ADSI

Framework de Colecciones Java - POO

Instructor: Gustavo Adolfo Rodríguez Q.

garodriguez335@misena.edu.co

ADSI

FRAMEWORK DE COLECCIONES

Como se ha presentado en elecciones anteriores, Java brinda la posibilidad de trabajar con arreglos como estructuras para la agrupación de un conjunto de datos tanto de tipos primitivos como de tipos complejos u objetos.

El trabajo en la vida real con arreglos dentro del lenguaje Java está fuertemente ligado a dispositivos o ambientes de ejecución con bajas capacidades tanto de procesamiento como de memoria. No significa esto que en ambientes potentes como computadores de escritorio o servidores no se pueda utilizar arreglos sino que existen otros elementos más evolucionados y más fáciles de utilizar que pueden hacer tanto o más cosas que los arreglos genéricos.

Estos elementos que permiten contener un conjunto de datos tanto de tipos primitivos como de tipos complejos están agrupados dentro de lo que se conoce como el ***Generic Collection Framework (GCF)*** de Java.

Para iniciar con las colecciones vamos a estudiar las clases **Vector** y **Hashtable** y posteriormente la interface **Enumeration.** Estas clases hacen parte de java desde su primera versión y están disponibles desde dispositivos móviles hasta sistemas servidores. Posteriormente, estudiaremos con más detalle las clases e interfaces del *Generic Collection Framework*

1. LA CLASE VECTOR

Podríamos decir que un vector es una estructura muy similar a un arreglo en Java. Entre las similitudes tenemos:

- Es una estructura que permite almacenar un conjunto de datos.
- Los datos son accedidos mediante un índice que empieza en ***0*** y termina en ***longitud -1***
- Se puede almacenar tanto tipos primitivos como tipos objetos.

Sin embargo, existen varias diferencias fundamentales entre las cuales encontramos:

- Un vector tiene un tamaño inicial que puede ser variado dinámicamente en tiempo de ejecución, mientras que el tamaño de un arreglo permanece invariante.
- No es necesario especificar el tamaño inicial del Vector.
- Un vector puede contener elementos de diferentes tipos al mismo tiempo, un arreglo sólo admite valores del mismo tipo.
- Para acceder a los elementos del Vector se utiliza el método **elementAt(int index),** mientras que un arreglo se accede con [int index]
- Un Vector puede ser clonado, obteniendo otro Vector con el mismo contenido.

La siguiente lista de métodos que posee la clase **Vector** nos da una muestra de su flexibilidad y potencia con respecto a los arreglos.

MÉTODO	DESCRIPCIÓN
void **addElement**(Object element)	Añade un elemento al final del vector
boolean **removeElement(**Object element**)**	Elimina del vector el elemento que concuerda con el objeto pasado como argumento
void **removeAllElements()**	Elimina todos los objetos del vector.
Object **clone()**	Retorna un nuevo vector con el mismo contenido que el original
void **trimToSize()**	Ajusta la capacidad del vector al tamaño actual del vector
int **capacity()**	Retorna la capacidad actual del vector
int **size()**	Retorna el tamaño actual del vector
boolean **isEmpty()**	Retorna un boolean que indica si el vector está o no vacío.
Enumeration **elements()**	Retorna un objeto Enumeration para realizar recorrido de los elementos del vector.

Por ejemplo, el siguiente fragmento de código crea un objeto de tipo Vector llamado cadenas.

```
Vector cadenas;
cadenas = new Vector();

cadenas.add("Hola");
cadenas.add("Para");
cadenas.add("Todos");
```

Nótese que a diferencia de lo que haríamos en un arreglo. En el Vector no especificamos su tamaño pero este cambia dinámicamente a medida que agreguemos elementos.

2. LA CLASE HASHTABLE

Un objeto de la clase ***Hashtable*** representa una tabla que relaciona un conjunto de **claves** con un conjunto de **valores.** Al igual que la clase **Vector** la clase **Hashtable** puede ser utilizada como una alternativa a los arreglos en escenarios particulares más específicos y especializados.

El conjunto de claves utilizadas para el **Hashtable** puede ser cualquier objeto que sea diferente de ***null.***

Para el conjunto de claves generalmente se utilizan objetos de la clase String. Sin embargo, puede utilizarse cualquier objeto diferente de **null**. La única condición es que las claves no se repitan, o dicho de otra forma, Las claves en un **Hashtable** deben ser únicas.

A continuación se muestra una representación gráfica de la estructura interna de un **Hashtable**

KEY	VALUE
Object key1	Objeto asociado a la clave **key1**
Object key2	Objeto asociado a la clave **key2**
Object key3	Objeto asociado a la clave **key3**

El comportamiento de **Hashtable** es similar al de **Vector** en cuanto que ambos ajustan automáticamente su capacidad a medida que se necesita. **Hashtable** utiliza un parámetro llamado *loadFactor* para realizar la reserva de memoria en las operaciones de redimensionamiento.

La siguiente lista de métodos que posee la clase **Hashtable** nos da una muestra de su flexibilidad y potencia.

MÉTODO	DESCRIPCIÓN
Object **get**(Object element)	Devuelve el objeto que corresponde a la clave pasada como argumento.
Object **put(**Object key, Object value**)**	Agrega una nueva pareja clave-valor dentro del Hashtable.
Object **remove(**Object key**)**	Elimina la pareja clave-valor que corresponde con la clave suministrada
boolean **contains(**Object value**)**	Retorna un boolean indicando la existencia del valor indicado dentro del Hashtable.
boolean **containsKey(**Object key**)**	Retorna un boolean indicando la existencia de la clave especificada dentro del Hashtable
Object **clone()**	Retorna un nuevo vector con el mismo contenido que el original
void **rehash()**	Amplía la capacidad del Hashtable utilizando el loadFactor.
int **capacity()**	Retorna la capacidad actual del vector
int **size()**	Retorna el tamaño actual del vector
boolean **isEmpty()**	Retorna un boolean que indica si el vector está o no vacío.
Enumeration **keys()**	Retorna un objeto Enumeration para realizar recorrido de los objetos que hacen parte de las claves
Enumeration **elements()**	Retorna un objeto Enumeration para realizar recorrido de los objetos que hacen parte de los valores

A continuación se muestra un fragmento de código donde se ilustra la utilización de un **Hashtable.**

```
//declaración del objeto de tipo Hashtable
Hashtable tablaNumeros = new Hashtable();

//declaración de las claves que se utilizarán
String clave1 = "Numero1";
String clave2 = "Numero2";
String clave3 = "Numero3";

//Agregación de elementos al Hashtable
tablaNumeros.put(clave1, new Integer(1));
tablaNumeros.put(clave2, new Integer(250));
tablaNumeros.put(clave3, new Integer(345));
```

3. LA INTERFAZ ENUMERATION

La interface Enumeration define métodos útiles para recorrer una colección de objetos. Pueden existir diferentes clase que implementen esta interfaz y cada una tendrá en esencia un comportamiento similar.

La inteface Enumeration declara dos métodos:

public boolean hasMoreElements(): Retorna un valor boolean indicando si existen más elementos para recorrer o si se ha llegado al final de la colección.

public Object nextElement(): Retorna el siguiente objeto dentro de la colección. Este método lanzará una excepción ***NoSuchElementException*** si es invocado y no existen más elementos para retornar. Cada vez que se haga llamado a este método, automáticamente se posiciona en el siguiente elemento para retornar en una próxima oportunidad hasta alcanzar el final.

Como puede observarse, los métodos de definidos en Enumeration, nos permitirán realizar el recorrido sobre la colección de elementos.

Tanto la clase Vector como la clase Hashtable contiene métodos que nos permiten obtener una Enumeration tanto de valores como de claves.

4. RECORRIDO DE ELEMENTOS EN UN VECTOR

Cuando estamos utilizando un objeto de la clase Vector podemos realizar el recorrido mediante un ciclo *for* o mediante la utilización de la interfaz **Enumeration**. A continuación se muestran un fragmento de código donde se realiza el recorrido utilizando las dos formas:

```
//declaración del vector
Vector cadenas;

//construcción del vector
cadenas = new Vector();

//agregamos elementos al vector
cadenas.add("Hola");
cadenas.add("Para");
cadenas.add("Todos");

//recorrido utilizando un for
for (int i = 0; i < cadenas.size(); i++) {
    String cad = (String) cadenas.elementAt(i);
    System.out.println("cadena en la posicion " + i + " "+ cad);
}

//recorrido utilizando Enumeration
Enumeration enumCadenas = cadenas.elements();
while (enumCadenas.hasMoreElements()) {
    String cad = (String)enumCadenas.nextElement();
    System.out.println("Cadena dentro del vector " + cad);
}
```

5. RECORRIDO DE ELEMENTOS EN UN HASHTABLE

De forma similar a como se hace el recorrido de elementos en un **Vector**, también podemos hacer un recorrido por la claves o valores de un **Hashtable**. Como se muestra en el siguiente fragmento de código:

```
//declaración del objeto de tipo Hashtable
Hashtable tablaSillas = new Hashtable();

//declaración de las claves que se utilizarán
String clave1 = "1";
String clave2 = "2";
String clave3 = "3";
String clave4 = "4";

//Agregación de elementos al Hashtable
tablaSillas.put(clave1, new String("libre"));
tablaSillas.put(clave2, new String("libre"));
tablaSillas.put(clave3, new String("libre"));
tablaSillas.put(clave4, new String("ocupado"));

//Recorrido de las claves del Hashtable
Enumeration enumClaves = tablaSillas.keys();
while (enumClaves.hasMoreElements()) {
    String clave = (String)enumClaves.nextElement();
    System.out.println("Clave encontrada " + clave);
}

//Recorrido de los valores del Hashtable
Enumeration enumValores = tablaSillas.elements();
while (enumValores.hasMoreElements()) {
    String valor = (String)enumValores.nextElement();
    System.out.println("Valor econtrado " + valor);

}
```

Si se desea realizar en recorrido sobre claves y valores al mismo tiempo, por ejemplo para saber cuál valore corresponde con cada clave, podemos utilizar el código siguiente:

```java
//declaración del objeto de tipo Hashtable
Hashtable tablaProductos = new Hashtable();

//declaración de las claves que se utilizarán
String codigo1 = "1010";
String codigo2 = "1011";
String codigo3 = "1012";
String codigo4 = "1013";

//Agregación de elementos al Hashtable
tablaProductos.put(codigo1, new String("Crema"));
tablaProductos.put(codigo2, new String("Locion"));
tablaProductos.put(codigo3, new String("Jabon"));
tablaProductos.put(codigo4, new String("Cepillo"));

//Recorrido de las claves del Hashtable
//Luego obtenemos el valor a parti de la clave
Enumeration enumClaves = tablaProductos.keys();
while (enumClaves.hasMoreElements()) {
    //Obtener la clave en la tabla
    String clave = (String)enumClaves.nextElement();
    //Obtener el valor correspondiente a esa clave
    String valor = (String)tablaProductos.get(clave);

    System.out.println("La clave "+clave+" tiene el valor " + valor);
}
```

CLASES DEL FRAMEWORK GENERICO DE COLECCIONES

6. JERARQUÍA DE CLASES DEL GCF

El Framework Genérico de Colecciones (GCF) se encuentra definido mediante clases e interfaces de programación como se puede observar en el siguiente diagrama de clases **UML**.

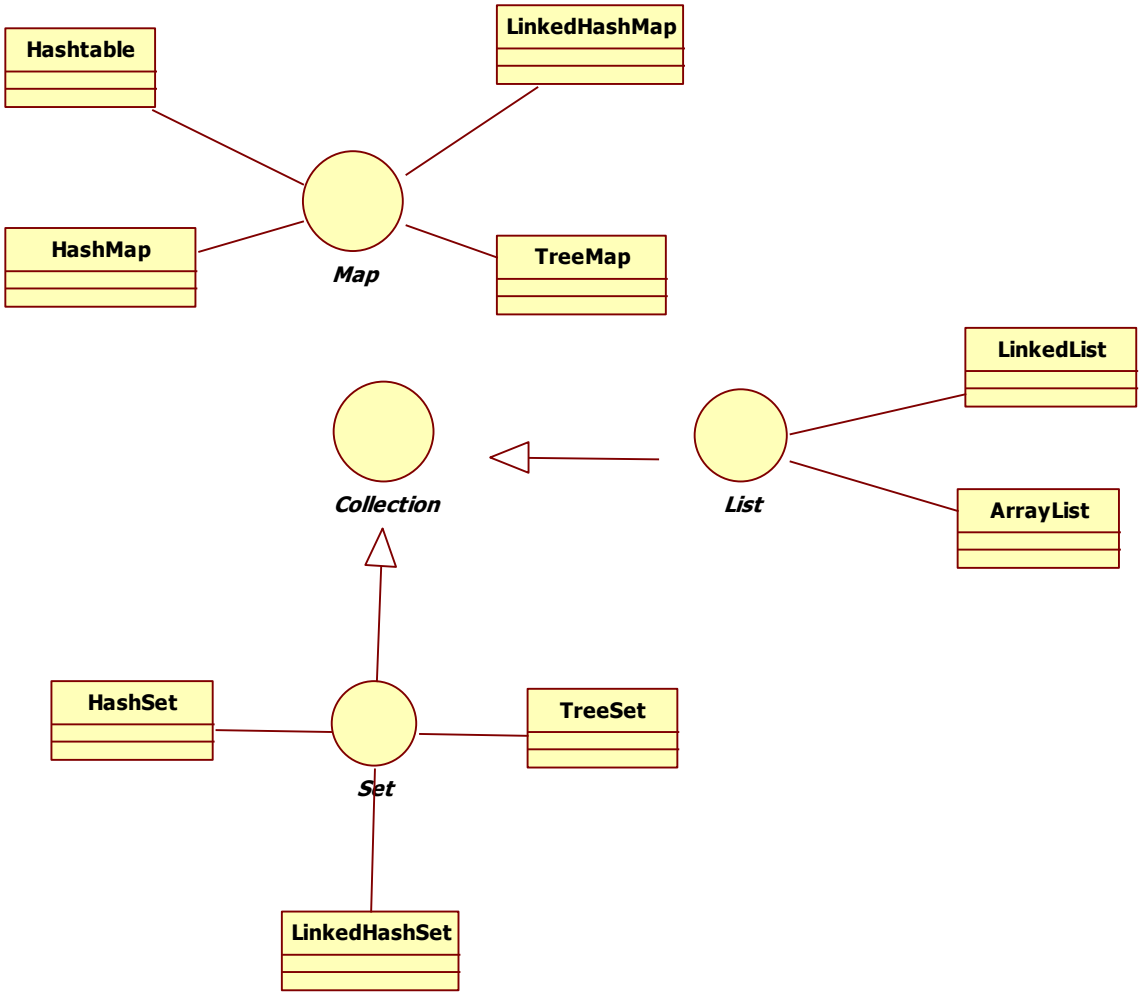

Figura 1 Principales clases del Framework Genérico de Colecciones

Este diagrama muestra sólo las principales interfaces y las implementaciones concretas más comúnmente utilizadas.

A continuación se explica con mayor detalle cada una de las interfaces mostradas en el diagrama de clases anterior.

6.1.INTERFACE COLLECTION

Es la raíz de la jerarquía de interfaces, representa a un grupo de objetos que se conocen como *"sus elementos"*. Algunas colecciones permiten almacenar objetos duplicados otras no, algunas colecciones permiten almacenar los elementos de forma naturalmente ordenada otras no.

6.2.INTERFACE LIST

Esta interface que hereda de la interface ***Collection*** representa una colección ordenada de objetos (también conocida como *secuencia*). Esta interface define métodos que permiten un control preciso acerca de dónde se inserta cada elemento.

Los elementos pueden ser accedidos utilizando su índice de localización como en los arreglos, e incluso buscando el elemento concreto en la lista. Pero a diferencia de los arreglos, el tamaño de una lista crece o se estrecha automáticamente a medida que se insertan o eliminan objetos.

A diferencia de la interface ***Set***, la interface ***List*** permite la duplicación de objetos dentro de la colección.

Las listas, de igual forma que los arreglos, basan su indización iniciando en 0, es decir, una lista que contenga 10 elementos, tendrá índices que van desde 0 hasta 9.

6.3.INTERFACE SET

Esta interface que hereda de la interface ***Collection*** representa una colección de objetos no duplicados

Algunas implementaciones de la interface Set tienen restricciones con respecto al tipo de elementos que puede contener, por ejemplo, algunas prohíben la inclusión de elementos ***null***.

A diferencia de la interface ***List***, ***Set*** no basa su indización en números, de hecho, en ***Set*** no existe indización. Para poder recuperar y recorrer los elementos de un ***Set*** se debe utilizar el ***Iterator*** asociado a él.

Iterator tiene básicamente las mismas funcionalidades que el ***Enumeration*** descrito para la clase ***Vector***.

6.4.INTERFACE MAP

Esta interface hace parte del Framework Genérico de Colecciones pero a diferencia de las interfaces ***List*** y ***Set***, la interface ***Map*** no hereda de ***Collection*** dado que no representa a una colección de objetos sino una estructura de datos que permite almacenar objetos en forma de pares clave – valor.

Podrá notar que esta descripción realizada de la interface ***Map*** es muy similar a la descripción dada para la clase ***Hashtable*** y esto se debe a que en realidad la clase ***Hashmap*** es una implementación concreta de la interface ***Map***

7. IMPLEMENTACIONES CONCRETAS DEL GCF

Como se puede notar del diagrama de clases resumido del Framework Genérico de Colecciones, existen varias implementaciones concretas para cada una de las siguientes interfaces ***List***, ***Set*** y ***Map***. Sin embargo, sólo discutiremos aquí aquellas clases más comúnmente utilizadas en el desarrollo de aplicaciones con Java.

7.1.LA CLASE ARRAYLIST

Como se puede observar del diagrama de clases resumido del Framework Genérico de Colecciones, la clase ***ArrayList*** es una implementación concreta de la interface ***List***.

En términos prácticos un ***ArrayList*** presta al programador funcionalidades muy similares a la clase Vector. Sin embargo, la utilización de uno o de otro depende en gran medida del tipo de programa que se desea crear.

Existen algunos factores que determinan cuándo utilizar Vector y cuándo utilizar ***ArrayList*** y estos tiene que ver generalmente con el nivel de carga y descarga de objetos, la velocidad de acceso a los datos y la sincronización en aplicaciones con múltiples hilos. (Thread safe operations)

La siguiente tabla muestra un resumen de los métodos más relevantes de la clase ***ArayList***

MÉTODO	DESCRIPCIÓN
boolean **add**(E e)	Agrega el elemento pasado como argumento a la lista
boolean **addAll(**Collection c**)**	Agrega la colección pasada como argumento a la lista
void **clear()**	Remueve todos los elementos de la lista
E **get**(int index)	Retorna el elemento que ocupa el índice pasado como argumento
boolean **contains(**Object o**)**	Retorna un booleano indicando si el elemento que se pasa como argumento está contenido dentro de la lista
boolean **remove(**Object o**)**	Remueve de la el objeto pasado como argumento
int **size()**	Retorna un entero con la cantidad de elementos que contiene la lista
Object[] **toArray()**	Retorna un arreglo de objetos con los elementos que contiene la lista

Cuando utilizamos un ***ArrayList*** se hace necesario especificar el tipo de datos que va a almacenar mediante la utilización de <>. Los siguientes son ejemplos de la utilización de ***ArrayList*** con diferentes tipos de elementos

```java
//Una lista de String
ArrayList<String> listaString = new ArrayList<String>();

//Una lista de Integers
ArrayList<Integer> listaInteger = new ArrayList<Integer>();

//Una lista de objetos de la clase Usuario
ArrayList<Usuario> listaUsuarios = new ArrayList<Usuario>();
```

A continuación se presenta un fragmento de código donde se ejemplifica la utilización de la clase ***ArrayList***

```java
//Agregar elementos a la lista
listaString.add("Elemento uno");
listaString.add("Elemento dos");
//remover un elemento de la lista
listaString.remove("Elemeto uno");
//remover todos los elementos de la lista
listaString.clear();

//Agregar elementos a la lista
listaInteger.add(new Integer(5));
listaInteger.add(new Integer(15));
//verificar si la lista tiene elementos
if(listaInteger.isEmpty()){
    System.out.println("la lista de Integer está vacia");
}
//obtener el manaño actual de la lista
int tamanio = listaInteger.size();

//Agregar elementos de la lista
listaUsuarios.add(new Usuario("123", "123"));
//Obtener un elemento particular de la lista utilizan el indice
Usuario user = listaUsuarios.get(0);
```

A continuación se muestra un fragmento de código donde se ejemplifica cómo recorrer los elementos de ***ArrayList***

```
//recorrer la lista listaUsuarios
for (Usuario usuario : listaUsuarios) {
    System.out.println("USuario es " + usuario.toString());
}

//recorrer la lista listaString
for (Iterator<String> it = listaString.iterator(); it.hasNext();) {
    String s = it.next();
    System.out.println("Elemento es : " + s);
}

//recorrer la lista listaInteger
for (Integer integ : listaInteger) {
    System.out.println("valor es " + integ.toString());
}
```

7.2.LA CLASE HASHSET

Como se puede observar del diagrama de clases resumido del Framework Genérico de Colecciones, la clase ***HashSet*** es una implementación concreta de la interface Set.

En su implementación interna contiene un ***HashMap*** lo cual hace que no haya garantía sobre la conservación del orden en que se recuperan los elementos del conjunto cuando vamos a realiza una iteración o recorrido.

Una de las principales características de ***HashSet*** es que el acceso a los elementos del conjunto no es sincronizado, lo cual se traduce un mayor rendimiento con respecto a otras implementaciones sincronizadas de la interface ***Set***.

Esta clase presenta un rendimiento lineal en el tiempo para realizar las operaciones básicas (agregar, remover, consultar), es decir, entre más elementos se tenga en el conjunto mayor será el tiempo necesario para ejecutar las operaciones mencionadas.

Como se había mencionado anteriormente respecto a la interface ***Set***, no existe una forma de recuperar los elementos utilizando un índice como en ***ArrayList*** o ***Vector***. Debemos utilizar el ***Iterator*** para poder recuperar los elementos.

La siguiente tabla muestra un resumen de los métodos más relevantes de la clase ***HashSet***

MÉTODO	DESCRIPCIÓN
boolean **add**(E e)	Agrega el elemento pasado como argumento a la lista
void **clear()**	Remueve todos los elementos de la lista
boolean **contains(**Object o**)**	Retorna un booleano indicando si el elemento que se pasa como argumento está contenido dentro de la lista
boolean **remove(**Object o**)**	Remueve de la el objeto pasado como argumento
int **size()**	Retorna un entero con la cantidad de elementos que contiene la lista
Object[] **toArray()**	Retorna un arreglo de objetos con los elementos que contiene la lista
boolean **isEmpty()**	Retorna un boolean indicando si el HashSet está vacío.

Cuando utilizamos un ***HashSet***, de forma similar a ***ArrayList***, se hace necesario especificar el tipo de datos que va a almacenar mediante la utilización de <>. Los siguientes son ejemplos de la utilización de ***HashSet*** con diferentes tipos de elementos

```
//Un conjunto de cadenas
HashSet<String> conjuntoString = new HashSet<String>();

//Un conjunto de instancias de la clase  Double
HashSet<Double> conjuntoDouble = new HashSet<Double>();

//Un conjunto de instancias de la clase Rol
HashSet<Rol> conjuntoRoles = new HashSet<Rol>();
```

A continuación se presenta un fragmento de código donde se ejemplifica la utilización de la clase ***HashSet***

```
//Agregamos elementos al conjunto de cadenas
conjuntoString.add("cadena1");
conjuntoString.add("cadena2");
//remover elementos del conjunto de de cadenas
conjuntoString.clear();
//verificar si el conjunto esta vacio
if(conjuntoString.isEmpty()){
    System.out.println("conjunto vacio");
}

//Agregamos elementos al conjunto de Double
conjuntoDouble.add(new Double(4.5));
conjuntoDouble.add(Math.PI);
if(conjuntoDouble.contains(Math.PI)){
    System.out.println("El número PI está en el conjunto");
}

//Agregamos elementos al conjunto de Rol
conjuntoRoles.add(new Rol(1, "Administrador", "Ninguna"));
conjuntoRoles.add(new Rol(2, "Usuario", "Ninguna"));
int size = conjuntoRoles.size();
System.out.println("El conjunto de roles tiene " + size + " roles");
```

A continuación se muestra un fragmento de código donde se ejemplifica cómo recorrer los elementos de la clase ***HashSet***.

```java
//recorrer el conjunto de cadenas
for (Iterator<String> it = conjuntoString.iterator(); it.hasNext();) {
    String s = it.next();
    System.out.println("Cadena del conjunto " + s);
}

//recorrer el conjunto de double
for (Iterator<Double> it = conjuntoDouble.iterator(); it.hasNext();) {
    Double d = it.next();
    System.out.println("Double del conjunto " + d);
}

//recorrer el conjunto de roles
for (Rol rol : conjuntoRoles) {
    System.out.println("Rol del conjunto " + rol.toString());
}
```

7.3.LA CLASE HASHMAP

Como se puede observar del diagrama de clases resumido del Framework Genérico de Colecciones, la clase ***HashMap*** es una implementación concreta de la interface ***Set***.

La clase ***HashMap*** es equivalente en comportamiento a la clase ***Hashtable***, la diferencia principal radica en que a diferencia de ***Hashtable***, la clase ***HashMap*** permite agregar valores ***null*** tanto es sus claves como en sus valores. La segunda diferencia es que en la clase ***Hashtable*** los métodos de acceso a los elementos son sincronizados en aplicaciones con múltiples hilos. (Thread safe operations) mientras que ***HashMap*** no son sincronizados.

Los métodos y forma de utilización en el código son totalmente iguales tanto en ***HashMap*** como en ***Hashtable***.