Centro Asociado Palma de Mallorca

Introducción Práctica de Programación Java

Antonio Rivero Cuesta

Sesión

X

Colecciones

		Implementaciones				
		Tabla Hash	Array Dinámico	Árbol Balanceado	Listas Enlazadas	Tabla Hash Listas Enlazadas
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Una **colección** es una agrupación de objetos relacionados que forma una única entidad.

Java dispone de un conjunto de clases predefinidas que implementan estructuras de control muy potentes para agrupar objetos.

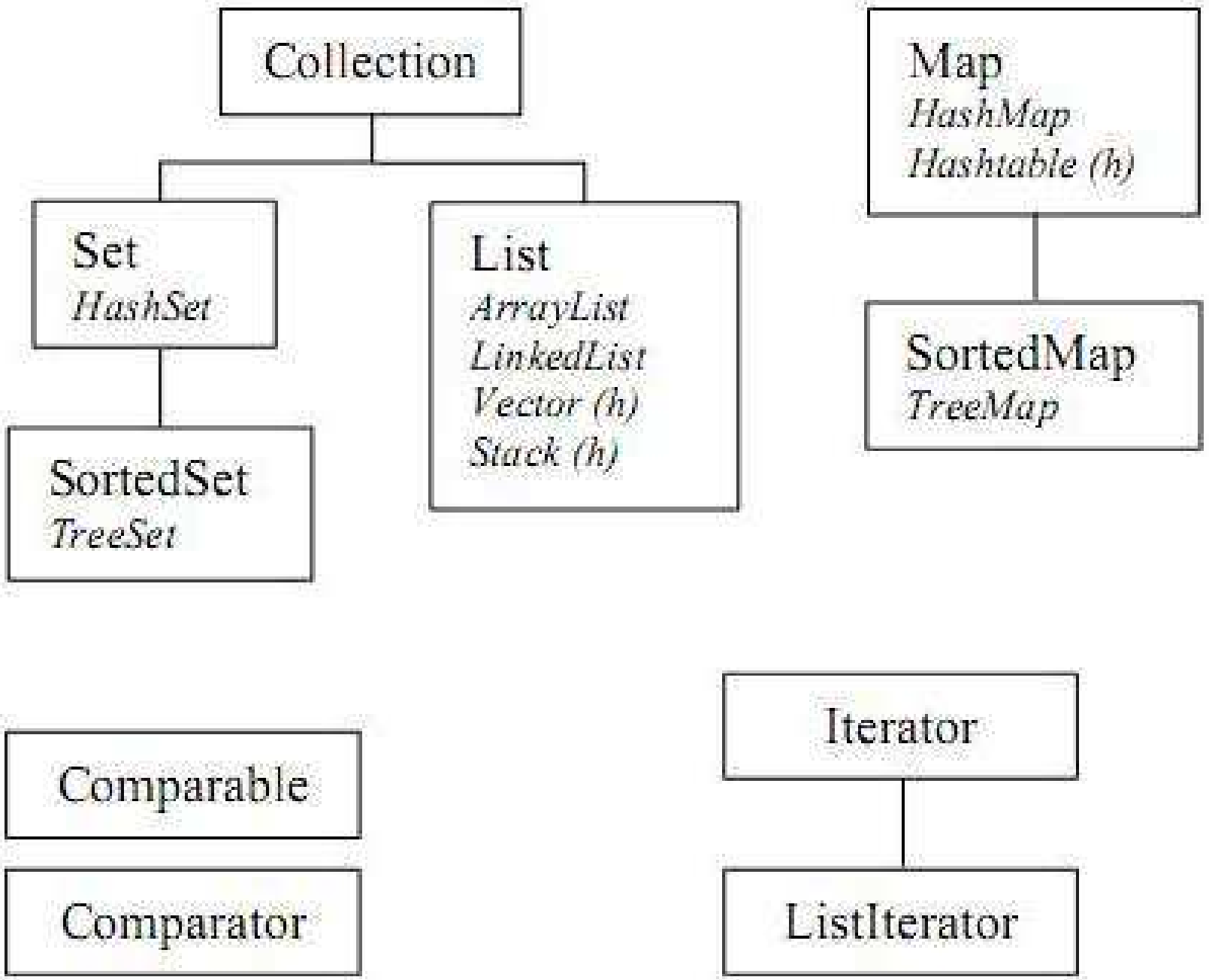
Cada clase organiza los objetos de una forma particular.

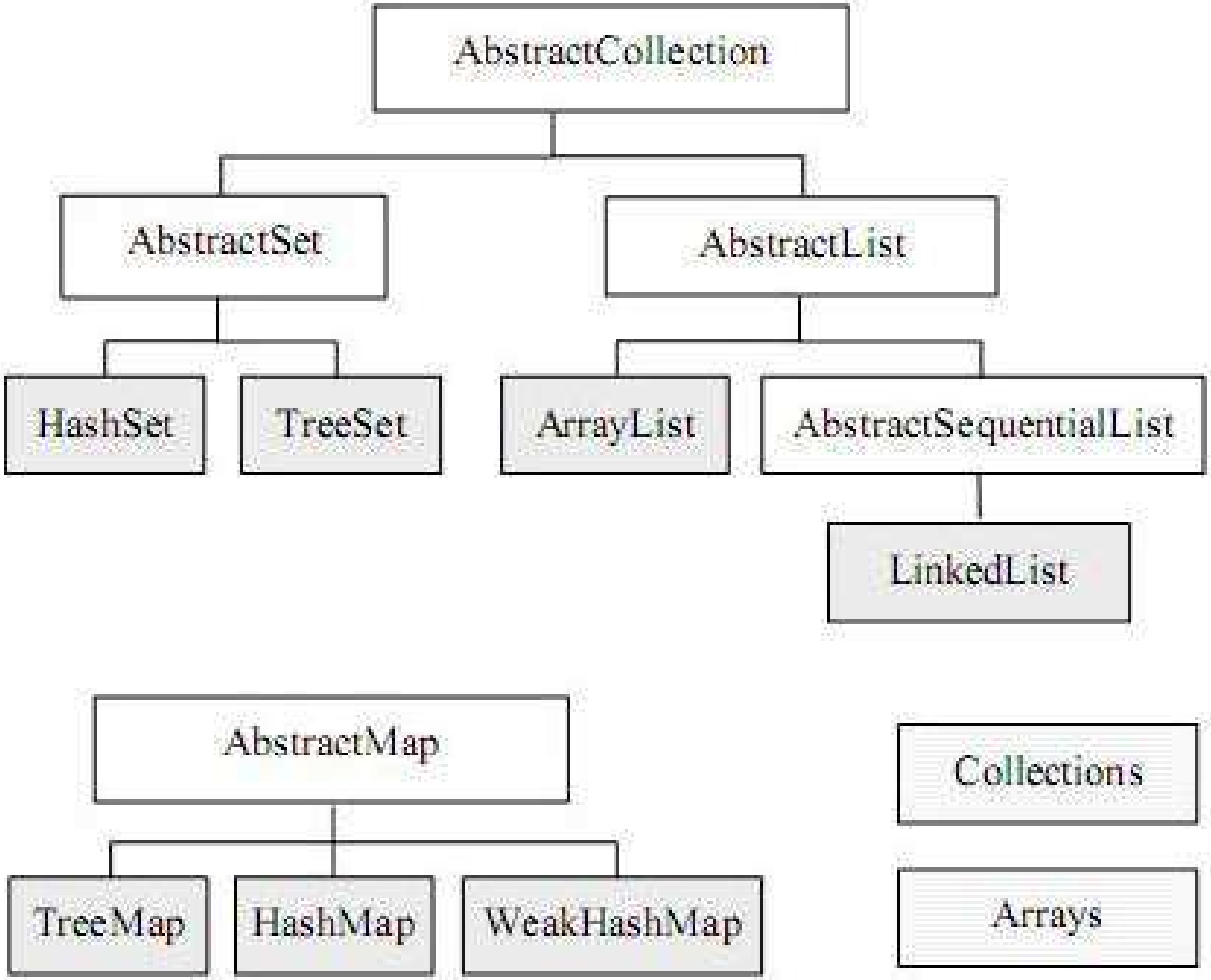
Las implementaciones más utilizadas derivan de la interfaz **`Collection`**.

Hay tres tipos de colecciones descritas por los interfaces:

- Set.
- List.
- Map.

Las dos primeras derivan de Collection.


Collection
Set
HashSet
SortedSet
TreeSet
List
ArrayList
LinkedList
Vector (h)
Stack (h)
Map
HashMap
Hashtable (h)
SortedMap
TreeMap
Comparable
Comparator
Iterator
ListIterator


AbstractCollection
AbstractSet
AbstractList
HashSet
TreeSet
ArrayList
AbstractSequentialList
LinkedList
AbstractMap
Collections
TreeMap
HashMap
WeakHashMap
Arrays

Son importantes para el desarrollo de programas profesionales.

Facilitan considerablemente el diseño y construcción de aplicaciones basadas en estructuras de datos tales como:

- Vectores.
- Listas.
- Conjuntos.
- Mapas.

Proporcionan programación genérica para muchas estructuras de datos.

Las colecciones incluyen:

- ***Clases contenedoras*** para almacenar objetos.
- ***Iteradores*** para acceder a los objetos en el interior de los contenedores
- ***Algoritmos*** para manipular los objetos, métodos de clases.

Las clases **Colección** guardan objetos de cualquier tipo.

El elemento base es **`Object`**.

Debido a la conversión automática, se podrá añadir a la colección un objeto de cualquier tipo.

`Collection` declara métodos que serán implementados por las distintas clases.

Interfaz List

Una lista es una agrupación lineal de elementos, que pueden duplicarse.

A una lista se añaden elementos por la *cabeza*, por el *final* y, en general, por cualquier punto.

También, se pueden eliminar elementos de uno en uno, o bien todos aquellos que estén en una colección.

Existen dos tipos de listas:

- Secuenciales.
- Enlazadas.

El concepto general de lista está representado por la interfaz **`List`.**

Esta interfaz es la raíz de la jerarquía y por conversión automática toda colección de tipo lista se puede tratar con una variable de tipo **`List`**.

ArrayList

Es una clase que permite almacenar datos en memoria de forma similar a los Arrays.

Los elementos que almacena lo hace de forma dinámica.

No es necesario declarar su tamaño como pasa con los Arrays.

Implementa la interfaz **`List`**.

Extiende de la clase abstracta **`AbstractList`**.

La clase **`ArrayList`** declara muchos métodos entre ellos:

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

Características importantes de la clase **`ArrayList`** son:

- Es capaz de aumentar su capacidad interna tanto como se requiera.
- Cuando se agregan más elementos, simplemente crea el espacio necesario para ellos.
- Mantiene su propia cuenta privada de la cantidad de elementos almacenados, que se obtiene mediante **`size`**.

- Mantiene el orden de los elementos que se agregan.
- El método **add** almacena cada nuevo elemento al final de la lista.
- Posteriormente se pueden recuperar en el mismo orden.

Recorrido Colecciones

Una de las situaciones más habituales en la programación es recorrer una lista de elementos.

En Java se puede realizar de varias maneras:

- Utilizando Iterator
- Utilizando un bucle for each.
- Utilizando un bucle for.
- Enumeration.

Ejemplo: ArrayList Lenguaje

Iteradores en ArrayList

Otra cosa muy importante a la hora de trabajar con los ArrayList son los ***Iteradores***.

Sirven para recorrer los ArrayList y poder trabajar con ellos.

Métodos de Iterator

`hasNext()` para comprobar que siguen quedando elementos en el iterador.

`next()` para que nos dé el siguiente elemento del iterador.

`remove()` para eliminar un elemento del iterador.

```
Iterator<String> it = pelis.iterator();
  while (it.hasNext()){
     String elemento = it.next();
     System.out.println(elemento);
  }
```

Bucle for each

Se incorpora en la versión 5 de Java.

Esta estructura nos permite recorrer una Colección o un array de elementos de una forma sencilla.

Evita el uso de Iteradores o de un bucle **`for`** normal.

```java
// Declaro un ArrayList
ArrayList<String> pelis = new ArrayList<String>();


// Estructura for each
for (String nombre: pelis)
    System.out.println(nombre);
```

El resultado es más legible.

Sin embargo en ocasiones los iteradores aportan cosas interesantes que los bucles for each no pueden abordar.

Vamos a borrar todos los objetos de una lista con un nombre concreto.

La operación parece tan sencilla como hacer lo siguiente:

```
for (String nombre:pelis){
   if (nombre.equals("Casablanca")){
   pelis.remove("Casablanca");
   }
}
```

El código no funciona y lanza una excepción.

```
Exception in thread "main" java.util.ConcurrentModificationException
   at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
   at java.util.ArrayList$Itr.next(ArrayList.java:851)
   at Cine.main(Cine.java:83)
```

El interface Iterator dispone de un método adicional que permite eliminar objetos de una lista mientras la recorremos.

El método **`remove`**.

Enumeration

La interfaz **`Enumeration`** declara métodos que recorren una colección.

Este tipo de iterador permite acceder a cada elemento de una colección.

Al ser de sólo lectura no permite modificar la colección.

Forma parte del paquete **`java.util.`**

Su declaración es la siguiente:

```
public interface Enumeration{
  boolean hasMoreElements();
  Object nextElement();
}
```

`nextElement()` devuelve el siguiente elemento.

Levanta la excepción **`NoSuchElementException`** si no hay más elementos, es decir, si ya se ha recorrido toda la colección.

La primera llamada devuelve el primer elemento.

`hasMoreElements()` devuelve true si no se ha accedido a todos los elementos de la colección.

Normalmente se diseña un bucle, controlado por este método, para acceder a cada elemento de la colección.

ListIterator

Este iterador es específico de las colecciones que implementan la interfaz List y deriva de Iterator.

Permite:

- Recorrer una lista en ambas direcciones.
- Eliminar.
- Cambiar.
- Añadir elementos de la lista.

LinkedList

Esta clase organiza los elementos de una colección como una lista doblemente enlazada.

Las operaciones de inserción y borrado en posiciones intermedias son muy eficientes.

Por el contrario, el acceso a un elemento por índice es ineficiente.

Más rápida que **`ArrayList`** añadiendo elementos al principio y eliminando elementos en general.

La debemos utilzar en lugar de **`ArrayList`** si se realizan más operaciones de inserción, en posición 0, o de eliminación que de lectura.

La diferencia de rendimiento es enorme.

Las búsquedas o las inserciones en posiciones intermedias de una lista con gran número de elementos es más rápida en el caso de **`ArrayList`**.

Esto es debido a que ésta accede a la posición de manera directa.

La clase dispone de un constructor sin argumentos que crea una lista vacía, y otro constructor que crea la lista con los elementos de otra colección.

```
public LinkedList();
public LinkedList(Collection c);
```

Implementa la interfaz **`Cloneable`**.

Las operaciones generales de las listas y métodos específicos que operan sobre el primer y último elemento son:

```
public Object getFirst()

public Object getLast()

public void addFirst(Object ob)

public void addLast(Object ob)

public Object removeFirst()

public Object removeLast()
```

Se puede usar para crear una estructura de datos **Cola** o **Pila**.

Añadir datos a la **Cola** se hace con **`addLast()`**.

Eliminar con **`removeFirst().`**

Obtener el primer elemento con **`getFirst().`**

En cuanto a ArrayList y LinkedList vemos que las inserciones en posiciones iniciales o finales son por regla general más rápidas con LinkedList.

Igualmente pasaría con las eliminaciones o borrados.

Aunque observamos que las búsquedas o las inserciones en posiciones intermedias de una lista con gran número de elementos es más rápida en el caso de ArrayList.

Esto es debido a que ésta accede a la posición de manera directa.

Para finalizar esta comparación vemos que el método **add** por defecto sin posición es más rápido que insertar en posiciones iniciales o intermedias en ambas implementaciones.

Conjuntos

La estructura de datos **`Conjunto`** se basa en el concepto matemático de conjunto: colección de elementos no duplicados.

Algebraicamente los elementos de un conjunto no tienen que mantener un orden.

Sin embargo, hay dos tipos de implementaciones, una, a partir de la interfaz **`SortedSet`**, mantiene en orden los elementos, otra sin un orden establecido.

La interfaz **`Set`** declara las operaciones generales de los conjuntos y es la raíz de la jerarquía.

Los métodos que declara son los mismos que **`Collection.`**

Aunque pone las restricciones que derivan de no tener elementos duplicados.

Todas las operaciones matemáticas de los conjuntos: *unión*, *intersección*, ..., se realizan con los métodos de las clases concretas **`HashSet`** y **`TreeSet`**.

HashSet

La clase **`HashSet`** guarda los elementos de un conjunto sin mantener un orden.

Los elementos del conjunto se guardan en una tabla hash (**`HashMap`**).

Utilizando los constructores de la clase **`HashSet`** se puede crear un conjunto vacío o un conjunto con los elementos de otra colección.

```
public HashSet();
public HashSet(Collection c);
```

TreeSet

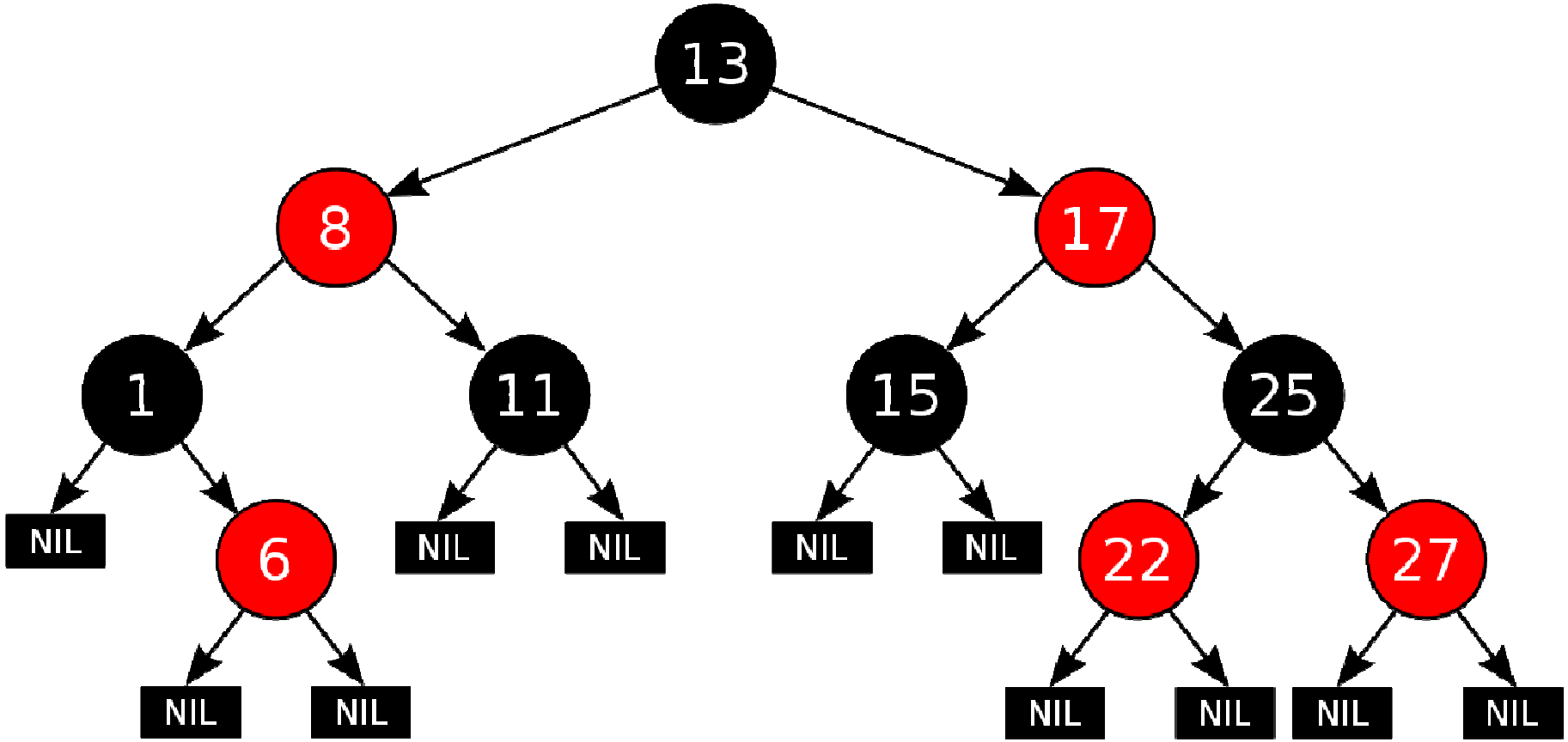
Estos conjuntos se diferencian de los **`HashSet`** por mantener en orden los elementos.

La ordenación puede ser:

- Ascendente, es decir en el orden natural determinado por la interfaz **`Comparable`**.
- O por el orden que establece la implementación de la interfaz **`Comparator`**.

Los elementos del conjunto se organizan en un *árbol roji-negro.*

Operación	**Tiempo Ejecución**
Búsqueda	O(log *n*)
Inserción	O(log *n*)
Eliminación	O(log *n*)
Iteración en orden	O(*n*)



Árbol binario de búsqueda auto-balanceable o equilibrado

El comportamiento, es decir los métodos, de esta implementación se encuentran en la interfaz **`SortedSet`** que la clase **`TreeSet`** implementa.

Mapas

Un diccionario agrupa elementos identificados mediante claves únicas.

Es una colección cuyos elementos son pares y están formados por un dato y su clave.

Identifica de manera unívoca al elemento.

Son contenedores asociativos.

También denominados *mapas*.

Las colecciones históricas definen los diccionarios con la clase abstracta **`Dictionary`** y la clase concreta **`HashTable`**.

La interfaz **`Map`** especifica los métodos comunes a los mapas de las colecciones desarrolladas en Java 2.

Dictionary

Esta clase abstracta es la interfaz para crear diccionarios en las colecciones históricas.

Procesa la colección como si fuera un *array* asociativo al que se accede por una clave.

Tanto el campo dato como la clave de cada elemento del diccionario son objetos.

Todos los métodos de la clase son abstractos.

Hashtable

La clase histórica **`Hashtable`** extiende **`Dictionary`** y se puede utilizar para agrupar datos asociativos.

Se comporta como una tabla *hash*.

Dispersa los elementos según el código que devuelve el método **`hashCode()`** del objeto clave.

Java 2 ha cambiado la declaración histórica de **`Hashtable.`**

Ahora implementa la interfaz **`Map`** y por consiguiente es compatible con las nuevas clases diccionario.

Map

Esta Interface nos permite representar una estructura de datos para almacenar pares “clave/valor”.

Un Map no puede tener claves duplicadas.

La clave funciona como un identificador único.

Las claves pueden ser de cualquier tipo de objetos.

Si añadimos un nuevo elemento clave/valor cuando la clave ya existe, se sobrescribe el valor almacenado.

Los más utilizados como clave son los objetos predefinidos de Java como:

- **`String.`**
- **`Integer.`**
- **`Double.`**

Los Map no permiten datos atómicos.

No deriva de **`Collection.`**

Declaración de un **`Map`** (un **`HashMap`**) con:

- Clave “**`Integer`**”.
- Valor “**`String`**”.

```
Map<Integer,String>nombreMap=newHashMap<Integer,String>();
```

Los principales métodos para trabajar con los Map son los siguientes:

```
nombreMap.size();
```

Devuelve el número de elementos del Map.

`nombreMap.isEmpty();`

Devuelve:

- **`true`** si no hay elementos.
- **`false`** si hay elementos.

`nombreMap.put(K clave, V valor);`

Añade un elemento al Map.

`nombreMap.get(K clave);`

Devuelve el valor de la clave que se le pasa como parámetro o "**`null`**" si la clave no existe.

`nombreMap.clear();`

Borra todos los componentes del Map.

`nombreMap.remove(K clave);`

Borra el par clave/valor de la clave que se le pasa como parámetro.

`nombreMap.containsKey(K clave);`

Devuelve true si en el map hay una clave que coincide con K.

`nombreMap.containsValue(V valor);`

Devuelve true si en el map hay un Valor que coincide con V.

`nombreMap.values();`

Devuelve una “Collection” con los valores del Map.

Iteradores en Map

Es otro elemento importante a la hora de trabajar con los Maps.

Sirven para recorrer los Map y poder trabajar con ellos.

Métodos de Iterator

`hasNext()` para comprobar que siguen quedando elementos en el iterador.

`next()` para que nos dé el siguiente elemento del iterador.

`remove()` para eliminar un elemento del iterador.

```
Iterator<String> it = pelis.iterator();
  while (it.hasNext()){
     String elemento = it.next();
     System.out.println(elemento);
  }
```

En realidad se puede prescindir de los iteradores para trabajar con los Map.

La gran ventaja de los Map frente a los ArrayList, es que estos tienen una clave asociada al objeto y se les puede buscar por la clave

Aunque se debe saber utilizar los iteradores con los Map.

Vamos a ver las dos formas de recorrer un Hashmap:

- **`keySet()`**
- **`entrySet()`**

keySet()

Lo que se obtiene son las claves.

Mediante un iterador se recorre la lista de claves.

De esta forma si queremos saber también el valor de cada elemento tenemos que usar el método get(clave).

entrySet()

Se obtienen los elementos enteros y al igual que en el caso anterior con un iterador se recorre el **`hashMap`**.

De esta forma hay que crear una variable de tipo **`Map.Entry`** para almacenar el elemento y con los métodos **`getKey()`** y **`getValue()`** de **`Map.Entry`** se obtienen los valores.

Con esta segunda forma es necesario usar una variable **`Map.Entry`** y realizar el **`import`** pertinente.

El resultado es el mismo y el número de líneas de código es igual.

Esta segunda forma es más eficiente puesto que ya tenemos ambos valores y no hay que realizar la búsqueda adicional.

Java tiene implementadas varias clases Map.

- HashMap.
- TreeMap.
- LinkedHashMap.

HashMap

Los elementos que inserta en el Map no tendrán un orden específico.

No aceptan claves duplicadas.

Los mapas tipo **`HashMap`** organizan los elementos en una *tabla hash*, proporciona una eficiencia constante a las operaciones de búsqueda e inserción.

Esta clase permite que haya claves y valores **`null`**, a diferencia de **`Hashtable`** que levanta una excepción si la clave es **`null`**.

En general, el comportamiento de **`HashMap`** es similar a **`Hashtable`**.

No es sincronizado, por lo que tiene una mejor performance.

TreeMap

El Map lo ordena de forma natural.

Si la clave son valores enteros, entonces los ordena de menor a mayor.

Mantiene en orden sus elementos para lo que utiliza la estructura *árbol roji-negro*.

Este orden está determinado por el campo clave.

Puede ser en orden natural establecido por el método **`compareTo()`** del objeto clave.

O bien por el comparador con el que se crea **`TreeMap`**.

Es requisito imprescindible que la clave implemente a **`Comparable`**, o bien **`Comparator`**.

LinkedHashMap

Inserta en el Map los elementos en el orden en el que se van insertando.

No tiene una ordenación de los elementos como tal, por lo que esta clase realiza las búsquedas de los elementos de forma más lenta que las demás clases.

Listas y conjuntos

Estructura	get	add	remove	contains
ArrayList	O(1)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(1)	O(n)
HashSet	O(1)	O(1)	O(1)	O(1)
LinkedHashSet	O(1)	O(1)	O(1)	O(1)
TreeSet	O(log n)	O(log n)	O(log n)	O(log n)

Mapas:

Estructura	get	put	remove	containsKey
HashMap	O(1)	O(1)	O(1)	O(1)
LinkedHashMap	O(1)	O(1)	O(1)	O(1)
TreeMap	O(log n)	O(log n)	O(log n)	O(log n)