PROGRAMACIÓN

TERCER TRIMESTRE

1ª Tutoría – Ut7. Colecciones

COLECCIONES DE DATOS EN Java

<table>
<tr><td>Simples</td><td colspan="2">Datos primitivos:
int, char, float, double, boolean, ..</td></tr>
<tr><td rowspan="3">Complejos</td><td colspan="2">Clases</td></tr>
<tr><td rowspan="2">Estructuras de almacenamiento</td><td>Estáticas: arrays</td></tr>
<tr><td>Dinámicas: colecciones</td></tr>
</table>

DIFERENCIAS entre ARRAYS y COLECCIONES

ARRAYS	COLLECTIONS
• Tamaño estático	• Tamaño Dinámico
• El tamaño se conoce por el atributo length	• Su tamaño se conoce mediante el método size()
• Puede almacenar datos tanto primitivos como complejos	• Solo puede almacenar datos de tipo complejo
• Solo puede albergar elementos de un tipo	• Puede albergar elementos de distinto tipo

COLECCIONES

* Las colecciones son clases Java que se utilizan para almacenar y manipular datos.

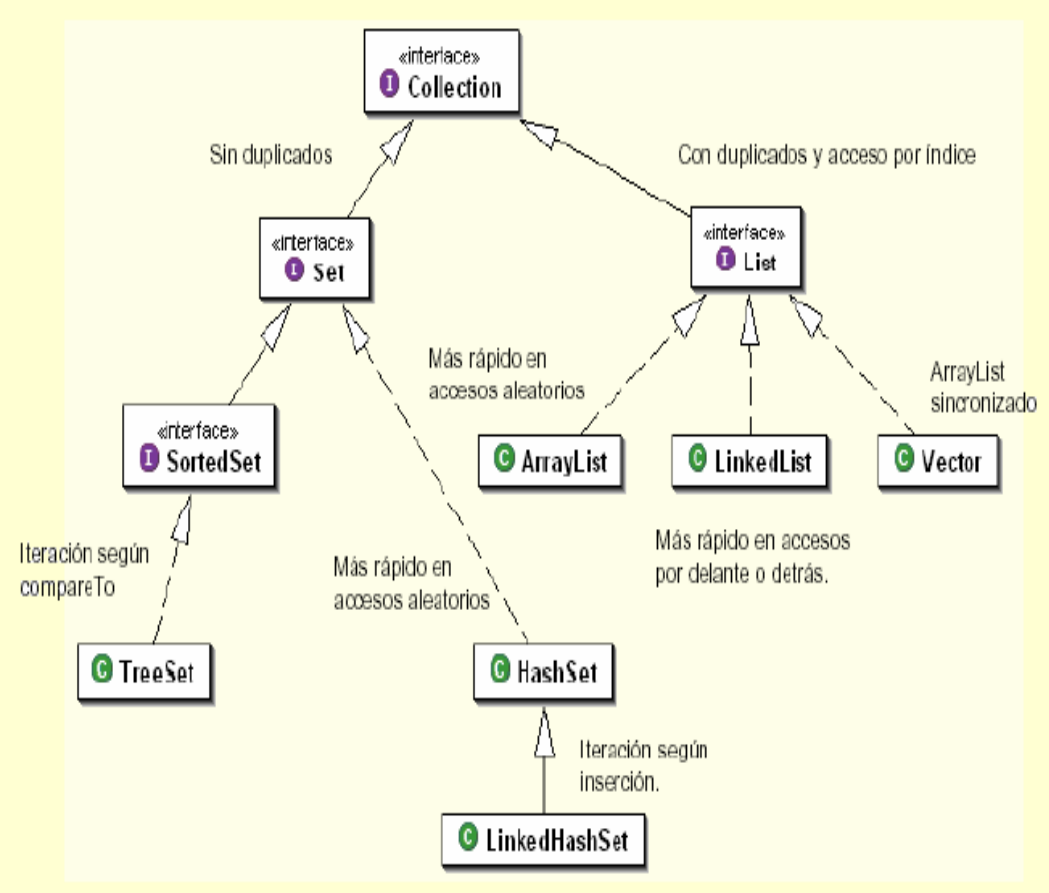
* Java dispone de una serie de clases predefinidas que simplifican el trabajo con colecciones, ya que incorporan las operaciones más utilizadas como la búsqueda, acceso por índices, ordenación, etc.

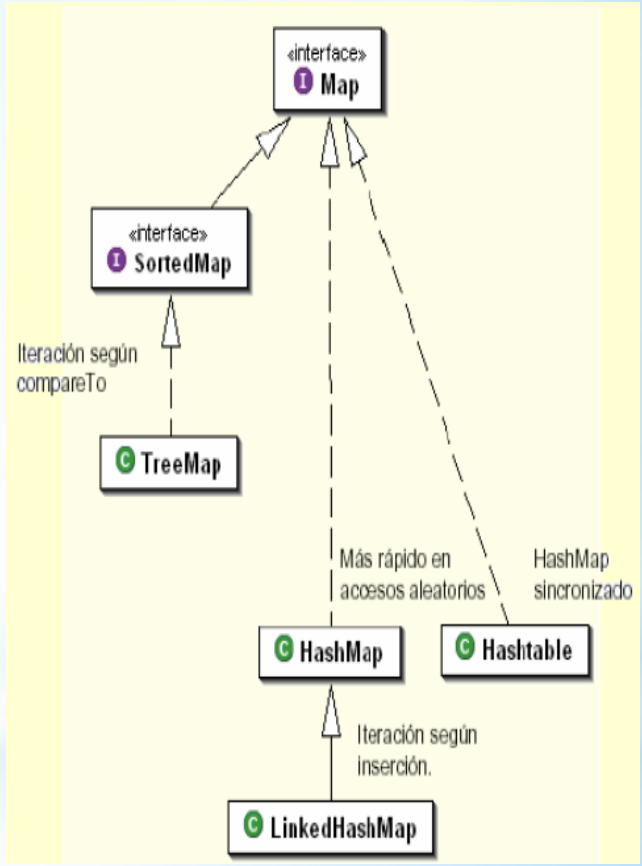
* Todas las colecciones se encuentran en **java.util.Collection**

* java.util.Collection es la raíz de las colecciones y contiene la definición de todos los **métodos genéricos** que pueden ser utilizadas por las clases predefinidas.

* Sus elementos siempre son **objetos**, por lo que no permiten almacenar tipos primitivos, en su lugar tendremos que utilizar las clases envolventes (wrapper): Integer, Double, Float…….

CLASES para implementar COLECCIONES





COLECCIONES.- Declaración

Genéricos
Permiten definir clases que pueden usar objetos de distintas clases.

TipoColección <tipoDato> varObjeto;

varObjeto = new TipoColección <tipoDato>();

Ejemplo:

```
//defino colecciones creando referencia
HashSet<String> ciudades;
ArrayList<Integer> numeros;

//creo objetos con el operador new
ciudades = new HashSet<String>();
numeros = new ArrayList<Integer>();
```

Sólo puedo usar objetos, no tipos primitivos.
Uso de envoltorios: Integer, Float, Double, ..

Métodos de la interfaz **Collection**

* **Operaciones básicas:**
  - int **size**(); // número de elementos que contiene
  - boolean **isEmpty**(); //Comprueba si está vacía
  - boolean **contains**(Object element); //si contiene ese elemento
  - boolean **add**(Object element); // añade un elemento
  - void **remove** (Object element); // elimina un elemento
  - Iterator **iterator**(); //devuelve una instancia de Iterator
* **Operaciones masivas:**
  - boolean **containsAll**(Collection c); // si contiene todos esos elementos
  - boolean **addAll** (Collection c); // añade todos esos elementos
  - boolean **removeAll** (Collection c); // elimina todos esos elementos
  - boolean **retainsAll** (Collection c); // elimina todos menos esos elementos
  - void **clear**(); //elimina todos los elementos
* **Operaciones con arrays:**
  - Object[] **toArray**(); //devuelve un array con todos los elementos
  - Object[] **toArray**(Object a[]); //Idem, el tipo será el del array enviado

```
ciudades.add("Gijón");
int n = ciudades.size();
```

El interfaz **Iterator**

* Representa un componente que permite recorrer todos los elementos de una colección.

* Todas las colecciones ofrecen una implementación de Iterator por medio del método:

  public Iterator iterator();

* Sus métodos son:
  - boolean **hasNext**(); //si tiene más elementos
  - Object **next**(); //devuelve el elemento actual y apunta al siguiente
  - void **remove**(); //elimina el actual y apunta al siguiente

```
void imprimir(){
        Iterator<T> it = coleccion.iterator();
        while (it.hasNext())
            System.out.println(it.next());
}
```

Colecciones **CON DUPLICADOS**. java.util.List

* Heredan del interfaz Collection, implementan el interfaz List.
* Incorpora nuevos métodos de tipo posicional, posibilitando el acceso a los elementos a través de un índice, por lo que se podrá:
  - **Acceder** a un elemento concreto **a través de una posición.**
  - **Insertar** un elemento **en una posición.**
* Las clases que se implementan son:

ArrayList	Funciona como un Array sin preocuparse del tamaño y ofrece un tiempo de acceso óptimo cuando este tipo de acceso es aleatorio, pero lento si se desea insertar o eliminar en medio de la lista
LinkedList	Su orden de iteración es su orden de inserción. Permite manejar lista de objetos como una cola o una pila. El acceso es rápido si se realiza al principio o al final de la lista, en caso contrario se hace lento y su modificación es poco costosa. Implementa addFirst(), addLast(),getFirst(), getLast(), removeFirst() y removeLast()

Colecciones **CON DUPLICADOS**. java.util.List

* Los métodos que añade son:
  - Object **get**(int indice); // devuelve el elemento de esa posición
  - Object **set**(int indice, Object x); //reemplaza el elemento de esa posición por el objeto x
  - void **add**(int indice, Object x); //inserta el elemento x en esa posición
  - Object **remove**(int indice); // elimina el elemento de esa posición
  - boolean **addAll**(int indice, Collection c); //inserta todos los elementos en esa posición
  - int **indexOf**(Object x); //devuelve la posición de la primera ocurrencia de ese elemento
  - int **lastIndexOf**(Object x); //devuelve la posición de la última ocurrencia de ese elemento
* Otros métodos para LinkedList:

getFirst(), **getLast**(), **removeFirst**(), **removeLast**(), **addFirst**(). **addLast**()

Colecciones **CON DUPLICADOS.** java.util.List

ArrayList:

Un arrayList, al contrario de lo que sucedía con los arrays comunes (tablas), no tiene un tamaño fijo, es decir, puede variar la cantidad de nodos que lo forman según las necesidades que vaya habiendo a lo largo de la ejecución del programa.

El acceso a los elementos de un arrayList es rápido, ya que indicando su posición, accedemos directamente a ese elemento.

El principal inconveniente de este tipo de colecciones es borrar un elemento.

Esta clase permite el almacenamiento de datos en memoria, de forma similar a los arrays convencionales, pero con la gran ventaja de que el número de elementos que puede almacenar es dinámico.

Para especificar el tipo de datos que va a contener la lista, debemos indicarlo entre los caracteres '<' y '>'.

Podemos declarar objetos de esta clase de la forma siguiente:

```
ArrayList<nombreClase> nombreLista = new ArrayList <nombreClase> ( );
```

En caso de almacenar datos de un tipo básico de Java como byte, boolean, char, int, double, float, long, short, se debe especificar el nombre de la clase envolvente: Byte, Boolean, Character, Integer, Double, Float, Long, Short.

```
ArrayList<Double> numeros= new ArrayList <Double>();
```

Ejemplo de colecciones de tipo **ArrayList**

Almacenamiento según orden de INTRODUCCIÓN, o en POSICIÓN DADA

```
import java.util.*;
public class TestArrayList {
  public static void main(String[] args) {
    ArrayList<String> ciudades;
    ciudades = new ArrayList<String> ();
    ciudades.add("Gijón");
    ciudades.add("Madrid");
    ciudades.add("Barcelona");
    ciudades.add("Madrid");
    ciudades.add(1,"Sevilla");
    ciudades.add(3,"Valencia");
    ciudades.add("Oviedo");
    System.out.println("Número de elementos de la colección "+ ciudades.size());
    Iterator it = ciudades.iterator();
    while (it.hasNext()) {
        System.out.println("Ciudad: "+it.next());
    }
    System.out.println("El 2º elemento de la lista es "+ ciudades.get(1));
  }
}
```

```
Número de elementos de la colección 7
Ciudad: Gijón
Ciudad: Sevilla
Ciudad: Madrid
Ciudad: Valencia
Ciudad: Barcelona
Ciudad: Madrid
Ciudad: Oviedo
El 2º elemento de la lista es Sevilla
```

Colecciones **CON DUPLICADOS**. java.util.List

LinkedList:

Se las llama listas doblemente enlazadas. Cada uno de los nodos tiene dos campos de enlace; uno apunta al nodo que va por detrás de dicho nodo y el otro campo de enlace apunta al nodo que va por delante del mismo. La principal ventaja de este tipo de listas es que permite borrar e insertar nodos de una manera fácil, ya que lo único que hace es cambiar las referencias de los nodos que van delante y detrás del nodo que queremos borrar o del que queremos insertar.

Con la clase LinkedList se pueden implementar las pilas, colas LIFO y colas FIFO, agregando, o eliminando elementos, al inicio o al final de la misma.

Ejemplo de colecciones de tipo **LinkedList**

```java
import java.util.*;
public class TestLinkedList {
  public static void main(String[] args) {
    LinkedList<String> ciudades;
    ciudades = new LinkedList<String>();
    ciudades.addFirst("Gijón");    //aqui lo gestiono como una PILA
    ciudades.addFirst("Madrid");
    ciudades.addFirst("Barcelona");
    ciudades.addFirst("Madrid");
    ciudades.addFirst("Sevilla");
    ciudades.addFirst("Valencia");
    ciudades.addFirst("Oviedo");
    System.out.println("Número de elementos de la colección "+ ciudades.size());
    Iterator it = ciudades.iterator();
    while (it.hasNext()){
      System.out.println("Ciudad: "+it.next());
    }
  }
}
```

```
Número de elementos de la colección 7
Ciudad: Oviedo
Ciudad: Valencia
Ciudad: Sevilla
Ciudad: Madrid
Ciudad: Barcelona
Ciudad: Madrid
Ciudad: Gijón
```

Colecciones **SIN DUPLICADOS.** java.util.Set

* El interfaz Set hereda del interfaz Collection, pero no define ningún método nuevo.
* Representa colecciones que no permiten tener **ningún elemento duplicado.**
* Para saber si un elemento está duplicado o no, hace uso del método:

  public boolean equals(Object o);
* Existen distintas implementaciones de este interfaz:

HashSet	Realiza todas las operaciones habituales (añadir, borrar, comprobar). Ofrece un acceso rápido cuando dicho acceso es aleatorio. Su orden de iteración es impredecible
LinkedHashSet	Su orden de iteración es su orden de inserción
TreeSet	Su orden de iteración depende de la implementación que los elementos hagan del interface **java.lang.Comparable**, es decir todos los elementos están ordenados en su orden natural por defecto o como se indique el método **compareTo** de la clase

Ejemplo colecciones de tipo **HashSet**

Almacenamiento ALEATORIO

```java
import java.util.*;
//El almacenamiento será aleatorio aplicando un algoritmo Hash
public class TestHashSet {
    public static void main(String[] args) {
        HashSet<String> ciudades;     //defino la colección de datos ciudades
        ciudades = new HashSet<String>();
        ciudades.add("Gijón");
        ciudades.add("Madrid");
        ciudades.add("Barcelona");
        ciudades.add("Oviedo");
        ciudades.add("Madrid");    //no lo añadiría por estar duplicado
        System.out.println("Número de elementos de la coleccion "+ ciudades.size());
        Iterator it = ciudades.iterator();
        while (it.hasNext())
            System.out.println("Ciudad: "+it.next());
        /* Otra forma de recorrer la colección con un bucle for-each
        for(String ciudad:ciudades)
            System.out.println("Ciudad: "+ciudad);
        */
    }
}
```

```
Número de elementos de la coleccion 4
Ciudad: Barcelona
Ciudad: Madrid
Ciudad: Oviedo
Ciudad: Gijón
```

Ejemplo colecciones de tipo **TreeSet**

Almacenamiento ORDENADO

```java
import java.util.*;
//almacenamiento por orden natural o según compareTo implementando Comparable
//si fuera una clase compuesta Alumno,… lo haría según el método CompareTo
public class TestTreeSet {
    public static void main(String[] args) {
        TreeSet<String> ciudades; //defino la colección de datos ciudades
        ciudades = new TreeSet<String>();
        ciudades.add("Gijón");
        ciudades.add("Madrid");
        ciudades.add("Barcelona");
        ciudades.add("Oviedo");
        ciudades.add("Madrid"); //duplicado no lo añadiría
        System.out.println("Numero de elementos de la coleccion "+ ciudades.size());
        Iterator it = ciudades.iterator();
        while (it.hasNext())
            System.out.println("Ciudad: "+it.next());
    }
}
```

```
Numero de elementos de la coleccion 4
Ciudad: Barcelona
Ciudad: Gijón
Ciudad: Madrid
Ciudad: Oviedo
```

Ejemplo colecciones de tipo **TreeSet**

Otro ejemplo con números

```java
import java.util.TreeSet;
public class TestOtroTreeSet {
  public static void main(String[] args) {
    TreeSet<Integer> numeros; //defino la colección de datos numeros
    numeros = new TreeSet<Integer>();
    numeros.add(4);
    numeros.add(8);
    numeros.add(3);
    numeros.add(1);
    System.out.println("Número de elementos de la colección "+ numeros.size());
    for(int numero: numeros)
      System.out.println("Número: "+numero);
  }
}
```

```
Número de elementos de la colección 4
Número: 1
Número: 3
Número: 4
Número: 8
```

Ejemplo colecciones de tipo **TreeSet** usando **compareTo**

Debe implementar el interfaz Comparable

```
Número de elementos de la colección 4
Número: 1
Número: 3
Número: 4
Número: 8
```

```java
public class Numero implements Comparable<Numero>{
  int n;
  public Numero(int n) {
    this.n = n;
  }
  public String toString() {
    return String.valueOf(n);
  }
  public int compareTo(Numero n1) {
    if (n == n1.n)
      return 0 ;
    else if (n>n1.n)
      return 1;
    else
      return -1;
  }
}
```

```java
import java.util.Iterator;
import java.util.TreeSet;
public class TestTreeSet2 {
  public static void main(String[] args) {
    TreeSet<Numero> numeros; //defino la  colección de datos numeros
    numeros = new TreeSet<Numero>();
    numeros.add(new Numero(4));
    numeros.add(new Numero(3));
    numeros.add(new Numero(8));
    numeros.add(new Numero(1));
    System.out.println("Nuúmero de elementos de la coleccion "+ numeros.size());
    Iterator it = numeros.iterator();
    while (it.hasNext())
      System.out.println("Número: "+it.next());
  }
}
```

Colecciones **clave/valor**. java.util.Map

* No hereda de Collection, sin embargo se utiliza para tratar colecciones de objetos.
* Representa colecciones de valores con parejas de objetos, formadas por: una **clave** y un **valor**.
* No permite claves duplicadas pero sí valores duplicados.
* Existen distintas implementaciones de este interfaz:

HashMap	Ofrece un tiempo de acceso óptimo cuando dicho acceso es aleatorio. Su orden de inserción e iteración es imprevisible
Hashtable	Es la versión sincronizada de HashMap
LinkedHashMap	Iteración según orden de inserción
TreeMap	Su inserción e Iteración dependerá de la implantación o no del método compareTo()

¿QUÉ COLECCIÓN UTILIZAR?

* **ArrayList**, **HashMap** y **HashSet** son las implementaciones primarias de List, Map y Set respectivamente.

* Se utilizarán estas implementaciones a no ser que se utilice alguna característica adicional (orden de inserción y ordenación).

* Para que los elementos de la colección se puedan ordenar, es necesario implementar la interfaz Comparable y será necesario redefinir el método compareTo.

Implementar el método **compareTo**

* Este método es el único miembro de la interfaz **Comparable**. Cuando una clase implementa esta interfaz, estaremos indicando que sus objetos van a poder ser comparados y por lo tanto ordenados.

* Implementar esta interfaz nos permite incorporar a la colección los métodos:
  - Collections.**sort**()
  - Collections.**binarySearch**()

* Devuelve un entero que será:
  - Negativo: si el objeto en cuestión es menor
  - Positivo: si el objeto en cuestión es mayor
  - 0: si ambos son iguales

Implementar el método **compareTo**

```
public class DVD implements Comparable<DVD>{
    String codigo;
    String titulo, director;
    int año;
    public int compareTo(DVD x) {
        final int MENOR = -1;
        final int MAYOR = 1;
        final int IGUAL = 0;
        if (this==x) return IGUAL;
        int esteCodigo = new Integer (this.codigo);
        int otroCodigo = new Integer(x.codigo);
        if (esteCodigo < otroCodigo) return MENOR;
        if (esteCodigo > otroCodigo) return MAYOR;
        return IGUAL;
    }
}
```

Para ordenar la lista de DVD's:

```
ArrayList<DVD> listaDVD = new ArrayList<DVD>();
...
Collections.sort(listaDVD);
```