Estructuras de datos dinámicas.

8.1.- Introducción

8.2.- Listas.

8.3.- Pilas.

8.4.- Colecciones.

8.4.1.- La clase Arrays.

8.4.2.-Interfaces.

8.4.3.- Clases.

8.4.4.-Algoritmos.

8.5.- La clase Date paquete java.util.

Tipos de datos. simples (sin estructura) y compuestos (estructurados).

- **Tipos de datos**
  - **Simples**
    - **Numéricos** (integer, real).
    - **Lógicos** (boolean).
    - **Carácter** (char, String).
  - **Estructuras de datos**
    - **Internas**
      - **Estáticas** (Vectores, Matrices)
      - **Dinámicas (Listas, Pilas, Colas)**
    - **Externas**
      - **Ficheros**
      - **Bases de Datos**

ESTRUCTURAS DE DATOS DINÁMICAS .

- Una estructura de **datos dinámica** es una colección de elementos denominados **nodos de la estructura.**
- Las **estructuras dinámicas** de datos se clasifican en :

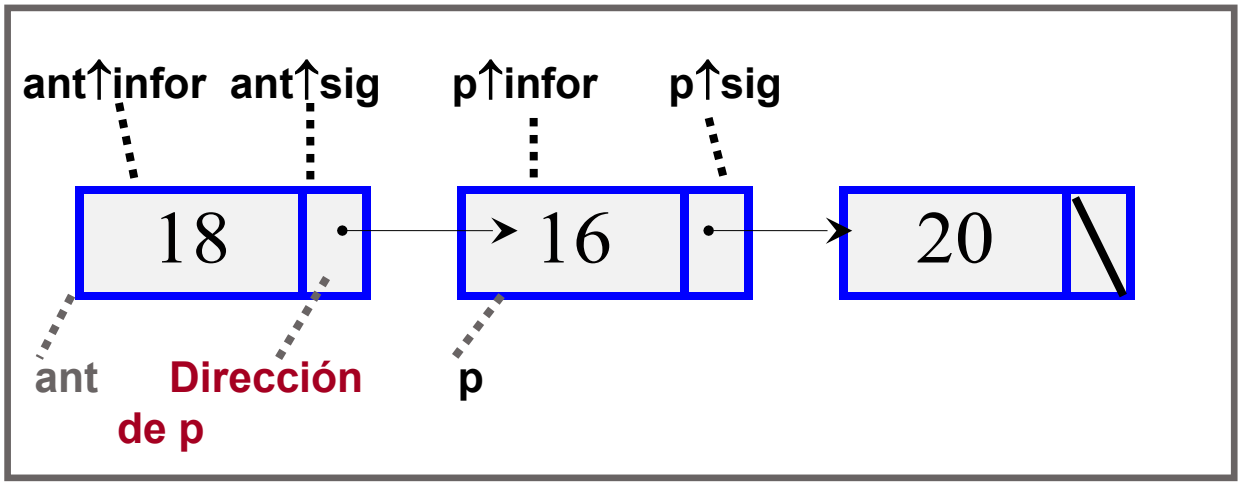
Lineales	Listas enlazadas	Simples
		Dobles
		Circulares
	Pilas	
	Colas	
No lineales	Árboles	
	Grafos	

ESTRUCTURAS DE DATOS DINÁMICAS

- ***LISTAS ENLAZADAS*.** Una lista enlazada es un conjunto de elementos denominados **nodos.**
- Un nodo tiene al menos, un campo de datos y un enlace (puntero) con el siguiente nodo de la lista.
- El campo enlace, apunta, (proporciona la dirección de el siguiente nodo).
- El último nodo de la lista se suele representar por un enlace con la palabra reservada **null** (nulo) o con una barra inclinada.

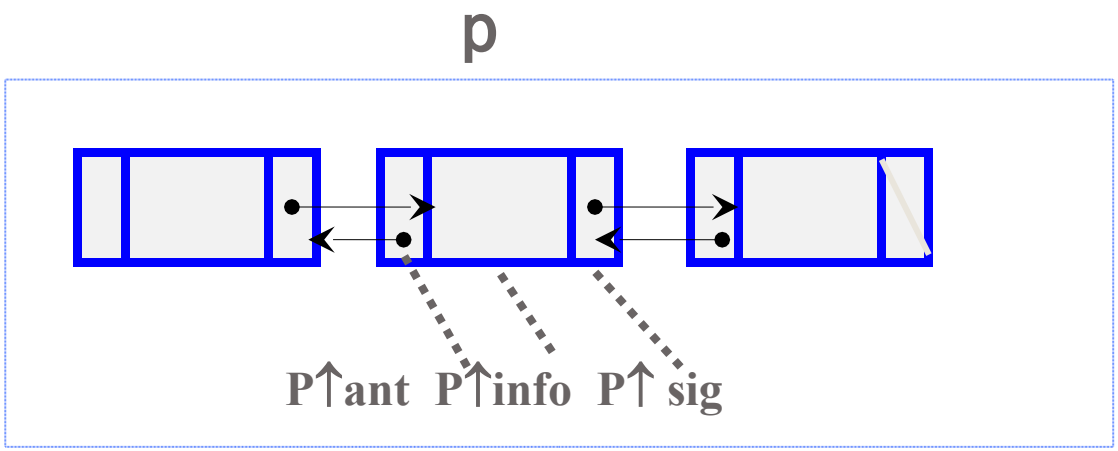
ESTRUCTURAS DE DATOS DINÁMICAS

Representación : *LISTAS ENLAZADAS*



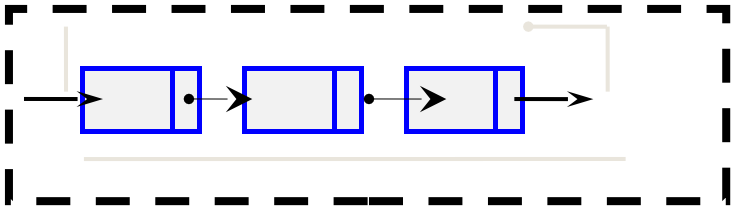
- A partir de ahora utilizaremos el término **puntero (p)** para describir el enlace entre dos elementos o nodos de una lista enlazadas.

- ***LISTAS DOBLEMENTE ENLAZADAS*** **.** Se pueden recorrer en dos direcciones, izquierda y derecha. Están formadas por un campo de información y dos punteros , uno apunta al nodo siguiente y otro al anterior.



ESTRUCTURAS DE DATOS DINÁMICAS

- LISTAS CIRCULARES. Son aquellas en las que el último elemento de la lista apunta al primero o al principio de la lista.



Ventajas

- Cada nodo es accesible desde cualquier otro nodo.
- Las operaciones de concatenación o división son más eficaces en listas circulares.

Inconvenientes:

- Se pueden producir bucles infinitos.

ESTRUCTURAS DE DATOS DINÁMICAS

- **PILA.** 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** (top).
- 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.

ESTRUCTURAS DE DATOS DINÁMICAS

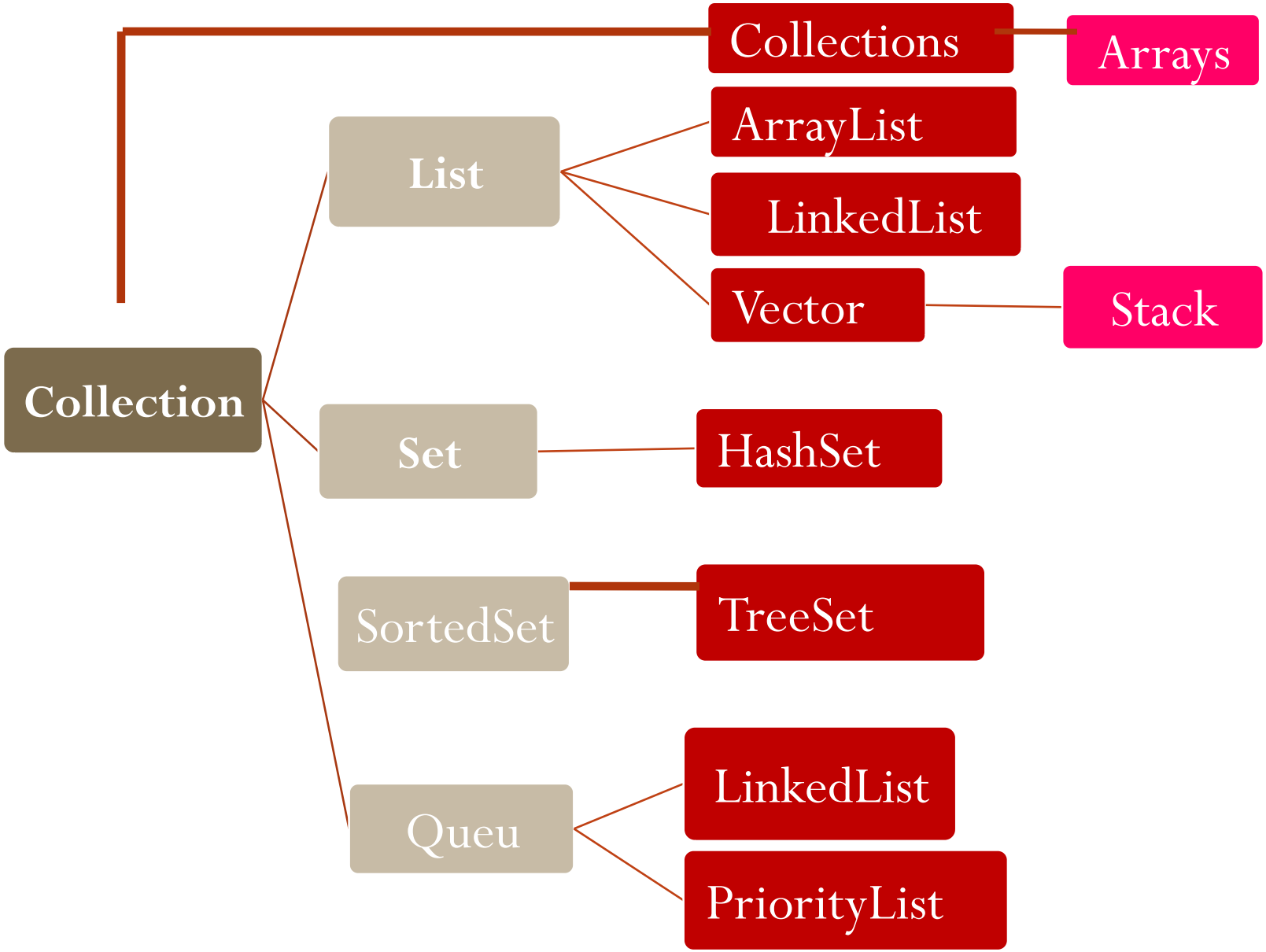
- **COLAS .** Es un tipo especial de lista lineal en la que los elementos **se insertan sólo por el final** y se borran y **se sacan sólo por el principio.**

- Las operaciones básicas en una cola son:
  - **Insertar o poner un elemento**
  - **Borrar o sacar un elemento**
  - **Preguntar si está vacía**.

Cada una de estas operaciones , como en las pilas , se debe realizar en una método.

COLECCIONES.

- Java dispone de un conjunto de clases e interfaces (API) que facilitan la tarea del programador para trabajar con colecciones de objetos.
- **Las colecciones** trabajan como los arrays, pero **pueden modificar su tamaño de forma dinámica**, y poseen operaciones (métodos) más avanzados que los arrays.
- La mayoría de las colecciones se encuentran en el paquete **java.util**


Collections
Arrays
List
ArrayList
LinkedList
Vector
Stack
Collection
Set
HashSet
SortedSet
TreeSet
Queu
LinkedList
PriorityList

La clase Arrays

Esta clase proporciona métodos para manipular arrays.

METODOS	DESCRIPCIÓN
sort()	Ordena un array de forma ascendente.
birarySearch()	Busca un elemento en un array ordenado.
equals()	Compara arrays.
fill()	Coloca valores en un array.

Ejemplo. Clase Arrays

```
package tema8;
import java.util.Arrays;
/**
 *
 * @author Mariana
 */
public class VerArrays {
 private static int arrayInt[]={2, 4, 6, 8, 10};
 private static double arrayDou[]={2.5, 4.5, 6.5, 8.5, 10.5};
 private static int arrayLleno[], copiaArray[];

 public static void main(String arg[]){
   arrayLleno=new int[10];
   copiaArray= new int[arrayInt.length];

   Arrays.fill(arrayLleno, 3); // llenamos de treses el array
   Arrays.sort(arrayDou);// Ordena el array de dobles
```

```
System.arraycopy(arrayInt,0,copiaArray,0,arrayInt.leng
//Imprimimos todos los arrays
System.out.print("Array de enteros :");
for(int arrInt:arrayInt) {
        System.out.printf("%d   " , arrInt);}
        System.out.println("\n");


System.out.print("Array de reales : ");
for(double arrD:arrayDou) {
        System.out.printf("%.2f   " , arrD);
}
        System.out.println("\n");

System.out.print("Array Lleno de treses : ");
for(int arrInt: arrayLleno) {
        System.out.printf("%d   " , arrInt);
   }
```

d indica entero

% indica la parte entera

. y **2** decimales

f el tipo de dato

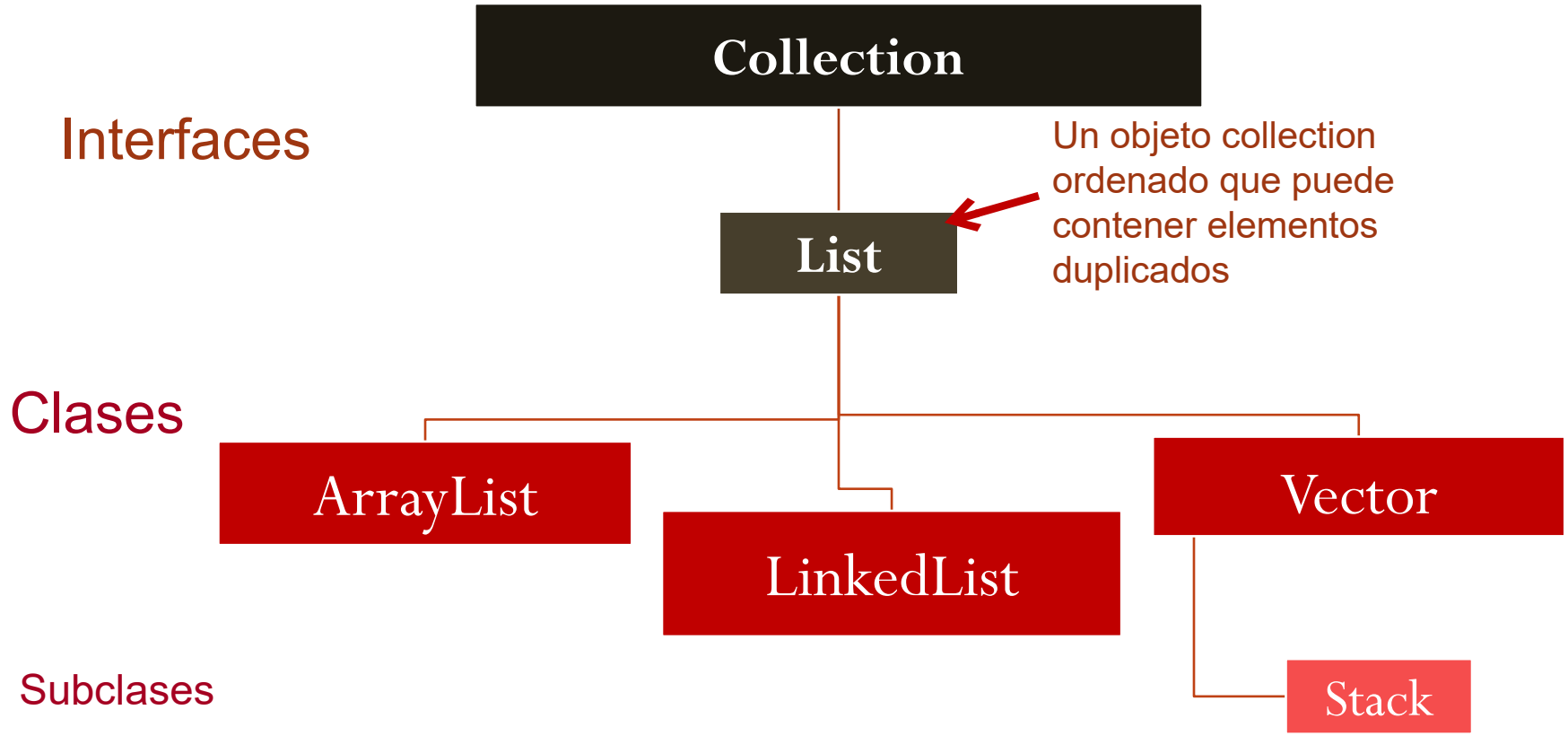
```
System.out.print("El Array copia : ");
  for(int arrInt: copiaArray) {
        System.out.printf("%d  " , arrInt);
    }
  System.out.println("\n");
     /* Buscamos elementos en un array ( nos devuelve la posición si está
  o un valor negativo en caso contrario */
  Arrays.binarySearch(arrayInt, 6);
  System.out.println(Arrays.binarySearch(arrayInt, 6));

  //Compara el array de enteros con la copia
  boolean b= Arrays.equals(arrayInt, copiaArray);

  System.out.printf("arrayInt %s copiaArray\n", (b ? "==" : "!=") );

  b= Arrays.equals(arrayInt, arrayLleno);
  System.out.printf("arrayInt %s arrayLleno\n", (b ? "==" : "!=") );
 }
}
```

s cadena / string

Si b es cierto
? Entonces
Sentencia_1 ("==")
: else
Sentencia_2 ("!=")

Condicional tipo C

Interface List



La clase ArrayList

- Una de las implementaciones mas importantes de la interfaz List es la clase ArrayList.
- Creación:

- ArrayList ejem1= new ArrayList();
- ArrayList <tipo objeto> ejem2=new ArrayList<tipo objeto>();

Como cualquier otro objeto

Indicamos el tipo de objetos que va a contener.

En el primer caso podemos guardar cualquier objeto.

En el segundo, solo objetos del tipo indicado.

Formas de recorrer y leer todas las posiciones de un ArrayList

Mediante un **bucle for**:

```
String valor;
 for (int i=0; i<ejem1.size(); i++){
    valor= ejem1.get(i);
    System.out.println("contenido : " + valor);
 }
```

Mediante un **bucle for each:**

```
for (String valor : ejem1){
    System.out.println("contenido : " + valor);
    }
```

Si suponemos el array de enteros llamado **numeros***:*

```
for(Integer n: numeros){
        System.out.println(n);
        }
```

Si el array contiene **objetos de tipos distintos** o desconocemos el tipo:

```
for(Object o: nombreArray){
        System.out.println(o);
            }
```

Mediante un **objeto Iterator**

La ventaja de utilizar un Iterador es que no necesitamos indicar el tipo de objetos que contiene el array.

hasNext(): devuelve true si hay más elementos en el array.
Next(): devuelve el siguiente objeto contenido en el array.

Ejemplo ArrayList e Iterator

```
ArrayList<Integer> numeros = new ArrayList<Integer>();
…..
// llenamos el Array numeros
.....
Iterator it = numeros.iterator();
// se crea el iterador it para el array numeros
while(it.hasNext())       {                  // mientras queden elementos
        System.out.println(it.next());  //se obtiene y se muestra
}
```

ArrayList e Iterator.

MÉTODO	DESCRIPCIÓN
size()	Devuelve el número de elementos (int)
add(X)	Añade el objeto X al final. Devuelve true.
add(posición, X)	Inserta el objeto X en la posición indicada.
get(posicion)	Devuelve el elemento que está en la posición indicada.
remove(posicion)	Elimina el elemento que se encuentra en la posición indicada. Devuelve el elemento eliminado.
remove(X)	Elimina la primera ocurrencia del objeto X. Devuelve true si el elemento está en la lista.
clear()	Elimina todos los elementos.
set(posición, X)	Sustituye el elemento que se encuentra en la posición indicada por el objeto X. Devuelve el elemento sustituido.
contains(X)	Comprueba si la colección contiene al objeto X. Devuelve true o false.
indexOf(X)	Devuelve la posición del objeto X. Si no existe devuelve -1

Ejemplo. Clase ArrayList

```
public class PruebaCollection {

    private static final String[] colores
            = {"MAGENTA", "ROJO", "BLANCO", "AZUL", " VER
    private static final String[] eliminarColores
            = {"ROJO", "BLANCO", "AZUL"};

    @SuppressWarnings("ManualArrayToCollectionCopy")
    public static void main(String argm[]) {
        List<String> lista = new ArrayList<String>();
         List<String> eliminarLista;
         eliminarLista = new ArrayList<String>();

// Pasamos los elementos del array  colores a la lista co
// con foreache
         for (String color : colores) {
             lista.add(color);
         }
         // Igual con los elementos de eliminarColores
```

```
        for (String color : eliminarColores) {
            eliminarLista.add(color);
        }
        System.out.println("ArrayList: ");
        // Visualizamos la lista

        for (int i = 0; i < lista.size(); i++) {
            System.out.printf("%s ", lista.get(i));
        }
/ elimina los colores contenidos en eliminarLista
        eliminaColores(lista, eliminarLista);

        System.out.println("\n\nArrayList: despues de eliminar elementos");
        // Visualizamos la lista
        for (String color : lista) {
            System.out.printf("%s ", color);
        }
        System.out.println();
           }
```

```
     * @param lista
     * @param eliminarLista
     */
    public static void eliminaColores(List<String> lista, Collection<String> eliminarLista) {
        Iterator<String> it = lista.iterator();

// Repite mientras la colección tenga elementos
        while (it.hasNext()) {
            if (eliminarLista.contains(it.next())) {
                it.remove(); // elimina el color actual de la lista
            }
        }

    }
}
```

LinkedList

Permite crear listas **ordenadas** con elementos **repetidos**.

Los métodos addFirts() 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.**

```
import java.util.*;

public class VerLinkedList {

    public static void main(String args[]) {
        LinkedList lista = new LinkedList();
        lista.add("Martes");
        lista.add("Miercoles");
        lista.add("Jueves");
        lista.add("Viernes");
        lista.add("Sabado");
        lista.addLast("Sabado");
        lista.addFirst("Domingo");
        System.out.println("\tLa semana \"ideal\"");
        ListIterator i = lista.listIterator(0);
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

Ejemplo. Clase LinkedList

```
public class PruebaList_LinkedList {

    private static final String colores[]
            = {"negro", "amarillo", "verde", "verde", "azul", "violeta", "plateado"};
    private static final String colores2[]
            = {"dorado", "blanco", "cafe", "azul", "gris", "plateado"};

    @SuppressWarnings("ManualArrayToCollectionCopy")
    public static void main(String argm[]) {
        List<String> lista1 = new LinkedList<String>();
        List<String> lista2;
        lista2 = new LinkedList<String>();

// Pasamos los elementos del array  colores a la lista1  con foreache
        for (String color : colores) {
            lista1.add(color);
        }
        visualizar(lista1);
        // Igual con los elementos de colores2
        for (String color : colores2) {
            lista2.add(color);
        }
```

```
        visualizar(lista2);

// Concatenamos las listas
        lista1.addAll(lista2);
        lista2 = null;
        System.out.println("\nLinkedList: Lista1");
        visualizar(lista1);
        convertirAMayus(lista1);

        // Eliminamosloselementos del 4 al 7
        lista1.subList(3, 7).clear();
        lista1.add(3, "VIOLETA");
        System.out.println("\nLista con elementos eliminados");
        visualizar(lista1);
        lista1.add(3, "VIOLETA");

        // visualizamos desde atrás hacia adelante
        System.out.println("\nLista desde atrás");
        visualizaDesdeAtras(lista1);
    }
```

```
static void visualizar(List<String> lista) {

    for (String color : lista) {
        System.out.printf("%s ", color);
    }
    System.out.println();
}
static void visualizaDesdeAtras(List<String> lista) {
    ListIterator<String> it = lista.listIterator(lista.size());

    while (it.hasPrevious()) {
        System.out.printf("%s ", it.previous());
    }
    System.out.println();
}
static void convertirAMayus(List<String> lista) {
    ListIterator<String> it = lista.listIterator();
    String color;
    while (it.hasNext()) {
        color = it.next();
        it.set(color.toUpperCase());
    }
```

La clase Vector.

- Permite crear arrays especiales de objetos de forma **ordenada,** que pueden **aumentar o disminuir** el número de sus 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.
- Al igual que un array de objetos, estos pueden ser accedidos por un índice.
- La variable **capacityIncrement**, establece el incremento que debe experimentar la capacidad de la colección de objetos

La clase Vector.

- La clase contiene un constructor que admite como parámetros la capacidad inicial y el incremento.
- También cuenta con un constructor sin argumentos.
- 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:

La clase Vector. **métodos**

MÉTODO	DESCRIPCIÓN
addElement(Object o)	Añade un elemento al final
boolean removeElement(Object o)	Elimina el primer objeto cuya referencia coincida con la del argumento.
void removeAllElement()	Elimina todos los objetos de la colección.
removeElementAt(int)	Elimina el elemento indicado por el índice
ElementAt(int)	Accede al elemento que ocupa la posición indicada en el argumento
Object clone()	Crea una copia de la colección.
void copyInto(Objeto miArray[ ])	Copia la colección en un Array.

La clase Vector. **métodos**

MÉTODO	DESCRIPCIÓN
boolean contains (Object o)	Investiga si la colección contiene o no un determinado objeto.
int indexOf (Object o) int lastindexOf (Object o) int indexOf (Object o, int indice)	Devuelve la posición donde se encuentra un obj. a partir del elemento que indica el índice.
boolean isEmpty()	Investiga si está vacío
Object firstElement() Object lastElement()	Devuelve el objeto que está en la primera/última posición ocupada

Ejemplo. Clase Vector

```
// Uso de la clase Vector.
import java.util.Vector;
import java.util.NoSuchElementException;

public class PruebaVector
{
   private static final String colores[] = { "rojo", "blanco", "azul" };

   public PruebaVector()
   {
      Vector< String > vector = new Vector< String >();
      imprimirVector( vector ); // imprime el vector

      // agrega elementos al vector
      for ( String color : colores )
         vector.add( color );

      imprimirVector( vector ); // imprime el vector

      // imprime los elementos primero y último
      try
      {
         System.out.printf( "Primer elemento: %s\n", vector.firstElement());
         System.out.printf( "Ultimo elemento: %s\n", vector.lastElement() );
      } // fin de try
      // atrapa la excepción si el vector está vacío
      catch ( NoSuchElementException excepcion )
      {
         excepcion.printStackTrace();
      } // fin de catch

      // ¿el vector contiene "rojo"?
      if ( vector.contains( "rojo" ) )
         System.out.printf( "\se encontro \"rojo\" en el indice %d\n\n",
            vector.indexOf( "rojo" ) );
      else
         System.out.println( "\no se encontro \"rojo\"\n" );

      vector.remove( "rojo" ); // elimina la cadena "rojo"
      System.out.println( "se elimino \"rojo\" );
      imprimirVector( vector ); // imprime el vector
```

```
      // ¿el vector contiene "rojo" después de la operación de eliminación?
      if ( vector.contains( "rojo" ) )
         System.out.printf(
            "se encontro \"rojo\" en el indice %d\n", vector.indexOf( "rojo" ) );
      else
         System.out.println( "no se encontro \"rojo\"" );

      // imprime el tamaño y la capacidad del vector
      System.out.printf( "\nTamanio: %d\nCapacidad: %d\n", vector.size(),
         vector.capacity() );
   } // fin del constructor de PruebaVector

   private void imprimirVector( Vector< String > vectorAImprimir )
   {
      if ( vectorAImprimir.isEmpty() )
         System.out.print( "el vector esta vacio" ); // vectorAImprimir está vacío
```

```
      else  // itera a través de los elementos
      {
         System.out.print( "el vector contiene: " );

         // imprime los elementos
         for ( String elemento : vectorAImprimir )
            System.out.printf( "%s ", elemento );
      } // fin de else

      System.out.println( "\n" );
   } // fin del método imprimirVector

   public static void main( String args[] )
   {
      new PruebaVector(); // crea objeto y llama a su constructor
   } // fin de main
} // fin de la clase PruebaVector
```

Resultados

```
el vector esta vacio

el vector contiene: rojo blanco azul

Primer elemento: rojo
Ultimo elemento: azul

se encontro "rojo" en el indice 0

se elimino "rojo"
el vector contiene: blanco azul

no se encontro "rojo"

Tamanio: 2
Capacidad: 10
```

La clase Stack.

Permite crear objetos conocidos como pilas, es decir colecciones de objetos cuya característica principal es la forma de almacenamiento.

El último elemento que llega a la pila es el primero en salir. LIFO

MÉTODO	DESCRIPCIÓN
Object push(Object item) Object pop()	Permiten introducir y sacar elementos de la pila.
Object peek()	Obtiene una copia del último elemento en entrar en la pila sin sacarlo de esta.
boolean empty()	Comprueba si la pila está vacía.
int search(Object o)	Busca el objeto especificado en la pila . Retorna la distancia desde la parte alta de la pila hasta donde se encuentre, o –1 si no existe.
Los métodos pop() y peek() pueden generar una excepción del tipo EmptyStackException si la pila está vacía.	

Ejemplo. Stack LIFO

```
import java.util.*;
public class VerPila {

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

Ejem. Clase Stack

```
// Programa para probar la clase java.util.Stack.
import java.util.Stack;
import java.util.EmptyStackException;

public class PruebaStack
{
   public PruebaStack()
   {
      Stack< Number > pila = new Stack< Number >();

      // crea números para almacenarlos en la pila
      Long numeroLong = 12L;
      Integer numeroInt = 34567;
      Float numeroFloat = 1.0F;
      Double numeroDouble = 1234.5678;

      // usa el método push
      pila.push( numeroLong ); // mete un long
      imprimirPila( pila );
      pila.push( numeroInt ); // mete un int
      imprimirPila( pila );
      pila.push( numeroFloat ); // mete un float
      imprimirPila( pila );
      pila.push( numeroDouble ); // mete un double
      imprimirPila( pila );
```

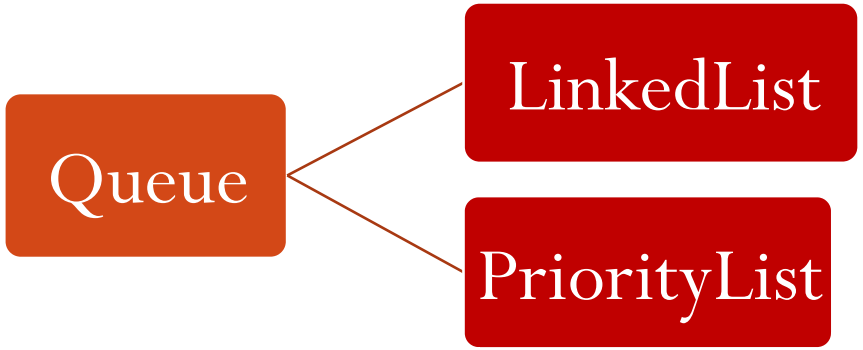
```
      // elimina los elementos de la pila
      try
      {
         Number objetoEliminado = null;

         // saca elementos de la pila
         while ( true )
      {
         objetoEliminado = pila.pop(); // usa el método pop
         System.out.printf( "%s se saco\n", objetoEliminado );
         imprimirPila( pila );
      } // fin de while
   } // fin de try
   catch ( EmptyStackException emptyStackException )
   {
      emptyStackException.printStackTrace();
   } // fin de catch
} // fin del constructor de PruebaStack

private void imprimirPila( Stack< Number > pila )
{
   if ( pila.isEmpty() )
      System.out.print( "la pila esta vacia\n\n" ); // la pila está vacía
   else  // la pila no está vacía
   {
      System.out.print( "la pila contiene: " );

      // itera a través de los elementos
      for ( Number numero : pila )
         System.out.printf( "%s ", numero );

      System.out.print( "(superior) \n\n" ); // indica la parte superior de la pila
   } // fin de else
} // fin del método imprimirPila
```

Completa una agenda de Contactos utilizando los métodos de la clase ArrayList para crear, ordenar, buscar, eliminar….

INTERFAZ java.util. Queu

Es una colección de objetos que utilizaremos para simular una cola FIFO.

clases



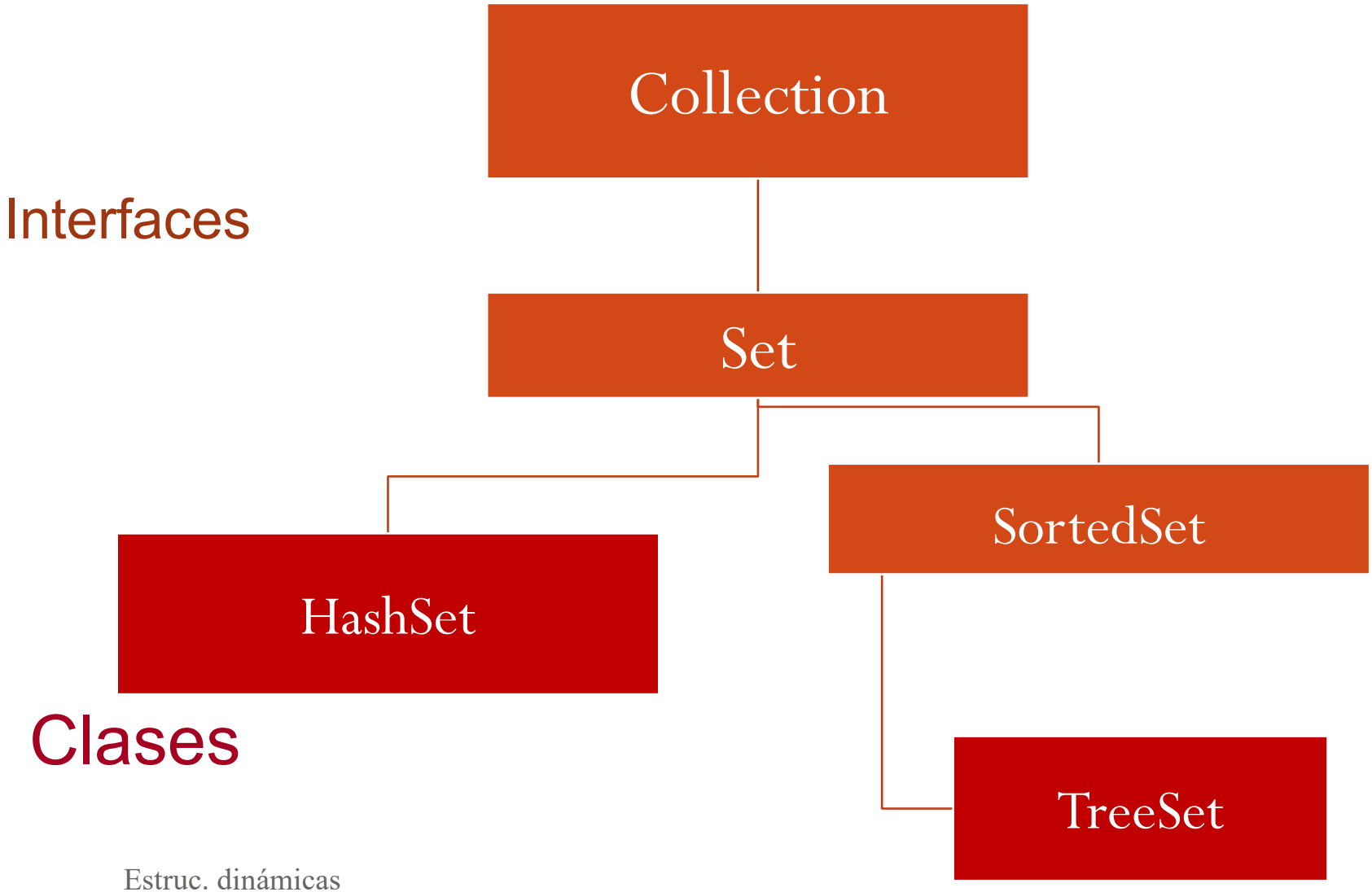
- La principal implementación de esta interfaz es LinkedList que además implementa la interfaz List y PriorityQueu

offer	Inserta un elemento en la cola
peek	Obtiene el primer elemento de la cola, pero no lo elimina.
poll	Obtiene el primer elemento de la cola, eliminándolo.
remove	Obtiene y elimina el primer elemento de la cola

PriorityQueue

- Permite inserciones en orden, y eliminaciones.
- Al añadir un elemento la inserción se realiza en orden de prioridad.

INTERFACE SET



La interface *Set*

- Sirve para acceder a una colección **sin** elementos **repetidos**.
- La colección puede estar o no ordenada .
- ***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

La interface *Set*

- Utilizando los métodos de ***Collection***, los ***Sets*** permiten realizar operaciones algebraicas de ***unión***, ***intersección*** y ***diferencia***

- **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*;
- **s1.removeAll(s2)** convierte ***s1*** en la diferencia entre ***s1*** y ***s2***.

La interface *SortedSet*

Extiende la interface ***Set*** y añade los siguientes métodos:

- ***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***.
- ***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 .

La interface *SortedSet*

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

Las clases HashSet y TreeSet

la clase ***HashSet*** implementa la interface ***Set.***

- Los elementos no mantienen el orden natural, ni el orden de introducción.
- Está basada en una hashtable .

la clase ***TreeSet*** implementa ***SortedSet***.

- Los elementos 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.

La clase HashSet – Ejemplo.

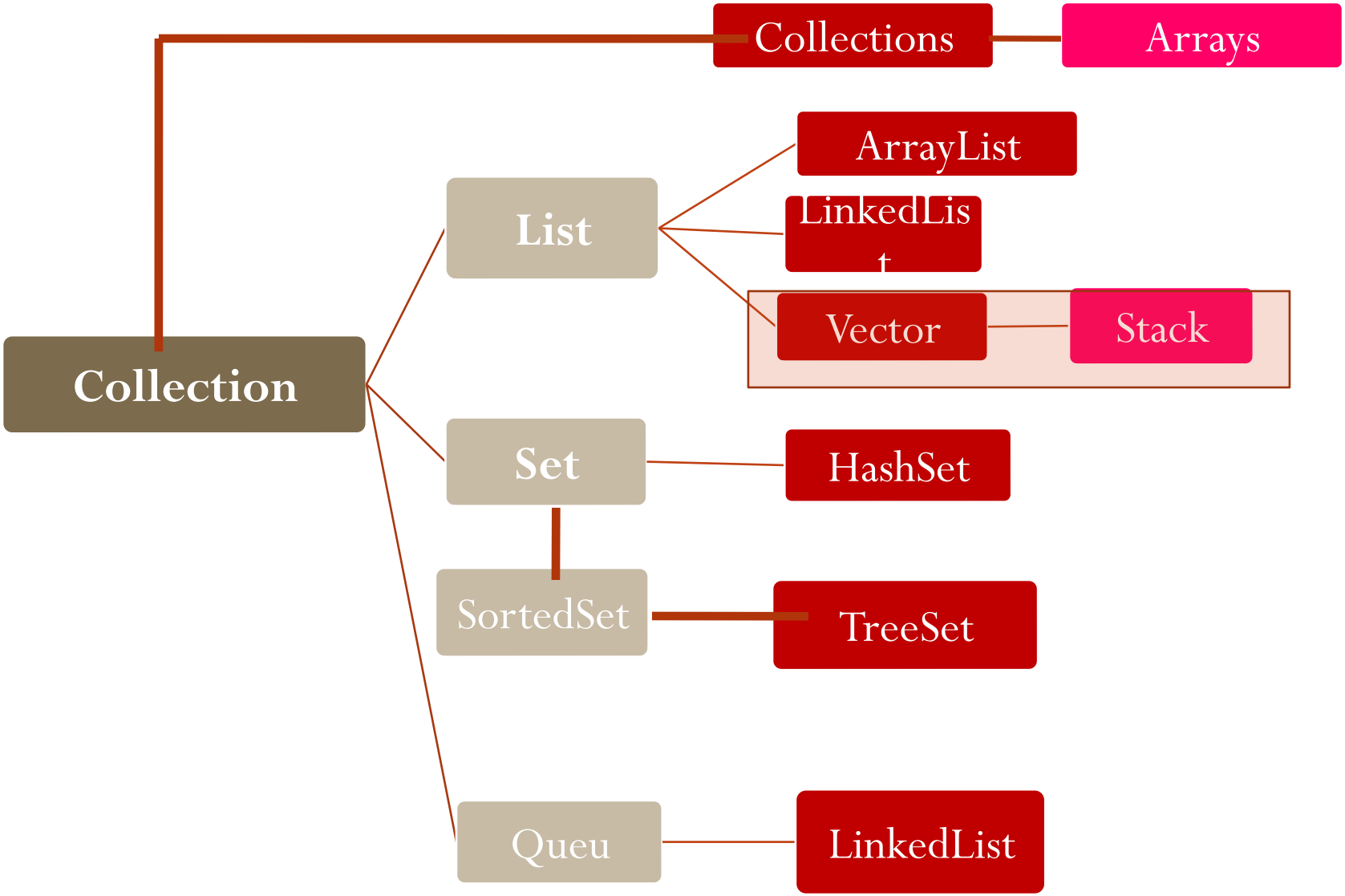
```
import java.util.*;
public class VerConjunto {


    public static void main(String args[]) {
        HashSet lista = new HashSet();
        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.println("La semana \"desordenada y sin repeticiones\"");
        Iterator i = lista.iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

HashSet

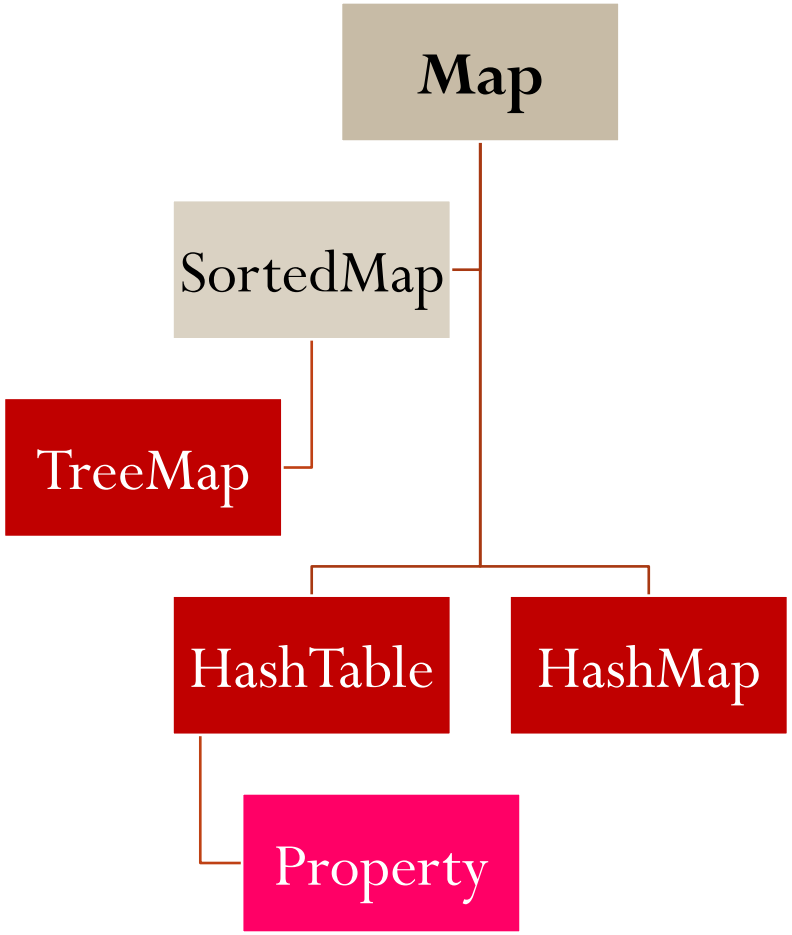
Permite construir listas en las que **no** puede haber **repeticiones** de datos.

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**.

Resumen:

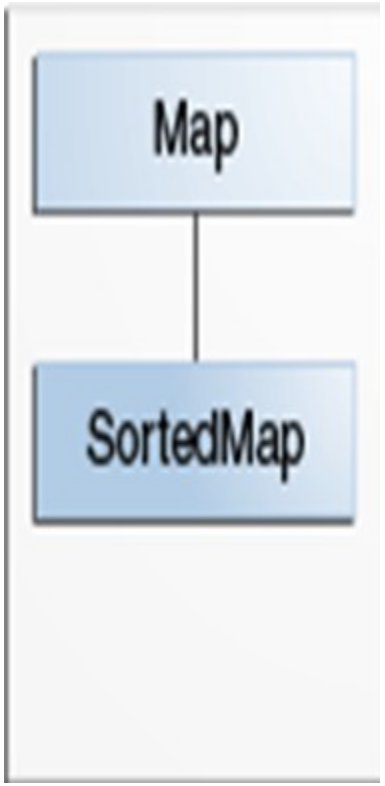

Collections
Arrays
ArrayList
List
LinkedLis
t
Vector
Stack
Collection
Set
HashSet
SortedSet
TreeSet
Queu
LinkedList

INTERFACE Map



Mapas

- Los objetos Map asocian claves a valores y no pueden contener claves duplicadas.
- Se diferencian de los objetos Set en que estos solo podían contener valores.



Map

- Los objetos HashTable y HashMap almacenan elementos en tablas de hash, los TreeMap almacenan elementos en arboles.
- Tres clases de las muchas que implementan esta interface son:
    - Hashtable
    - Hashmap
    - Treemap

La clase Hashtable

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.

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.

La clase Hashtable

- La clase Hashtable representa un tipo de colección basada en claves, donde los objetos almacenados en la misma (valores) no tienen asociado un índice numérico basado en su posición, sino una clave que lo identifica de forma única dentro de la colección. Una clave puede ser cualquier tipo de objeto.
- 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.

La clase Hashtable

- Por ejemplo, si se va a gestionar una colección de objetos de tipo Empleado, puede resultar mas práctico almacenarlos en un Hastable 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.

Principales métodos Hashtable

MÉTODO	DESCRIPCIÓN
Object put (Object key, Object valor)	permite insertar objetos en la lista tomando como argumento el identificador y el valor a guardar en la lista
Object get (Object key)	Devuelve el valor en forma de objeto que tiene asociada la clave que se indica en el parámetro
boolean containsKey(Object key)	Indica si la clave especificada existe ya en la colección.
Object remove(Object key)	**Elimina de la colección, el valor cuya clave se especifica en el parámetro.**

Para recorrer un Hashtable utilizaremos un objeto de la interfaz Enumeration.

La interfaz Enumeration dispone de los siguientes métodos:

- Object nextElement(). Provoca que este pase a apuntar al siguiente objeto de la colección, devolviendo el nuevo objeto apuntado.

- boolean hasMoreElements(). Indica si hay mas elementos por recorrer en la colección.

  Cuando el objeto Enumeration esté apuntando al último elemento, la llamada al método devolverá false.

La clase Properties

- Un objeto Properties es un objeto Hashtable persistente que generalmente almacena pares clave-valor de cadenas; suponiendo que utilicemos los métodos setProperty y getProperty para manipular la tabla en vez de los métodos put y get heredados de Hashtable.
- Persistente significa que el objeto Property, se puede escribir a través de un flujo de salida y recuperarlo a través de un flujo de entrada (ejem. Archivo).
- La clase Property extiende de Hashtable.

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

Estructuras dinámicas no lineales: ÁRBOLES

- **Los árboles** son estructuras dinámicas utilizadas para representar jerarquías, evaluar expresiones, determinar distintas posibilidades para solucionar un problema etc.
- Cada elemento del árbol recibe el nombre de **nodo**.
- Cada nodo de un nivel inferior está enlazado **únicamente** con otro nodo del nivel inmediato superior, pero puede tener desde **0** a **n** enlaces de nivel inmediatamente inferior.

Estructuras dinámicas no lineales : **ÁRBOLES**

- En el nivel superior del árbol hay un único elemento, que se llama **raíz**.
- En cualquier árbol se llama **padre** al nodo del que cuelga algún otro nodo, e **hijos** a los nodos que cuelgan del padre.
- Se denominan **descendientes** a todos los nodo que cuelgan directa o indirectamente de un nodo que llamaremos **antecesor.**
- Los nodos sin descendientes se llaman **hojas.**

Estructuras dinámicas no lineales : ÁRBOLES

- Según el número máximo de hijos que puede tener un nodo 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.
  - **Altura** de un árbol es el número máximo de **niveles** que tiene.

Estructuras dinámicas no lineales : ÁRBOLES

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

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

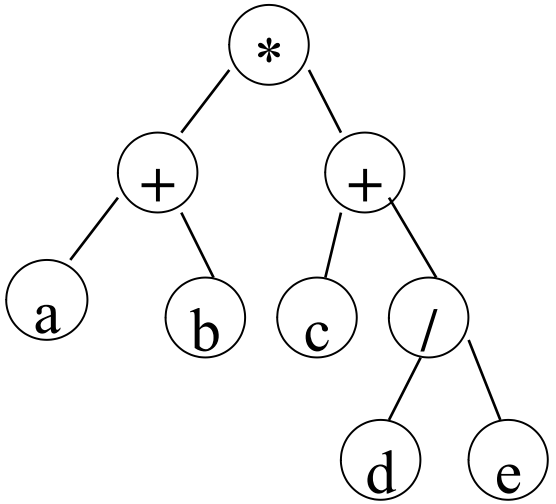
  - **Preorden.**
  - **Orden central o Inorden.**
  - **Postorden.**

Estructuras dinámicas no lineales : ÁRBOLES

- **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:** * + a b + c / d e
- **Inorden :** a + b * c + d / e
- **Postorden :** a b + c d e / + *



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***, que contiene los milisegundos transcurridos desde las 00:00:00 del 1 de enero de 1970.
- Otras clases permiten a partir de un objeto ***Date*** obtener información del año, mes, día, hora, minuto y segundo.

```
public java.util.Date();
public java.util.Date(long);
```

- 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.

Métodos de la clase Date

- 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 para un determinado objeto ***Date***.
- Otros métodos son consecuencia de las interfaces implementadas por la clase ***Date***.

Los objetos de esta clase se utilizan en combinación con las siguientes clases:

Clase 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

- Las horas se representan por enteros de 0 a 23 y los minutos y segundos por enteros entre 0 y 59.
- Los días del mes se representan por enteros entre 1 y 31.
- Los meses del año se representan mediante enteros de 0 a 11.
- 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 Date, Calendar y GregorianCalendar.

```
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class ManejadorFechas {

    public static void main(String[] args) {
        String Fecha, Hora;
        Fecha = getFechaActual();
        Hora = getHoraActual();
        System.out.println(Fecha + " " + Hora);
    }
```

La clase Date, Calendar y GregorianCalendar.

```
//Metodo usado para obtener la fecha actual
public static String getFechaActual() {
    Date ahora = new Date();
    SimpleDateFormat formateador = new SimpleDateFormat("dd-mmmm-yyyy");
    return formateador.format(ahora);
}
//Metodo usado para obtener la hora actual del sistema
//@return Retorna un <b>STRING</b> con la hora actual formato "hh:mm:ss“

public static String getHoraActual() {
    Date ahora = new Date();
    SimpleDateFormat formateador = new SimpleDateFormat("hh:mm:ss");
    return formateador.format(ahora);
}
}
```