



# Clases genéricas y colecciones

# Índice



1   Tipos genéricos	3
2   Colecciones, tipos y operaciones	6
3   Listas	7
4   Conjuntos (Sets)	9
5   Diccionarios y Mapas (Maps)	10
6   Colas	12
7   Iterable e Iterator	15
8   Comparable	17
9   Comparator	18

# 1. Tipos genéricos

Los tipos genéricos se introducen en la versión 5 del JDK, permiten escribir un código que resulta más seguro y más fácil de leer, que el código alternativo lleno de referencias a `Object` y moldeos (conversiones de tipo) a subclases de `Object`. Los tipos genéricos son especialmente útiles con las clases colección que se estudian en este capítulo.

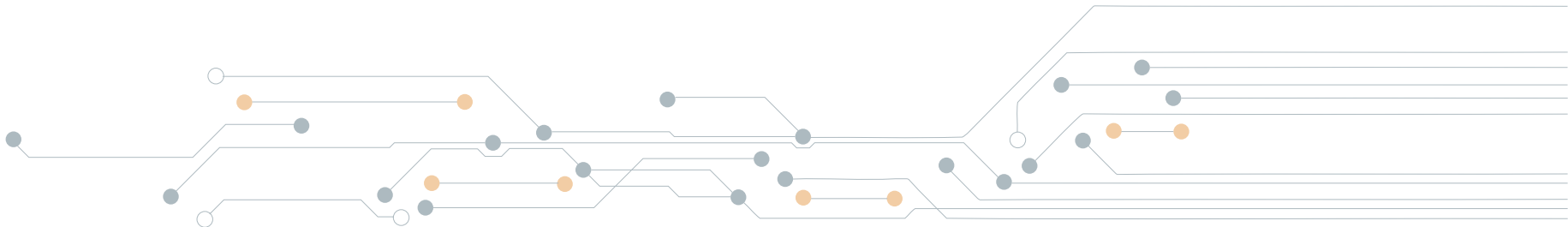
La programación con tipos genéricos consiste en escribir código que se puede reutilizar para objetos de distintos tipos. El tipo genérico conlleva la abstracción de las operaciones que se pueden realizar con conjuntos diferentes de datos. Por ejemplo en cualquier tipo de lista las operaciones a realizar serán casi siempre:

- Añadir elementos.
- Insertar elementos.
- Borrar elementos.
- Actualizar elementos.
- Obtener un elemento.
- Iterar por los elementos.

Independientemente del tipo de los elementos que formen la lista de los elementos, ya sea de `String`, de `File`, de objetos de clase `Persona`, de objetos de clase `Factura`.

Utilizar un tipo genérico a la hora de crear una colección permite:

- Eliminar la necesidad de moldeos.
- Controlar en tiempo de compilación que todos los elementos son del tipo genérico.



La tabla siguiente recoge como utilizar los tipos genéricos:

Categoría	Sin tipo genérico	Con tipo genérico
<b>Declaración de clase</b>	<code>public class Collection extends AbstractCollection Implements Lista</code>	<code>public class Collection&lt;E&gt; extends Abstract Collection &lt;E&gt; Implements Lista &lt;E&gt;</code>
<b>Declaración constructor</b>	<code>public Collection( int capacidad);</code>	<code>public Collection &lt;E&gt; ( int capacidad);</code>
<b>Declaración de método</b>	<code>public void add(Object o); public Object get(int index);</code>	<code>public void add(E o); public E get(int index);</code>
<b>Declaración de variables</b>	<code>Collection coleccion1; Collection coleccion2;</code>	<code>Collection &lt;String&gt;coleccion1; Collection &lt;Persona&gt;coleccion2;</code>
<b>Creación de objeto</b>	<code>coleccion1 = new Collection (10); coleccion2 = new Collection(20);</code>	<code>coleccion1 = new Collection &lt;String&gt; (10); coleccion2 = new Collection &lt;Persona&gt; (20);</code>

**Tabla 11.1: Utilización de tipos genéricos vs no genéricos**

<E> representa la variable de tipo, que será sustituida en la utilización concreta del tipo genérico por un tipo concreto, por ejemplo <String>.

En el ejemplo siguiente se muestra el código de declaración de una clase genérica sencilla. La clase permite gestionar parejas de valores de distintos tipos.

```
package com.juan.pareja;
public class Pareja<E> {
    private E primero;
    private E segundo;
    public Pareja() {
        primero = null;
        segundo = null;
    }
    public Pareja(E primero, E segundo) {
        this.primero = primero;
        this.segundo = segundo;
    }
    public E getPrimero() { return primero;}
    public E getSegundo() { return segundo;}
    public void setPrimero(E valor) { primero =
valor;}
    public void setSegundo(E valor) { segundo =
valor; }
}
```

Una utilización de este tipo genérico con tipo concreto String, es la que se muestra en el código siguiente:

```
public class PruebaPareja {

    public static void main(String[] args) {

        String[] palabras = { "tenedor", "cuchillo",
                               "cuchara", "taza", "vaso", "plato" };
        Pareja<String> mm = ArrayAlgo.
minmax(palabras);
        System.out.println("menor = " +
mm.getPrimero());
        System.out.println("mayor = " +
mm.getSegundo());
    }
}

class ArrayAlgo {
    /**Devuelve una pareja de palabras.
     * min es la menor en orden alfabético. max
     * es la mayor en orden alfabético */
    public static Pareja<String> minmax(String[] a) {
        if (a == null || a.length == 0) return
null;

        String min = a[0];
```

```
String max = a[0];
        for (int i = 1; i < a.length; i++) {
            if (min.compareTo(a[i]) > 0)
                min = a[i];
            if (max.compareTo(a[i]) < 0)
                max = a[i];
        }
        return new Pareja<String>(min, max);
    }
}
```

## 2. Colecciones, tipos y operaciones

En Java hay varios tipos de colecciones. Cada tipo distinto de colección tiene declaradas sus operaciones en una interfaz que la caracteriza. Una gran división de todas las colecciones, sería según que implementen la interface **Collection**, para colecciones de un sólo valor, o **Map** para colecciones de dos valores. A su vez de **Collection** se derivan dos grandes grupos, las que implementan la interface **Set** y las que implementan la interface **List**.

En el diagrama de la imagen se muestra algunas de las más utilizadas.

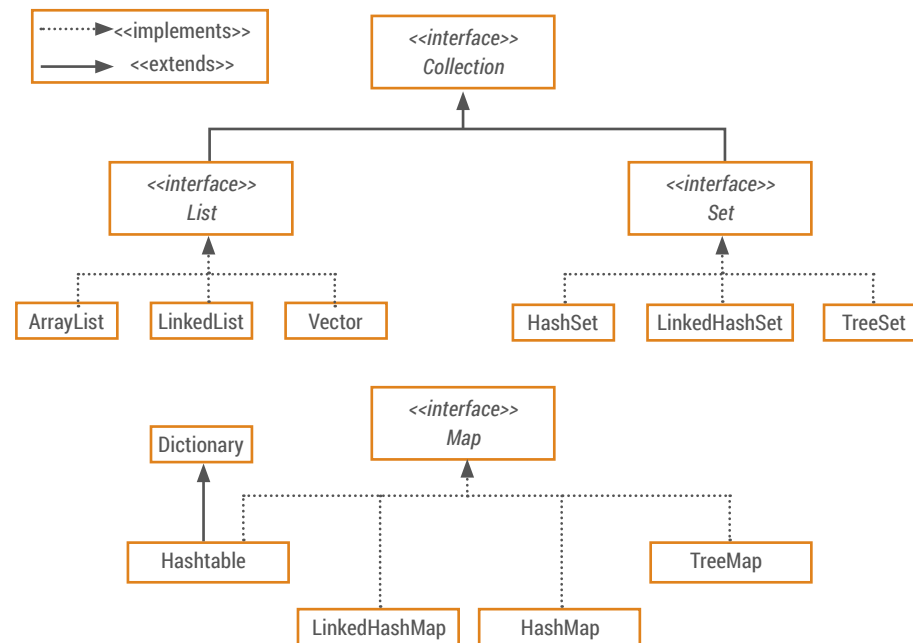


FIGURA 11.1: JERARQUÍA DE INTERFACES Y CLASES DE COLECCIONES.

Cualquier **Set** o **List**, las operaciones comunes que implementan por heredar de **Collection** son:

- **boolean add(E e)**: Añade un nuevo elemento al final de la colección.
- **boolean remove(E e)**: Elimina la primera ocurrencia del elemento indicado.
- **boolean contains(E e)**: Comprueba si el elemento especificado está en la colección.
- **void clear()**: Elimina todos los elementos de la colección.
- **int size()**: Devuelve el número de elementos en la colección.
- **boolean isEmpty()**: Comprueba si la colección está vacía.

Los siguientes métodos combinan dos colecciones. El carácter “?”, llamado comodín en estas funciones denota cualquier tipo de **Collection**:

- **boolean addAll(Collection<?> c)**: Añade todos los elementos de la colección c.
- **boolean removeAll(Collection<?> c)**: Elimina todos los elementos de la colección que están en la colección c.
- **boolean containsAll(Collection<?> c)**: Comprueba si coinciden las dos colecciones.
- **boolean retainAll(Collection<?> c)**: Elimina todos los elementos a no ser que estén en c. Permanecen en la colección los que están también la colección c.

## 3. Listas

Una lista es una estructura secuencial, donde cada elemento tiene un índice o posición. Los arrays vistos hasta ahora también tienen este comportamiento de las listas, pero no son dinámicos.

El índice de una lista es siempre un entero y el primer elemento ocupa la posición 0. Para trabajar con ellas se utiliza el interfaz **List<E>**. Las implementaciones más recomendadas son:

- **ArrayList<E>**, acceden a una posición de forma muy rápida.
- **LinkedList<E>**, para trabajar con inserciones y borrado muy rápidos.

La interfaz **List<E>** añade los elementos siguientes a los heredados de **Collection<E>**:

- **boolean add(int indice, E e)**: Inserta un nuevo elemento en una posición. El elemento que estaba en esta posición y los siguientes pasan a la siguiente.
- **E get(int indice)**: Devuelve el elemento en la posición especificada.
- **int indexOf(E e)**: Primera posición en la que se encuentra un elemento; -1 si no está.
- **int lastIndexOf(E e)**: Última posición del elemento especificado; o -1 si no está.
- **E remove(int indice)**: Elimina el elemento de la posición indicada.
- **E set(int indice, E e)**: Pone un nuevo elemento en la posición indicada. Devuelve el elemento que se encontraba en dicha posición anteriormente.

En el código siguiente se muestra la utilización de una lista de tipo **ArrayList<Number>**.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
public class PruebaArrayList {

    public static void main(String[] args) {
        Random aleatorio = new Random();
        List <Number> listanumeros = new
ArrayList<Number>();
        int limite = aleatorio.nextInt(24)+1;
        for(int i=1; i<limite; i++){
            int tipo = aleatorio.nextInt(4)+1;
            switch(tipo){
                case 1:
                    listanumeros.add(aleatorio.
nextInt(1000)+1); break;
                case 2: listanumeros.add(aleatorio.
nextLong()); break;
                case 3: listanumeros.add(aleatorio.
nextFloat()); break;
                case 4: listanumeros.add(aleatorio.
nextDouble()); break;
            }
        }
        System.out.println("Elementos lista: "
+ listanumeros.size());
        for (Number numero : listanumeros){
            System.out.println(numero);
        }
    }
}
```

```
        int indice = aleatorio.
nextInt(limite-1);
        System.out.println("Elemento "+indice+": "
+listanumeros.
get(indice));
        listanumeros.remove(indice);
        System.out.println("Elementos lista: "
+ listanumeros.size());
    }
}
```



## 4. Conjuntos (Sets)

Los conjuntos son estructuras de datos donde los elementos no conservan el orden de entrada como en los **List**. No permiten duplicados, cuando se producen no genera error, simplemente lo ignora. Para definirlos se utiliza la interface **Set<E>**, que no añade nuevos métodos a la interfaz **Collection<E>**.

Se pueden utilizar diferentes implementaciones:

- **HashSet<E>**, es la más eficiente (implementación con tabla hash).
- **TreeSet<E>**, los elementos quedan ordenados (implementación con árbol).

En la imagen siguiente se utilizan un **ArrayList**, un **HashSet** y un **TreeSet**, en las tres colecciones se intentan añadir números Integer en una cantidad aleatoria, para las tres por igual. Al ejecutar el código se comprueba:

- En **ArrayList**, están todos los números generados en el orden en que fueron añadidos.
- En **HashSet**, se han ignorado los repetidos y el orden no es previsible.
- En **TreeSet**, se han ignorado los repetidos y están ordenados de menor a mayor.

```

ist<Integer> listnumeros =
    new ArrayList<Integer>();
Set<Integer> hashsetnumeros =
    new HashSet<Integer>();
Set<Integer> treesetnumeros =
    new TreeSet<Integer>();

```

```

for (int i = 1; i < cantidad; i++) {
    n = aleatorio.nextInt(49) + 1;
    listnumeros.add(n);
    hashsetnumeros.add(n);
    treesetnumeros.add(n);
}

```

```

System.out.println("Elementos en el ArrayList: " + listnumeros.size());
for (Integer numero : listnumeros) {
    System.out.print(numero+ " ");
}
System.out.println();
System.out.println("Elementos en el HashSet: " + hashsetnumeros.size());
for (Integer numero : hashsetnumeros) {
    System.out.print(numero+ " ");
}
System.out.println();
System.out.println("Elementos en el TreeSet: " + treesetnumeros.size());
for (Integer numero : treesetnumeros) {
    System.out.print(numero+ " ");
}

```

```

C:\Windows\system32\cmd.exe
C:\codigos>java -cp . com.juan.conjuntos.PruebaSet
Elementos en el ArrayList: 23
20 31 21 44 10 46 48 37 46 34 46 5 20 32 4 13 12 18 15 36 23 35 13
Elementos en el HashSet: 19
32 34 35 4 36 37 5 10 44 12 13 46 15 48 18 20 21 23 31
Elementos en el TreeSet: 19
4 5 10 12 13 15 18 20 21 23 31 32 34 35 36 37 44 46 48
C:\codigos>

```

FIGURA 11.2: EJEMPLO UTILIZACIÓN ARRAYLIST, HASHSET Y TREESSET.

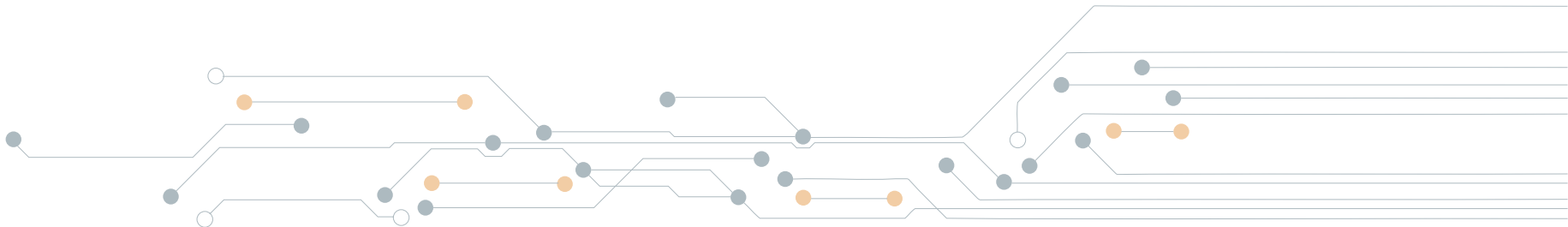
## 5. Diccionarios y Mapas (Maps)

Los diccionarios son estructuras de datos en las que cada elemento tiene asociado una clave utilizada para recuperarlo. Para definirlos se utiliza la interfaz **Map<K,V>**. En este caso se trabaja con dos clases una que se utiliza como **clave (K)** y otra para almacenar **los valores (V)**. Cada elemento se almacena mediante un par de objetos **(K,V)**. Esta estructura de datos permite obtener el **objeto V** muy rápidamente, a partir de **su clave K**. Por ejemplo, se pueden almacenar objetos de la clase Socios y utilizar como clave su "numero\_socio" en un String. Así, a partir del "numero\_socio" un diccionario encontrará el socio asociado rápidamente.

Algunas de las implementaciones de este interface son: **HashMap<K,V>** (ignora repetidos), **TreeMap<K,V>** (ordenados por clave) y **LinkedHashMap<K,V>** (orden de inserción).

Los métodos de la interface **Map<K,V>** son:

- **V put(K clave, V valor):** Añade un nuevo par clave-valor al diccionario.
- **V get(Object clave):** Da el valor asociado a una clave o null si no se encontró.
- **V remove(Object clave):** Elimina el par clave-valor que corresponde a la clave.
- **boolean containsKey(Object clave):** Comprueba si está la clave especificada.
- **boolean containsValue(Object valor):** Comprueba si está el valor.
- **Set keySet():** Devuelve un conjunto con las claves contenidas en el diccionario.
- **Collection values():** Devuelve una colección con solo los valores.
- **boolean isEmpty():** Comprueba si la colección está vacía.
- **int size():** Devuelve el número de elementos que contiene la colección.
- **void clear():** Elimina todos los elementos de la colección.



La iteración en los Map puede ser de tres formas:

- Con **keySet()**, por las claves

```
Map<String, Object> map = ...;
for (String key : map.keySet()) {
    // ...
}
```

- Con **values()**, por los valores:

```
for (Object value : map.values()) {
    // ...
}
```

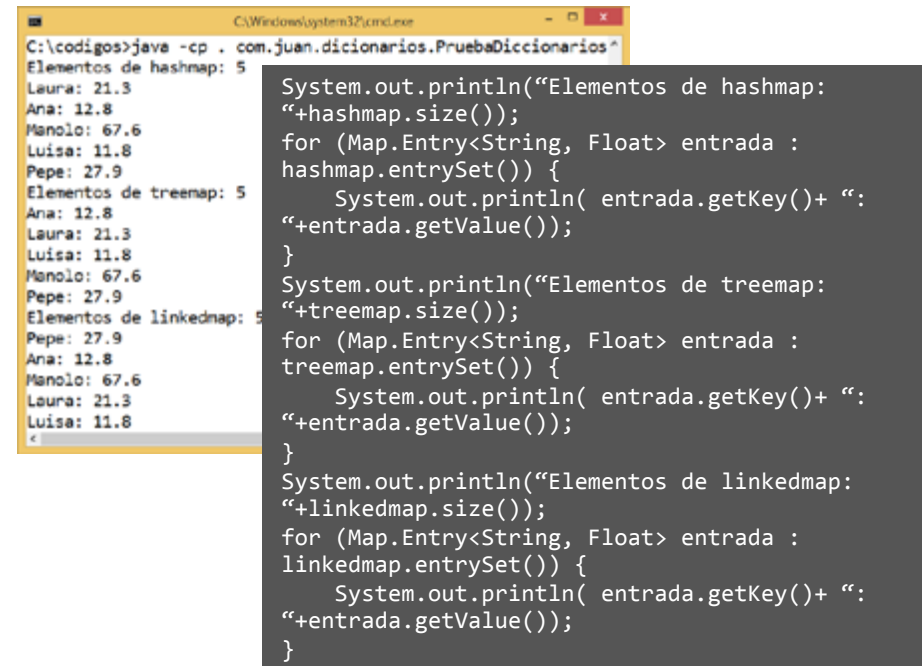
- Con **entrySet()**, por las claves y los valores:

```
for (Map.Entry<String, Object> entry : map.entrySet()) {
    String key = entry.getKey();
    Object value = entry.getValue();
    // ...
}
```

En la imagen siguiente se utilizan tres diccionarios uno de clase **HashMap**, otro de clase **TreeMap** y el tercero de clase **LinkedHashMap**, se comprueba que se ignoran los valores repetidos y se muestran según el criterio de orden de cada una de las tres clases.

```
Map<String, Float> hashmap =
    new HashMap<String,
Float>();
Map<String, Float> treemap =
    new TreeMap<String,
Float>();
Map<String, Float> linkedmap =
    new LinkedHashMap<String,
Float>();
```

```
for (int i=0; i<nombres.
length; i++){
    hashmap.
put(nombres[i], cuotas[i]);
    treemap.
put(nombres[i], cuotas[i]);
    linkedmap.
put(nombres[i], cuotas[i]);
}
```



```
System.out.println("Elementos de hashmap:
"+hashmap.size());
for (Map.Entry<String, Float> entrada :
hashmap.entrySet()) {
    System.out.println( entrada.getKey()+ ":
"+entrada.getValue());
}
System.out.println("Elementos de treemap:
"+treemap.size());
for (Map.Entry<String, Float> entrada :
treemap.entrySet()) {
    System.out.println( entrada.getKey()+ ":
"+entrada.getValue());
}
System.out.println("Elementos de linkedmap:
"+linkedmap.size());
for (Map.Entry<String, Float> entrada :
linkedmap.entrySet()) {
    System.out.println( entrada.getKey()+ ":
"+entrada.getValue());
}
```

FIGURA 11.3: EJEMPLO UTILIZACIÓN HASHMAP, TREEMAP Y HASHLINKEDMAP

## 6. Colas

Otra interfaz que hereda de **Collection** es **Queue**. Representa una colección ordenada de objetos al igual que una lista, pero en las colas se añaden elementos al final y se eliminan del principio, siguen el principio de “first in first out” (primero en entrar, primero en salir).

Tiene dos implementaciones importantes:

- **LinkedList<E>**, que implementa también la interfaz List y que como ya se indicó es muy eficiente para inserciones y borrados.
- **PriorityQueue<E>**, almacena los elementos ordenados por el criterio de orden natural del tipo concreto.

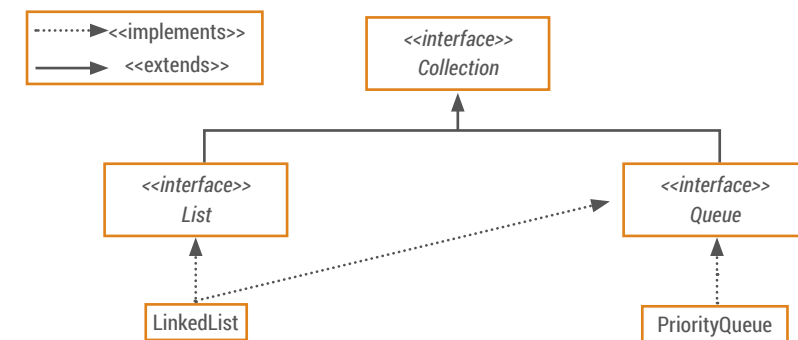


FIGURA 11.4: JERARQUÍA DE CLASES E INTERFACES PARA QUEUE.

Los métodos de la interfaz **Queue<E>** son:

- **E element()**: Retorna una referencia al primer elemento de la cola. Lanza la excepción `NoSuchElementException` si la lista esta vacía.
- **E peek()**: Retorna una referencia al primer elemento de la cola. Devuelve null si la cola esta vacía.
- **boolean offer(E e)**: Añade el elemento a la cola de la lista, lo coloca el último.
- **boolean add(E e)**: Añade el elemento a la cola de la lista, lo coloca el último. Lanza excepción si no se puede añadir el elemento (`IllegalStateException`, `ClassCastException`, `NullPointerException` o `IllegalArgumentException`).
- **E poll()**: Retorna el primer elemento de la cola y lo borra. Devuelve null si la lista esta vacía.
- **E remove()**: Retorna el primer elemento de la cola y lo borra. Lanza la excepción `NoSuchElementException` si la lista esta vacía.

En el ejemplo siguiente se crea una *Queue* de tipo **LinkedList**, se introducen tantos elementos de tipo Integer como la cantidad que indica el parámetro de llamada al programa. Se van eliminando uno a uno. Se utilizan las funciones **peek**, **poll** y **offer**.

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Random;

public class ContadorCola {
    public static void main(String[] args)
        throws InterruptedException {
        Random aleatorio = new Random();
        int time = Integer.parseInt(args[0]);
        int n = 0;
        Queue<Integer> cola = new LinkedList<Integer>();
        for (int i = time; i >= 0; i--) {
            System.out.print("primero: " + cola.peek());
            n = aleatorio.nextInt(1000);
            if (cola.offer(n)) {
                System.out.println(" ultimo: " + n);
            } else {
                System.out.println(" no se ha podido añadir " + n);
            }
        }
        for (Integer num : cola) {
            System.out.print(num + " ");
        }
        System.out.println();
        while (!cola.isEmpty()) {
            System.out.println("El primer elemento es: "
                + cola.poll());
            Thread.sleep(1000);
        }
        System.out.println("El primer elemento es: " + cola.poll());
    }
}
```

En el ejemplo siguiente se utiliza una **PriorityQueue** para ordenar los elementos de un **ArrayList**.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Random;
public class ColaPriorityAArrayList {
    static public void main(String [] args){
        Random aleatorio = new Random();
        int cant = aleatorio.nextInt(15);
        int num=0;
        List<Integer> lista = new
ArrayList<Integer>();
        for (int i=0; i<cant; i++){
            lista.add(num=aleatorio.nextInt(10));
        }
        System.out.println("ArrayList Antes");
        for(Integer n: lista){
            System.out.print(n+ " ");
        }
        System.out.println();
        lista = ordenarLista(lista);
        System.out.println("ArrayList Despues");
        for(Integer n: lista){
            System.out.print(n+ " ");
        }
    }
}
```

```
    }
    System.out.println();
}
static <E> List<E> ordenarLista(Collection<E> c)
{
    Queue<E> cola = new PriorityQueue<E>(c);
    List<E> lista = new ArrayList<E>();
    while (!cola.isEmpty())
        lista.add(cola.remove());
    return lista;
}
}
```

## 7. Iterable e Iterator

Todas las clases que implementan una interface **Collection** o subclase de esta, implementan la interface **Iterable**. La implementación de esta interface permite que esas colecciones se puedan recorrer con el bucle “**for-each**”, al igual que los arrays y los String.

En estos bucles no se puede alterar el tamaño de la colección, ni los valores de sus elementos. Para poder iterar por las colecciones con este tipo de operaciones se necesita la interface **Iterator**. La interface **Iterable** y por tanto todas las que heredan de ella y las clases que la implementan tienen el método **iterator()** que retorna una referencia a **Iterator**, con este iterador obtenido se pueden recorrer las colecciones y alterar su tamaño o cambiar los valores de sus elementos.

Las funciones de la interface Iterator son las siguientes:

- **boolean hasNext()**, devuelve true si la iteración tienen más elementos.
- **<E> next()**, devuelve el siguiente elemento de la iteración. lanza la excepción **NoSuchElementException** si no hay más elementos.
- **void remove()**, quita el elemento de la iteración de la colección, el elemento obtenido con **next**, sólo se puede llamar una vez por cada llamada de **next**. Si no se puede eliminar el elemento lanza una de estas dos excepciones **UnsupportedOperationException** o **IllegalStateException**. Este método es **default**, por tanto tiene ya definido un comportamiento que las clases que lo implementan no están obligadas a modificarlo.

La imagen siguiente muestra un diagrama de las relaciones entre las interfaces **Collection**, **Iterable** e **Iterator**.

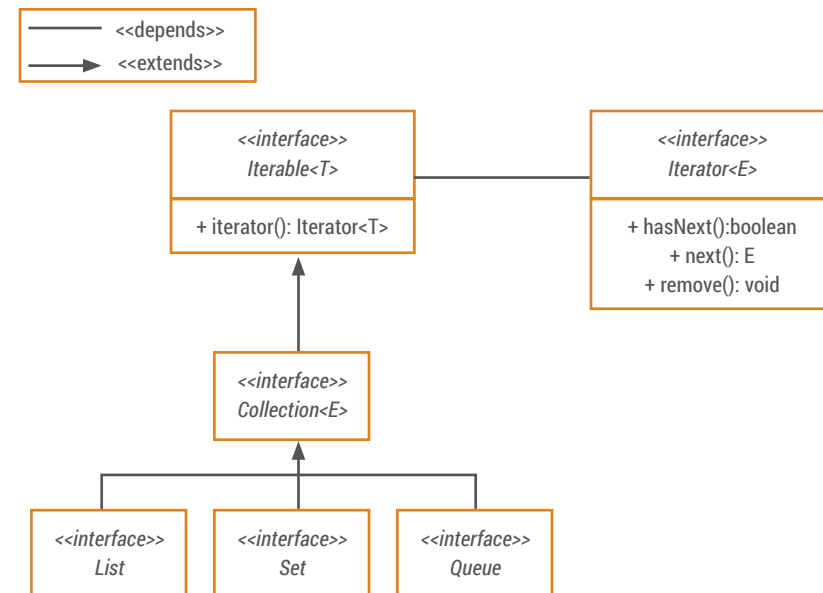


FIGURA 11.5: JERARQUÍA DE INTERFACES RELACIONADAS CON ITERABLE E ITERATOR.

En el ejemplo siguiente se utiliza **Iterator** para en un **ArrayList** de **Number** eliminar los elementos que sean de tipo **Float** y **Double**.

```
public static void main(String[] args) {
    Random aleatorio = new Random();
    List <Number> listanumeros = new
ArrayList<Number>();
    //Código que añade elementos a listanumeros

    for (Number numero : listanumeros){
        System.out.println(numero);
    }
    Iterator <Number> iterador= listanumeros.
iterator();
    while(iterador.hasNext()){
        Class clase= iterador.next().
getClass();
        if(clase.equals(Float.class)||
           clase.equals(Double.
class)){
```

```
            iterador.remove();
        }
    }
    for (Number numero : listanumeros){
        System.out.println(numero);
    }
}
```





## 8. Comparable

La interface **Comparable** se utiliza para determinar en una clase un orden distinto al natural, por ejemplo los **String** y los **Number** tienen ya su orden natural preestablecido, pero las subclases que se creen no tienen establecido un orden y por tanto si se necesita trabajar con colecciones y con funcionalidades de ordenación se debe implementar la interface **Comparable**.

La interfaz Comparable solamente tiene un método:

- **int compareTo(T o).** Se compara this con el objeto cuya referencia recibe como parámetro, y el valor devuelto será:

**-1, si el objeto es menor que el recibido como parámetro.**

**0, si son iguales.**

**1, si el objeto es mayor que el recibido como parámetro.**

En el ejemplo siguiente se muestra el uso de la implementación de **Comparable** en una clase.

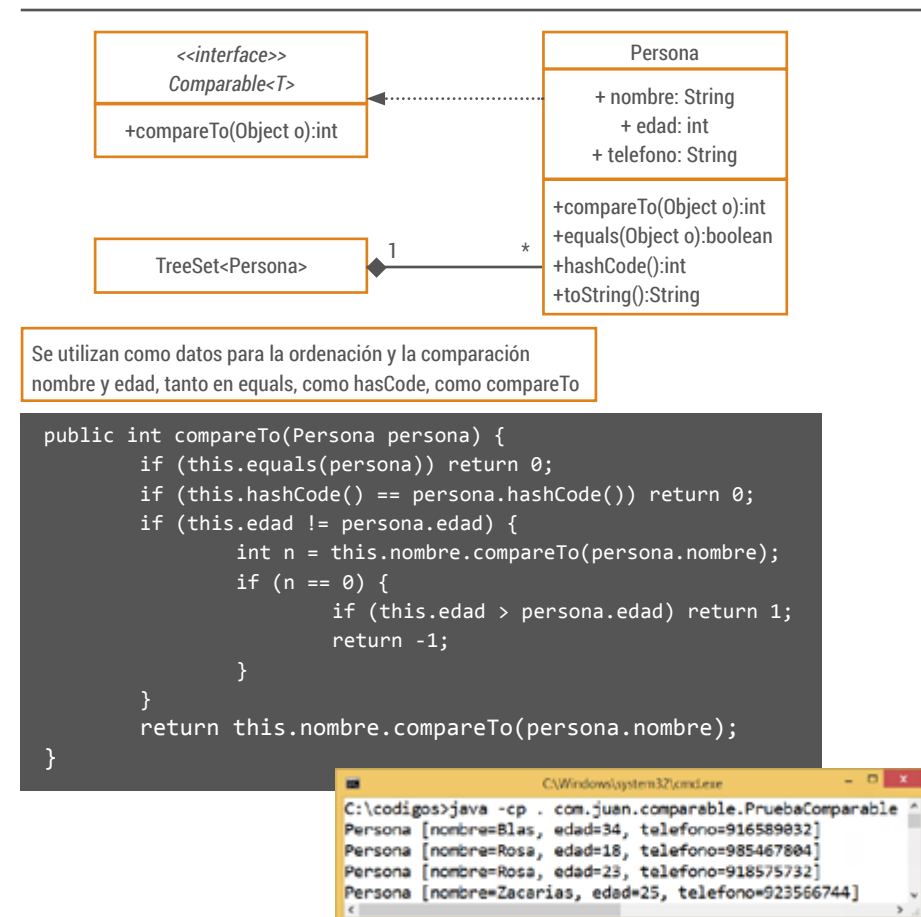


FIGURA 11.6: EJEMPLO DE LA INTERFACE COMPARABLE.

## 9. Comparator

La interfaz **Comparator**, al igual que **Comparable**, permite establecer el criterio de comparación entre objetos instanciados de una clase. La diferencia radica en que la clase que implementa **Comparator** es la que se indicará a la colección para que tenga en cuenta el método **compare** que define como se comparan los objetos.

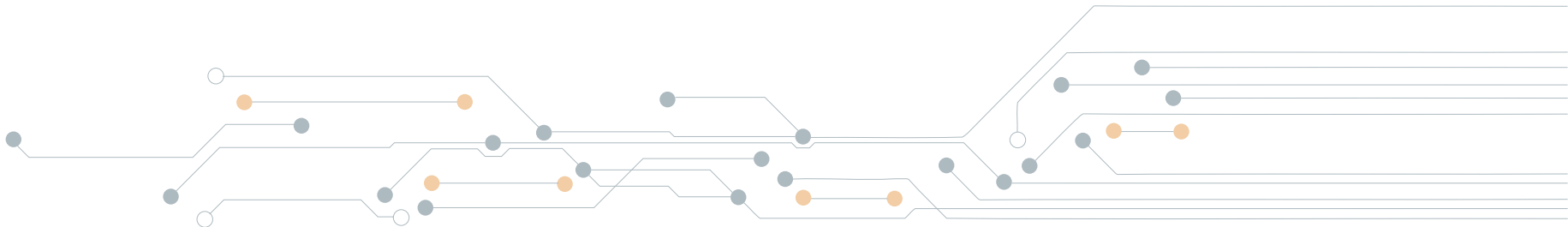
Esta interfaz también tiene solamente tiene un método:

- **int compare (T o, T o).** Se compara los dos objetos cuyas referencias se reciben como parámetros, y el valor devuelto será:
  - 1, si el primer objeto es menor que el segundo.
  - 0, si son iguales.
  - 1, si el primer objeto es mayor que el segundo.

Una forma de utilización de esta función es como se indica:

```
Collections.sort(<referencia_coleccion>, <referencia_
clase_comparator>);
```

Collections es una clase con utilidades static para trabajar con colecciones



En el ejemplo siguiente se muestra el uso de la implementación de **Comparator** con un criterio de comparación, utilizado en la funcionalidad de ordenación de **List** que tiene la clase de utilidades **Collections**.



FIGURA 11.7: EJEMPLO DE LA INTERFACE COMPARATOR

*Telefonica*

---

EDUCACIÓN DIGITAL