Tema 8

Estructuras de datos dinámicas.

8.1.- Introducción
8.2.- Listas.
8.3.- Pilas.
8.4.- Colas.
8.5.- Árboles.
8.6.- Estructuras en JAVA
- Colecciones:
  - Vector
  - Stack
  - Hashtable
  - LinkedList

8.7.- Clases de tipo Interfaz.
8.8.- Otras clases Java del paquete java.util

8.1 Introducción.

Los datos estudiados hasta ahora se denominan estáticos. Ello es debido a que las variables son direcciones simbólicas de posiciones de memoria; esta relación entre nombres de variables y posiciones de memoria es una relación estática que se establece por la declaración de las variables.

Aunque el contenido de una posición de memoria asociada con una variable puede cambiar durante la ejecución, es decir, el valor de la variable puede cambiar, las variables por sí mismas no se pueden crear ni destruir durante la ejecución.

En algunas ocasiones, no se conoce por adelantado cuánta memoria se requiere para un programa. En esos casos es conveniente disponer de un método para adquirir posiciones adicionales de memoria a medida que se necesiten durante la ejecución del programa y liberarlas cuando no se necesiten. Las variables dinámicas se utilizan para crear estructuras dinámicas de datos que se pueden ampliar y comprimir a medida que se requiera durante la ejecución del programa.

Una estructura de datos dinámica es una colección de elementos denominados ***nodos de la estructura.***

Pueden ser: Lineales
No lineales.

La potencia y flexibilidad de un lenguaje están directamente relacionadas con las estructuras de datos que posee. Cuando una aplicación particular requiere una estructura de datos no soportada por el lenguaje, se hace necesaria una labor de programación para representarla.

Una estructura de datos se dice que es estática cuando el tamaño de memoria que ocupa es fijo, es decir, siempre ocupa la misma cantidad de espacio en memoria. Por consiguiente si se define una lista, se debe anticipar la longitud de esa lista cuando se escribe un programa, y es imposible ampliar el espacio de memoria disponible.

Las estructuras se convierten en dinámicas cuando los elementos pueden ser insertados o suprimido directamente sin necesidad de algoritmos complejos. Se distinguen las estructuras dinámicas de las estáticas por los modos en que se realizan las inserciones y borrados de elementos.

Las estructuras dinámicas de datos se dividen en dos grandes grupos:

Lineales: listas, pilas, colas.

No lineales: árboles, grafos.

ESTRUCTURAS LINEALES DE DATOS.

8.2 .- LISTAS.

Primero los valores se almacenan en un nodo. Un nodo tiene al menos un campo de datos o valor y un enlace (referencia o puntero) con el siguiente nodo. El campo enlace apunta, (proporciona la dirección de) al siguiente nodo de la lista. El último nodo de la lista enlazada, por convenio, se suele representar por un enlace con la palabra reservada *null* (nulo) una barra inclinada.

Un puntero, es una variable cuyo valor es la dirección o posición de otra variable. En las listas enlazadas no es necesario que los elementos de la lista sean almacenados en posiciones físicas adyacentes, por consiguiente la inserción y borrado no exige desplazamiento como en el caso de las listas contiguas.

Una lista enlazada sin ningún elemento se llama lista vacía su puntero inicial o de cabecera tienen el valor nulo.

Una lista enlazada se define por:

- El tipo de sus elementos: campo de información y campo de enlace (puntero).
- Un puntero de cabecera que permite acceder al primer elemento de la lista.
- Un medio para detectar el último elemento de la lista: puntero nulo.

8.2.1 Operaciones con listas enlazadas.

Las operaciones que normalmente se ejecutan con listas incluyen:

1. Recuperar información de un nodo específico (acceso a un elemento)
2. Encontrar el nodo que contiene una información específica.(buscar)
3. Insertar un nuevo nodo en un lugar específico de la lista.
4. Insertar un nuevo nodo en relación a una información particular.
5. Borrar un nodo existente que contiene información específica.

8.3 .- PILAS. Una pila es una estructura en la que se almacenan datos de modo que cuando se desea obtener un dato de la misma, se obtendrá siempre el último introducido .Una pila (stack) es un tipo especial de lista lineal en la que **la inserción y borrado de nuevos elementos** se realiza **sólo por un extremo que se denomina cima**, tope (top). A estas estructuras también se denominan listas LIFO.

Las operaciones asociadas a las pilas son:

Push meter o poner un elemento en la pila

Pop sacar o quitar un elemento de la pila.

Se pueden representar mediante un vector de tamaño máximo determinado.

8.4.- COLAS.

Una cola es una estructura en la que se almacenan datos de modo que cuando se desea obtener un dato de la misma se obtiene siempre el primero introducido y aún no sacado. Es como una cola de espera en la que nadie se cuela y siempre entra primero el primero que llega. Las operaciones básicas que se permiten en una cola son: crearla, introducir un elemento, sacar un elemento y preguntar si está vacía.

Los elementos de una cola se eliminan por el principio y se insertan por el final.

Las colas y las pilas se suelen implementar utilizando vectores o listas enlazadas. La implementación más sencilla se hace con vectores simples aunque al utilizarlos tendremos que poner un límite al número de elementos que se podrán almacenar.

ESTRUCTURAS NO LINEALES

8.5 ÁRBOLES.

Los árboles son una estructura dinámica utilizada para representar jerarquías, evaluar expresiones, validar instrucciones, o determinar distintas posibilidades para solucionar un problema etc.

Los árboles se utilizan con profusión en inteligencia artificial, por ejemplo, en una partida de ajedrez a partir de una posición dada se puede estudiar todas las posibilidades del adversario, después, para cada una de ellas, se pueden reflejar la distintas posibilidades nuestras, y así sucesivamente hasta un determinado nivel en el que se puede evaluar cuál de todas las posibles jugadas es la que lleva a mejor fin.

En un árbol, cada nodo de un nivel inferior está enlazado únicamente con uno del nivel inmediato superior, y un determinado nodo puede tener desde 0 a ene enlaces con elementos del nivel inmediatamente inferior. En el nivel superior del árbol habrá uno y sólo un elemento, que se llamará **raíz**. Cada elemento del árbol recibe el nombre de **nodo**.

Se llama **padre** al nodo del que cuelga algún otro nodo e **hijos** a los nodos que cuelgan del padre. Extendiendo esta definición, se denominan descendientes a todos los nodos que cuelgan directa o indirectamente de un nodo que llamaremos antecesor. Los nodos sin descendientes se llaman **hojas**.

Según el número máximo de hijos que pueda tener un nodo cualquiera de un árbol, se definen distintos tipos:

Árbol binario: cada nodo tiene como máximo dos hijos.

Árbol ternario: cada nodo tiene como máximo tres hijos.

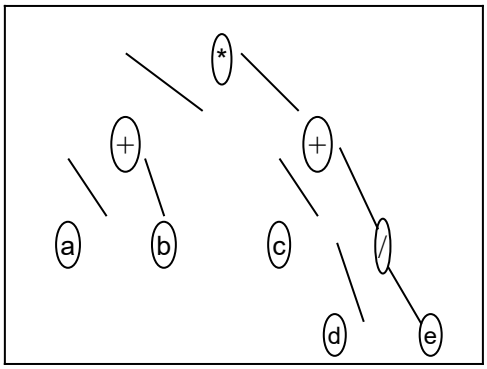
Se llama **altura** de un árbol al número máximo de niveles que tiene.

Se llama **árbol ordenado** al que tiene sus nodos en un determinado orden.

Para recorrer un árbol existen tres métodos diferentes que definen el orden en que se recorren los nodos del árbol.

- **Preorden:** raíz, subárbol izquierdo, subárbol derecho.
- **Orden central o Inorden:** subárbol izquierdo, raíz, subárbol derecho.
- **Postorden:** subárbol izquierdo, subárbol derecho, raíz.

Obtener la expresión equivalente del siguiente árbol:



Preorden: *+ab+c/de

Inorden : a + b * c + d / e

Postorden : a b + c d e / + *

Ejem. Algoritmo que nos permita contar las hojas de un árbol.

```
Procedimiento contarhojas (var  entero: cont; punt: raiz)
Inicio
  Si raiz <> nulo entonces
   Si raiz.izdo = nulo  y raiz.ddcho=nulo entonces
                        Cont  cont + 1
    Fin_si
```

Creación de un árbol binario.

```
Tipo Puntero_a_nodo_arbol : punt
Registro: nodo_arbol
     < tipo_elemento> : elemento
     punt: subiz, subder
fin_registro
```

8.6 COLECCIONES

Java dispone también de clases e interfaces para trabajar con colecciones de objetos, como las clases Vector y Hashtable, así como la interface Enumeration.
Estas clases están presentes en el lenguaje desde la primera versión.

8.6.1.-La clase Vector. (interfaz Enumeration)

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).

Vector representa un ***array de objetos*** (referencias a objetos de tipo ***Object***) que puede crecer y reducirse, según el número de elementos. Además permite acceder a los elementos con un ***índice***, aunque no permite utilizar los corchetes

Permite crear arrays especiales de objetos de forma ordenada, que pueden aumentar o disminuir el número de elementos a medida de las necesidades del programa, de forma dinámica. Se establece una capacidad inicial y posteriormente se va aumentando o disminuyendo esta capacidad.

La variable capacityIncrement, establece el incremento que debe experimentar la capacidad de la colección de objetos.

La clase contiene un constructor que admite como parámetros la capacidad inicial y el incremento, otro constructor sin argumentos, y un tercer constructor que admite como argumento sólo la capacidad inicial y deja por defecto el incremento.

Una vez construido el vector, añadir elementos, quitarlos o acceder a ellos, es muy fácil. Lo hacemos a través de los métodos:

void setElementAt(Object obj, int index)	Cambia el elemento que está en una determinada posición
void insertElementAt(Object obj, int index)	Inserta un elemento por delante de una determinada posición
addElement(Object o)	Añade un elemento al final
boolean removeElement(Object o) void removeAllElement() removeElementAt(int i)	Elimina el primer objeto cuya referencia coincida con el argumento. Elimina todos los objetos de la colección. Elimina el elemento indicado por el índice
elementAt(int i)	Accede al elemento que ocupa la posición indicada por i.
Object clone() void copyInto(Objeto miArray[ ])	Crea una copia de la colección. Copia la colección en un Array
boolean contains (Object o)	Investiga si la colección contiene o no un determinado objeto.
int indexOf (Object o) int indexOf (Object o, int indice)	Devuelve la posición en la que se encuentra un objeto.

int lastindexOf (Object o)	Dev. la posición donde se encuentra un obj. a partir del elemento que indica el índice.
boolean isEmpty()	Investiga si está vacio
Object firstElement() /Object lastElement()	Devuelve el objeto que está en la primera/última posición ocupada

8.6.3.- La clase Hashtable. (interfaz Enumeration)

Permite crear listas de datos con índice de acceso, creando códigos de direccionamiento para cada dato. Esta clase tiene tres constructores:

- El primero permite especificar el tamaño inicial y un factor de crecimiento.
- El segundo permite especificar el tamaño inicial y un factor de crecimiento por defecto.
- El tercero toma por defecto el tamaño inicial y el factor de crecimiento.

El factor de crecimiento es un float entre 0 y 1 de forma que 0,8 indica que cuando se llegue al 80% de ocupación, debe incrementar el tamaño de la lista, y redireccionar todos los datos insertados en la lista

Para recorrer la colección creamos un objeto de la interfaz Enumeration

Métodos

- Object **put(Object clave, Object valor)** permite insertar objetos en la lista tomando como argumento el identificador y el valor a guardar en la lista.

```
import java.util.*;
import java.io.*;
public class VerHashtable {
  public static void main(String args[]){
    String atributo= null, valor;
    Hashtable tabla = new Hashtable(10,0.75f);
    Scanner tec = new Scanner(System.in);
    System.out.println("Escribe parejas de datos");
    do{
      try{
        System.out.println("Atributo : (* finaliza)");
        atributo=tec.nextLine();
        if(atributo.charAt(0) != '*'){
              System.out.println("Valor: ");
             valor=tec.nextInt();
             tabla.put(atributo,valor);   }
       } catch(Exception e) {
           System.out.println(e.getMessage());  }
      } while (atributo.charAt(0) != '*');
```

- Object **Get(Object clave)** permite recuperar el valor de un dato identificado por su clave.
- boolean **hasMoreElements()** investiga si hay más elementos en la lista.
- **NextElement()** avanza hacia la siguiente referencia del siguiente dato.

8.6.4.- La clase HashSet.

Permite construir listas en las que no puede haber repeticiones de datos. La lista creada está desordenada y el método **add()** permite insertar un dato en la lista. Para acceder a los datos hay que utilizar un objeto de la interfaz **Iterator**.

```
import java.util.*;
public class VerTreeMap {
 public static void main(String args[]){
   TreeMap lista = new TreeMap();
   lista.put("1","Lunes");
   lista.put("2","Martes");
   lista.put("4","Jueves");
   lista.put("7","Domingo");
lista.put("5","Viernes");
   lista.put("6","Sabado");
   lista.put("3","Miercoles");
   lista.put("6","Sabado");
   System.out.println("La lista tiene "+lista.size()+" elementos");
   System.out.println("La semana \"ordenada y sin repeticiones\"");
     Collection coleccion = lista.values();
     Iterator i = coleccion.iterator();
     Collection colec = lista.keySet();
     Iterator j = colec.iterator();
   System.out.println ( lista );
   while(i.hasNext())
        System.out.println("\t"+j.next()+"\t\t"+i.next());
    }
}
```

8.6.5.- La clase linkedList.

Permite crear listas ordenadas con elementos repetidos. Los métodos addFirst() y addLast() permiten insertar un elemento al principio y al final de la lista. Para acceder a los datos hay que utilizar un objeto de la interfaz ListIterator.

ejem. VerLinkedList -- *Visualiza:*

8.6.6.- La clase TreeMap. (interface Collection)

Permite crear listas ordenadas sin elementos repetidos mediante pares de claves y valores. El métodos put permite añadir insertar un elemento a la lista asociada a una clave. Para ver los datos hay que utilizar un objeto de la **interfaz Iterator**, para lo cual es necesario crear un objeto de la interface **Collection**.

Ejem. VerArbol – *Visualiza :*

```
La semana "ideal"
        Domingo
        Martes
        Miercoles
        Jueves
        Viernes
        Sabado
        Sabado
```

```
La lista tiene 7 elementos
La semana "ordenada y sin repeticiones"
{1=Lunes, 2=Martes, 3=Miércoles, 4=Jueves, 5=Viernes, 6=Sábado, 7=Domingo}

1       Lunes
2       Martes
3       Miercoles
4       Jueves
5       Viernes
6       Sabado
7       Domingo
```

8.7.- Interfaces

Collection	HashSet	Iterator
Set	List	ListIterator
SortedSet	Map	Comparable
TreeSet	SortedMap	Comparator

- ***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.***

A partir del nombre, de los argumentos y del valor de retorno, la mayor parte de estos métodos resultan auto explicativos.

```
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[])[];
}
```

- 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.

- **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 no hay más elementos.

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());
}
```

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***.

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 *listas* 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.

- **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();
}
```

- *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***. En ***Java 1.2*** esta interface está implementada -entre otras- por las clases ***String***, ***Character***, ***Date***, ***File***, ***BigDecimal***, ***BigInteger***, ***Byte***, ***Short***, ***Integer***, ***Long***, ***Float*** y ***Double***.

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***

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
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 1.2 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***.

- *Sets y SortedSets*

La interface ***Set*** sirve para acceder a una colección sin elementos repetidos. La colección puede estar o no ordenada .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:

- 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 **Hashtable** 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.

- 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 :***

```
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.Objectput(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();
```

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***.

```
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);
}
}
```

Esta "vista" del ***Map*** como ***Set*** permite modificar y eliminar elementos del ***Map***, pero no añadir nuevos elementos.

- El método ***getkey()*** 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 ***Map*** 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.

```
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);
      }
```

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:

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.

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()***.

8.7.-Clases de propósito general:

Son las implementaciones de las interfaces de la *Java Collections Framework* (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 en el 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***.

8.7.1.- 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 staticos con diversas finalidades. 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;

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.

8.8.- Otras clases del paquete java.util.

El package ***java.util*** tiene otras clases interesantes para aplicaciones de distinto tipo, entre ellas algunas destinadas a considerar todo lo relacionado con fechas y horas

- La clase Random**.** Crea objetos que generan números aleatorios

import java.lang.*;

- **Clase Date.**

La clase ***Date*** representa un instante de tiempo dado con precisión de milisegundos.

La información sobre fecha y hora se almacena en un entero ***long*** de 64 bits, que contiene los milisegundos transcurridos desde las 00:00:00 del 1 de enero de 1970 GMT (*Greenwich mean time*). Ya se verá que otras clases permiten a partir de un objeto ***Date*** obtener información del año, mes, día, hora, minuto y segundo. A continuación se muestran los métodos de la clase ***Date***, habiéndose eliminado los métodos declarados obsoletos (*deprecated*) en el JDK 1.2:

public class java.util.Date extends java.lang.Object **implements** java.io.Serializable,
java.lang.Cloneable, java.lang.Comparable {

- El constructor por defecto ***Date()*** crea un objeto a partir de la fecha y hora actual del ordenador.
- El segundo constructor crea el objeto a partir de los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT.
- Los métodos ***after()*** y ***before()*** permiten saber si la fecha indicada como argumento implícito (***this***) es posterior o anterior a la pasada como argumento explícito.
- Los métodos ***getTime()*** y ***setTime()*** permiten obtener o establecer los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT para un determinado objeto ***Date***.
- Otros métodos son consecuencia de las interfaces implementadas por la clase ***Date***.

Los objetos de esta clase no se utilizan mucho directamente, sino que se utilizan en combinación con las siguientes clases:

- **Clases Calendar y GregorianCalendar.**

La clase ***Calendar*** es una clase ***abstract*** que dispone de métodos para convertir objetos de la clase ***Date*** en enteros que representan fechas y horas concretas.

La clase ***GregorianCalendar*** es la única clase que deriva de ***Calendar*** y es la que se utilizará normalmente.

Java tiene una forma un poco particular para representar las fechas y horas:

1.- Las horas se representan por enteros de 0 a 23 (la hora "0" va de las 00:00:00 hasta la 1:00:00), y los minutos y segundos por enteros entre 0 y 59.

2.- Los días del mes se representan por enteros entre 1 y 31 (lógico).

3.- Los meses del año se representan mediante enteros de 0 a 11 (no tan lógico).

4.- Los años se representan mediante enteros de cuatro dígitos. Si se representan con dos dígitos, se resta 1900. Por ejemplo, con dos dígitos el año 2000 es para Java el año 00.

La clase ***Calendar*** tiene una serie de variables miembro y constantes (variables ***final***) que pueden resultar muy útiles:

- La variable ***int*** AM_PM puede tomar dos valores: las constantes enteras AM y PM.
- La variable ***int*** DAY_OF_WEEK puede tomar los valores ***int*** SUNDAY, MONDAY,TUESDAY, WEDNESDAY, THURSDAY, FRIDAY y SATURDAY.
- La variable ***int*** MONTH puede tomar los valores ***int*** JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER. Para hacer los programas más legibles es preferible utilizar estas constantes simbólicas que los correspondientes números del 0 al 11.
- La variable miembro HOUR se utiliza en los métodos ***get()*** y ***set()*** para indicar la hora de la mañana o de la tarde (en relojes de 12 horas, de 0 a 11). La variable HOUR_OF_DAY sirve para indicar la hora del día en relojes de 24 horas (de 0 a 23).
- Las variables DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH, DAY_OF_MONTH (o bien DATE), DAY_OF_YEAR, WEEK_OF_MONTH, WEEK_OF_YEAR tienen un significado evidente.
- Las variables ERA, YEAR, MONTH, HOUR, MINUTE, SECOND, MILLISECOND tienen también un significado evidente.
- Las variables ZONE_OFFSET y DST_OFFSET indican la zona horaria y el desfase en milisegundos respecto a la zona GMT.

La clase ***Calendar*** dispone de un gran número de métodos para establecer u obtener los distintos valores de la fecha y/u hora. .

La clase ***GregorianCalendar*** añade las constante BC y AD para la ERA, que representan respectivamente antes y después de Jesucristo. Añade además varios constructores que admiten como argumentos la información correspondiente a la fecha/hora y –opcionalmente– la zona horaria.

- clases DateFormat y SimpleDateFormat.

DateFormat es una clase ***abstract*** que pertenece al package ***java.text*** y no al package ***java.util***, como las anteriores. La razón es para facilitar todo lo referente a la ***internacionalización***, que es un aspecto muy importante en relación con la conversión, que permite dar formato a fechas y horas de acuerdo con distintos criterios locales.

Esta clase dispone de métodos ***static*** para convertir ***Strings*** representando fechas y horas en objetos de la clase ***Date***, y viceversa.

La clase ***SimpleDateFormat*** es la única clase derivada de ***DateFormat***. Es la clase que conviene utilizar. Esta clase se utiliza de la siguiente forma: se le pasa al constructor un ***String*** definiendo el formato que se desea utilizar. Por ejemplo:

- Clases TimeZone y SimpleTimeZone.

La clase ***TimeZone*** es también una clase ***abstract*** que sirve para definir la zona horaria. Los métodos de esta clase son capaces de tener en cuenta el cambio de la hora en verano para ahorrar energía.

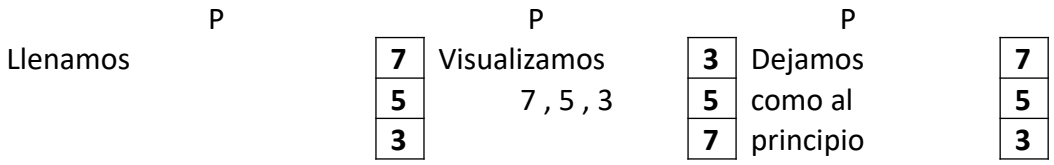
La clase ***SimpleTimeZone*** deriva de ***TimeZone*** y es la que conviene utilizar.

```
import java.util.*;
public class PruebaFechas {
  public static void main(String arg[]) {
    Date d = new Date();
    GregorianCalendar gc = new GregorianCalendar();
    gc.setTime(d);
    System.out.println("Era: "+gc.get(Calendar.ERA));
    System.out.println("Year: "+gc.get(Calendar.YEAR));
    System.out.println("Month: "+gc.get(Calendar.MONTH));
    System.out.println("Dia del mes: "+gc.get(Calendar.DAY_OF_MONTH));
    System.out.println("D de la S en mes:" +gc.get(Calendar.DAY_OF_WEEK_IN_MONTH));
    System.out.println("No de semana: "+gc.get(Calendar.WEEK_OF_YEAR));
    System.out.println("Semana del mes: "+gc.get(Calendar.WEEK_OF_MONTH));
    System.out.println("Fecha: "+gc.get(Calendar.DATE));
    System.out.println("Hora: "+gc.get(Calendar.HOUR));
    System.out.println("Tiempo del día: "+gc.get(Calendar.AM_PM));
    System.out.println("Hora del dia: "+gc.get(Calendar.HOUR_OF_DAY));
    System.out.println("Minuto: "+gc.get(Calendar.MINUTE));
    System.out.println("Segundo: "+gc.get(Calendar.SECOND));
    System.out.println("Dif. horaria: "+gc.get(Calendar.ZONE_OFFSET));
    }
}
```

El valor por defecto de la zona horaria es el definido en el ordenador en que se ejecuta el programa. Los objetos de esta clase pueden ser utilizados con los constructores y algunos métodos de la clase ***Calendar*** para establecer la zona horaria.

Prácticas tema 8 Dinámicas

1. Crea una Pila de nombres y realiza las siguientes operaciones:
Llena la pila visualizando los elementos que introduces.
Indica cuantos elementos tiene , visualízalos en el orden en que están almacenados.



2- Crea una Pila con las notas de 8 alumnos (0-10) .
Calcula la nota media y visualízala.
Guarda en la pila sólo los elementos superiores a la media.

3.Carga una pila de enteros y cambia de signo todos sus elementos, sin alterar la estructura de la pila.

4.Carga una pila de 10 elementos (String / enteros) y visualízala ordenada.