add(Object o)*, la declaración de este método en la clase ArrayList genérica a partir de la versión Java 5 tiene el formato *boolean add(E o)*.

Lo que significa que al ArrayList no se le puede añadir cualquier objeto, sino solamente objetos del tipo declarado en la colección (E). Al especificar un tipo concreto en la creación del objeto ArrayList, todas las referencias a E en los métodos de ese objeto serán sustituidas automáticamente por el tipo específico. Por ejemplo, para un objeto ArrayList de Integer, el formato del método add() quedaría convertido en *boolean add(Integer o)*.

Los tipos genéricos no se limitan solamente a las colecciones, también podemos definir clases o tipos genéricos propios. Por ejemplo, la definición del siguiente tipo genérico corresponde a una especie de clase de envoltorio capaz de encapsular cualquier tipo de objeto:

```
public class Wrapper<E> {
        private E data;          //dato encapsulado que puede ser de cualquier tipo objeto
        public void setData(E d) {
                data = d;
        }
         public E getData(){
                return data;
        }
}
```

El siguiente programa muestra un ejemplo de utilización de esta clase para encapsular cadenas de caracteres:

```
public class PruebaData {
        public static void main(String[] args) {
                Wrapper<String> w = new Wrapper<String>();
                w.setData("mi cadena");
                String d = w.getData();
                System.out.println("La cadena es: "+d);
        }
}
```

Es importante destacar que el parámetro de tipo E representa un tipo objeto, lo que significa que sólo podrán utilizarse como argumentos de tipo referencias a objeto, nunca tipos primitivos. La siguiente instrucción produciría, por tanto, un error de compilación:

Wrapper<char> w =new Wrapper<char> () ;

Como se puede ver, la utilización de tipos genéricos simplifica y optimiza el desarrollo de las aplicaciones, especialmente si se hace uso de colecciones.