Colecciones en Java

Puri Arenas Sánchez (Grupos A y D)
Yolanda García Ruiz (Grupo E)
Facultad de Informática (UCM)
Curso 2019/2020

Justificación

- Una colección representa un grupo de objetos, conocidos como elementos.
- Cuando queremos trabajar con un conjunto de elementos, necesitamos una construcción donde poder:
  - Guardarlos
  - Consultarlos
  - Buscarlos, etc.

- En Java, se emplean clases genéricas para implementar las colecciones de objetos.
- Ejemplos:

  List<T> **ArrayList<T>**

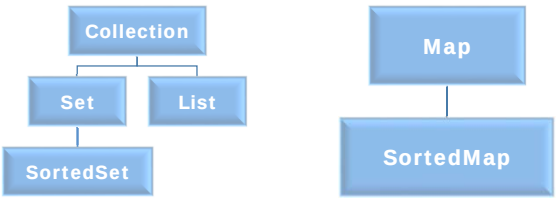
- Se encuentran implementadas en el `package java.util`

El uso de librerías reduce esfuerzos de programación

Colecciones

- Organización de las colecciones:
  - Interfaces: Manipulan los datos independientemente de los detalles de implementación.
  - Clases: Implementan las interfaces.

Interfaces Java Collections Framework	Se trata de un conjunto de interfaces que permiten que las colecciones sean manipuladas independientemente de su representación

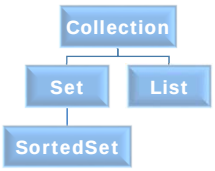


3

Interfaces Java Collections Framework

La interfaz genérica **Collection:**

- Permite almacenar cualquier tipo de objeto
- Permite usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección…
- Partiendo de la interfaz genérica **Collection** extienden otra serie de interfaces genéricas.
- Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.



Interface **List<T>:** Colección que mantiene sus elementos ordenados. Admite elementos duplicados.

Interface **Set<T> :** Colección que no puede tener objetos duplicados.

Interface **SortedSet: Set** que mantiene los elementos ordenados.

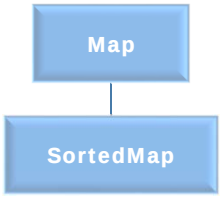
4

Interfaces Java Collections Framework

La interfaz **Map**

- Asocia claves a valores
- Esta interfaz no puede contener claves duplicadas
- Cada clave sólo puede tener asociado un valor

La subinterfaz **SortedMap:** es un **Map** que mantiene las claves ordenadas.



5

Interface Collection<T>

isEmpty(): ¿la colección está vacía?
size(): número de elementos en la colección
contains(Object e): ¿el objeto **e** se encuentra en la colección?
containsAll(Collection c): ¿todos los elementos de c están en la colección?

add(T e): añade el objeto en la colección
addAll(Collection<? extends T> c): añade la colección c
clear(): elimina todos los elementos de la colección
equals(Object): ¿es igual esta colección y la proporcionada?
remove(Object): elimina una aparición del objeto (opcional).
removeAll(Collection <?> c): elimina todos los elementos en c.
retainAll(Collection): se queda sólo con esos objetos (opcional).
toArray(): devuelve un array con los objetos de la colección.

6

```
public interface Collection<E> extends Iterable<E> {
   int size();
   boolean isEmpty();
   boolean contains(Object o);
   boolean add(E e);
   boolean remove(Object o);
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c);
   boolean removeAll(Collection<?> c);
   void clear();
   ...
}
```

extends Iterable

Permite recorrer los elementos de la colección

7

```
public interface Collection<E> extends Iterable<E> {
   int size();
   boolean isEmpty();
   boolean contains(Object o);
   boolean add(E e);
   boolean remove(Object o);
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c);
   boolean removeAll(Collection<?> c);
   void clear();
   ...
}
```

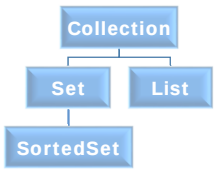
Tipo Object
se permite que se pueda eliminar o buscar un elemento a través de alguna de sus componentes

Tipo T
Solo se permite añadir un elemento de tipo T

8

Interface List<T>

- Colecciones (secuencias) en las que cada elemento ocupa una posición identificada por un índice.
- El primer índice es el 0.
- Las listas admiten duplicados.



- Además de las operaciones de **Collection<T>**, dispone algunas operaciones adicionales:

`add( int i, T e):` `añade el objeto en la posición indicada`

`T get(int i):` `devuelve el objeto de la posición i`

`set(int i, T e):` `reemplaza el objeto en esa posición por el objeto que se proporciona.`

`T remove(int i) :` `elimina el objeto de la posición i`

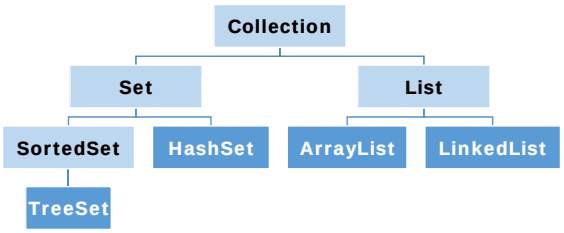
Interface List<T>

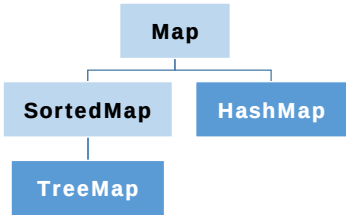
```
public interface List<T> extends Collection<T> {
   T get(int index);
   set(int index, T element);
   void add(int index, T element);
   T remove(int index);
   int indexOf(Object o);
   int lastIndexOf(Object o);
   List<T> subList(int fromIndex, int toIndex);
   ...
}
```

se puede buscar por alguna componente del objeto

Interfaces Java Collections Framework

- Estas interfaces son genéricas. No se proporciona una interface para cada posible tipo.
- Estas interfaces son implementadas en clases concretas que constituyen estructuras de datos reutilizables.



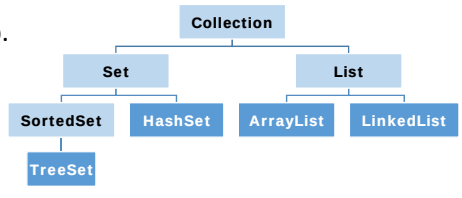


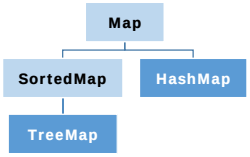
- ¿Cómo se trabaja con las colecciones?
  - Se elige una interface para la funcionalidad deseada.
  - Se elige una clase que implemente la interface con la eficiencia deseada o alguna otra característica particular.
  - Se adapta (extiende) si es necesario.

11

Implementaciones de las interfaces JFC.

- **Tablas hash:**
  - HashSet: implementa la interfaz Set.
  - HashMap: implementa la interfaz Map.
- **Arrays de tamaño variable:**
  - Vector: implementa la interfaz List.
  - Stack: implementa la interfaz List.
  - ArrayList<T>: implementa la interfaz List.
- **Árboles equilibrados:**
  - TreeSet: implementa la interfaz Set.
  - TreeMap: implementa la interfaz Map.
- **Listas enlazadas:**
  - LinkedList<T>: implementa la interfaz List.

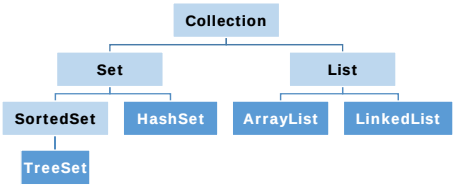




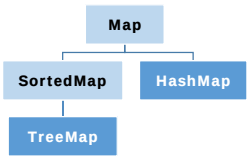
12

Implementaciones de las interfaces JFC.

- La clase **ArrayList<T>** implementa la interfaz **List<T>.**
- **ArrayList<T >** implementación basada en arrays redimensionables
- Las operaciones de creación y consulta son rápidas.
- Dificultades para crecer (reservar memoria nueva, copiar los elementos del array antiguo y liberar la memoria) y para insertar y/o borrar elementos intermedios (desplazar los elementos que están detrás del elemento borrado o insertado).



- La clase **LinkedList<T>** implementa la interfaz **List<T>.**
- Su implementación está basada en listas doblemente enlazadas.
- Las operaciones de añadir y modificación son más rápidas, sobre todo al principio y final de la lista.
- El acceso a elementos para consulta o modificación son lentos (excepto el primero y el último).



La clase `Vector` implemanta `List<T>`

- La clase **Vector<T>** implementa la interface **List<T>**
- Similar a **ArrayList<T>**
  - La operación **resize** duplica su dimensión, mientras que **ArrayList** solo crece la mitad de su tamaño
  - Sólo una hebra puede manipular un **Vector** (sincronizado), mientras que un **ArrayList** puede ser manipulado simultáneamente por varias hebras (no sincronizado).

La clase **Vector** se añadió con posterioridad a las colecciones. *Sólo conviene usarlos cuando en concurrencia no queremos que varias hebras accedan a la colección*.

La clase Vector: Ejemplo de uso

```
public class Myvector extends Vector<Integer>{

  public static void main( String[] args) {
     Myvector v = new Myvector();
     v.add(new Integer(4));

     }
}
```


La clase Stack extiende Vector<T>

- La clase **Stack<T>** extiende la clase **Vector<T>** y ofrece las funcionalidades concretas de una pila (push y pop).

Ejemplo: listas de elementos de cualquier tipo

Supongamos que queremos crear una clase que sirva para almacenar cualquier tipo de objetos:

- Lista de instrucciones
- Lista de comandos
- Lista de direcciones de correo
- …

```
public class ListaElementos <Elem>{

    private Vector<Elem> v;

    public ListaElementos() {
        v = new Vector<Elem>();
    }
    public Elem getInstruction(int i){
        return v.elementAt(i);
    }
    public void addInstruction(Elem e){
        v.add(e);
    }
```

```
ListaElementos<Instruction> program = new ListaElementos<Instruction>();
ListaElementos<Integer> posiciones = new ListaElementos<Integer>();
ListaElementos<Commands> comandos = new ListaElementos<Commands>();
```

20

Ejemplo con ArrayList

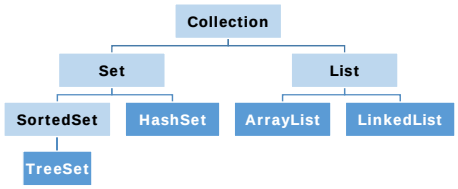
```
import java.util.*;

public class Listas {
    public static void main(String[] args){
    ArrayList<Persona> lista = new ArrayList<Persona>();
    Persona p1 = new Persona("223344", 32, "Juan", "Pardo Gil");
    lista.add(p1);
    Persona p2 = new Persona("11133322", 21, "Rosa", "Garrido Aguado");
    lista.add(p2);

    int posicion = lista.indexOf(p1);
    lista.remove(posicion);
    lista.set(0, p1);
    if(lista.isEmpty())
      System.out.println("Lista vacía");
    }
```

24

Interface Set<T>

- La interface **Set<T>** extiende la interface **Collection<T>**.
- No puede contener elementos repetidos.
- No garantiza que los elementos se mantendrán en ningún orden en particular.
- La interface **Set<T>** no declara ningún método adicional a los de **Collection<T>**.

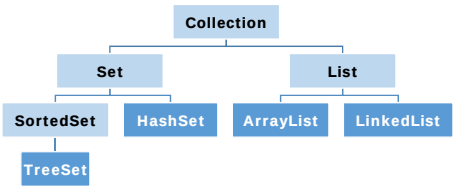
- Utilizando los métodos de **Collection**, los **Sets** permiten realizar operaciones algebraicas de unión, intersección y diferencia.

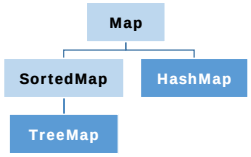


25

Interface Set<T>

- La clase **HashSet<T>** implementa la interfaz **Set<T>.**
- **HashSet<T>** guarda los elementos del conjunto en una tabla hash. Los elementos repetidos se evitan comparando los hashCode (se generan automáticamente) de los elementos.
- En caso de ser iguales se compara con **equals**.
- La interfaz **SortedSet<T>** extiende a la interfaz **Set<T>.** Es similar, pero los elementos se mantienen internamente ordenados.
- **SortedSet<T>** es implementada por la clase **TreeSet<T>,** que implementa los conjuntos ordenados basándose en árboles binarios balanceados. Para poder utilizarlos hay que definir un orden.
- Las operaciones de añadir y modificación son más lentas en **TreeSet** que en **HashSet**. Las búsquedas son más rápidas en estructuras ordenadas.
- En general, es preferible usar un **HashSet** y posteriormente ordenarlo y transformarlo en un **TreeSet**.





26

Interface SortedSet

- La interface ***SortedSet*** extiende la interface ***Set*** y añade los métodos:

`Object first()`: El elemento mas pequeño

`SortedSet headSet(Object)`: Elementos menores que Object

`Object last()`: El elemento mayor

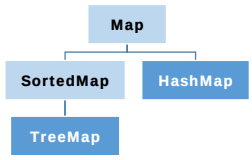
`SortedSet subSet(Object, Object)` : Elementos entre el primer objeto proporcionado y el segundo

`SortedSet tailSet(Object)`: Elementos mayores o iguales que Object

La interfaz Map<K,V>

- Un ***Map*** es una estructura de datos agrupados en parejas (***clave,valor)***.
- Pueden ser considerados como una tabla de dos columnas.
- La ***clave*** debe ser única y se utiliza para acceder al ***valor***.
- Aunque la interface ***Map*** no deriva de ***Collection***, es posible ver los ***Maps*** como colecciones de ***claves***, de ***valores*** o de parejas (***clave,valor)***.

- En su implementación contiene distintas colecciones:
  - Conjunto de claves (**Set<K>**)
  - Colección de valores (**Collection<V>**)
  - Conjunto de pares **<clave,valor>** (**Map.Entry<K,V>**)



La interfaz Map<K,V>

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

`Set <Map.Entry<K,V>> entrySet():` devuelve una "vista" del Map como Set. Es decir conjunto de pares Map.Entry<K,V>

`V get(K clave):` permite obtener el valor a partir de la clave.

`Set<K> keySet() :` Devuelve una "vista" de las claves como Set.

`V put(K clave, V valor):` permite añadir una pareja clave/valor

`V remove(K clave):` elimina una pareja clave/valor a partir de la clave.

`Collection<V> values():` devuelve la colección de los valores

Los iteradores - Interface Iterator

- Los **iteradores** permiten recorrer colecciones de elementos sin preocuparse de la implementación subyacente.
- La interfaz **Iterable<T>** es implementada por aquellas clases sobre las que se puede iterar, por ejemplo las colecciones.
- También es posible recorrer una colección usando la instrucción **`for each`**

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    ...
}
```

```
public interface Iterable<E> {
    Iterator<T> iterator();
}
```

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Recorrido de la colección con Iterator

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

```
public interface Iterable<E> {
    Iterator<T> iterator();
}
```

hasNext(): ¿hay un elemento siguiente?

next(): devuelve el siguiente elemento de la colección.

remove(): elimina el último elemento accedido por el iterador. Solo se puede invocar una vez tras cada next().

Ejemplo

```
Public static void imprime(Collection<String> col){
    Iterator<String> it = col.iterator();
    while(it.hasNext())
            System.out.println(it.next());
}
```

35

Recorrido de la colección con Iterator

Ejemplo

```
public static void eliminarPares(List<Integer> lista){

    List<Integer> listaPares = new LinkedList<Integer>();

    Iterator<Integer> it = lista.iterator();

    while(it.hasNext())
         Integer n = it.next();
         if (n % 2 == 0)
             listaPares.add( n );

    lista.removeAll(listaPares);

}
```

36

Recorrido de la colección con for each

Ejemplo

```
public class Recorrido {

   public static void eliminaPares(List<Integer> lista) {
      List<Integer> listaPares = new LinkedList<Integer>();
      for (Integer i : lista)
         if (i % 2 == 0) listaPares.add(i);
      lista.removeAll(listaPares);
   }
   public static void main(String[] args) {
        List<Integer> lista = new LinkedList<Integer>();
        lista.add(4);
        lista.add(3);
        lista.add(2);
        lista.add(1);
        System.out.println(lista);
        eliminaPares(lista);
        System.out.println(lista);
    }
}
```

```
[ 4, 3, 2, 1 ]
[ 3, 1 ]
```


Map.Entry

- El método entrySet() de Map nos devuelve un set en el que cada elemento es del tipo Map.Entry<clave,valor>.

Ejemplo

Se puede entonces obtener un iterador para recorrerlo.

```
Iterator<Map.Entry<K,V>> it = miMapa.entrySet().iterator();

   while (it.hasNext()) {
      Map.Entry<K,V> e =(Map.Entry<K,V>)(it.next());
      if (e.getValue()==null)
               System.out.println(e.getKey());
   }
```

Interface Iterator

Ejemplo

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

    public static void prueba(Collection<String> c) {
        String[] lista = {"uno", "dos" , "tres", "cuatro", "tres"};
        for(int i = 0; i < lista.length; i++ )
        c.add( lista[i] );
        Iterator<String> it = c.iterator();
        while(it.hasNext())
        System.out.println( it.next() );
        System.out.println( "-------------------------");
    }
    public static void main(String args[]) {
        Collection<String> c;
        c = new ArrayList<String>(); prueba(c);
        c = new LinkedList<String>(); prueba(c);
        c = new HashSet<String>(); prueba(c);
        c = new TreeSet<String>(); prueba(c);
        }
    }
```


Interface ListIterator

- La interface **ListIterator** permite recorrer una lista en ambas direcciones, y hacer algunas modificaciones.
- Un objeto ListIterator nos permite no solo recorrer los elementos de una lista, sino realizar inserciones y eliminaciones de objetos en posiciones intermedias.

void add(Object): inserta en la posición actual el elemento proporcionado.

boolean hasNext(): ¿hay un elemento siguiente yendo hacia el final de la lista?

boolean hasPrevious(): ¿hay un elemento anterior yendo hacia el principio?

Object next(): devuelve el siguiente elemento de la lista.

int nextIndex(): devuelve el índice del siguiente elemento de la lista.

Object previous(): devuelve el elemento anterior de la lista.

int previousIndex(): devuelve el índice del elemento anterior de la lista.

void remove(): elimina el último elemento devuelto por next() o previous().

void set(Object): reemplaza el último elemento devuelto por next() o previous().

Colecciones ordenadas

Interfaces Comparable y Comparator

- Las colecciones ordenadas **SortedSet** y **SortedMap** necesitan mantener el orden.
- Para ello es necesario definir dicho orden.
- Para ello es necesario implementar la interfaz **Comparable<T>** que se encuentra en ***java.lang***

- También se puede utilizar la interfaz **Comparator<T>** en la constructora

```
public interface Comparable<T> {
        public int compareTo(T e);
}

public interface Comparator<T> {
        public int compare(T e1, T e2);
}
```

Este método compara su argumento implícito con el que se le pasa como argumento. Devuelve un entero negativo, cero o positivo según el argumento implícito sea ***anterior***, ***igual*** o ***posterior*** al objeto ***obj***.

Colecciones ordenadas
Interfaces Comparable y Comparator

```
public class Cuenta implements Comparable<Cuenta> {
    private int numCuenta;
    private int saldo;

    public Cuenta(int nc, int s) {
        this.numCuenta = nc;
        this.saldo = s;
    }
    @Override
    public int compareTo(Cuenta cuenta) {
        if (this.numCuenta == cuenta.numCuenta) return 0;
        else if (this.numCuenta < cuenta.numCuenta) return -1;
        else return 1;
    }
    public String toString() {
        return "Numero cuenta: " + this.numCuenta + "\n"
        + "Saldo: " + this.saldo;
}
```

```
public interface Comparable<T> {
        public int compareTo(T e);
}

public interface Comparator<T> {
        public int compare(T e1, T e2);
}
```

51

Colecciones ordenadas
Interfaces Comparable y Comparator

```
public interface Comparable<T> {
        public int compareTo(T e);
}

public interface Comparator<T> {
        public int compare(T e1, T e2);
}
```

```
public class Cuenta implements Comparable<Cuenta> {
        @Override
    public int compareTo(Cuenta cuenta) {

        ...
```

```
public class Cliente {
    private int nif;                                    // no se fija orden en la constructora
    private TreeSet<Cuenta> cuentas = new TreeSet<Cuenta>();
...
    // se añade teniendo en cuenta el orden definido en Cuenta
    public boolean addCuenta(Cuenta c)
                { return cuentas.add(c); }
    public String toString() {
            Iterator<Cuenta> it = cuentas.iterator();
            String s = "";
            while (it.hasNext()) {
                Cuenta c = it.next();
                s = s + c.toString() + "\n";
            }
            return s;
    }
}
```

5

Colecciones ordenadas
Interfaces Comparable y Comparator

```
public static void main(String[] args) {
        Cliente c = new Cliente(2);
        Cuenta cuenta1 = new Cuenta(1,1000);
        c.addCuenta(cuenta1);
        Cuenta cuenta3 = new Cuenta(3,100);
        c.addCuenta(cuenta3);
        Cuenta cuenta2 = new Cuenta(2,2000);
        c.addCuenta(cuenta2);
        System.out.println(c.toString());
}
```

```
Numero de cuenta: 1
Saldo: 1000
Numero de cuenta: 2
Saldo: 2000
Numero de cuenta: 3
Saldo: 100
```


Colecciones ordenadas
Interfaces Comparable y Comparator

- Se puede fijar el orden de una colección en su constructora utilizando la interfaz

```
public class Cliente {
...
        private TreeSet<Cuenta> cuentas;
        public Cliente(int nif, Comparator<Cuenta> comp) {
           this.nif = nif;
           cuentas = new TreeSet<Cuenta>(comp); // pasamos un orden
        }
        public boolean addCuenta(Cuenta c){ return cuentas.add(c); }
}

public class Cuenta {                    // no implementa a Comparable<T>
        private int numCuenta;
        private int saldo;
        ...
        public int getNumCuenta() { return this.numCuenta; }
        public int getSaldo(){ return this.saldo; }
}
```

Colecciones ordenadas

Interfaces Comparable y Comparator

```
// ordena por número de cuenta
public class OrdenarPorNumeroCuenta implements Comparator<Cuenta> {
    @Override
    public int compare(Cuenta cuenta1, Cuenta cuenta2) {
        if (cuenta1.getNumCuenta() == cuenta2.getNumCuenta()) return 0;
        else if (cuenta1.getNumCuenta() < cuenta2.getNumCuenta()) return -1;
        else return 1;
    }
}
// ordena por saldo
public class OrdenarPorSaldo implements Comparator<Cuenta>{
    @Override
    public int compare(Cuenta cuenta1, Cuenta cuenta2) {
        if (cuenta1.getSaldo() == cuenta2.getSaldo()) return 0;
        else if (cuenta1.getSaldo() < cuenta2.getSaldo()) return -1;
        else return 1;
    }
}
```


Colecciones ordenadas

Interfaces Comparable y Comparator

- Cada cliente tiene un orden distinto en sus cuentas

```
Cliente c1 = new Cliente(2, new OrdenarPorNumeroCuenta());
Cuenta cuenta1 = new Cuenta(1,1000);
c1.addCuenta(cuenta1);
Cuenta cuenta3 = new Cuenta(3,3000);
c1.addCuenta(cuenta3);
Cuenta cuenta2 = new Cuenta(2,2000);
c1.addCuenta(cuenta2);
System.out.println(c1.toString());

Cliente c2 = new Cliente(2, new OrdenarPorSaldo());
cuenta1 = new Cuenta(1,3000);
c2.addCuenta(cuenta1);
cuenta3 = new Cuenta(3,1000);
c2.addCuenta(cuenta3);
cuenta2 = new Cuenta(2,2000);
c2.addCuenta(cuenta2);
System.out.println(c2.toString());
```

Ejemplo

```
public static void test2(Map<Person, String> m) {
        System.out.println("------");
        m.put(new Person("Mike", 23), "Prof.");
        m.put(new Person("John", 23), "Student");
        m.put(new Person("Andres", 5), "Admin");
        System.out.println(m);
}
```

```
public class Person {
        private int id;
        private String name;
        public Person(String name, int id) {...}
        @Override
        public String toString() { return name +":"+id; }
}
```

```
public static void main(String[] args) {
        test2(new HashMap<Person, String>()); // estructura sin orden
}
```

```
------
{Mike:23=Prof., John:23=Student, Andres:5=Admin}
```

```
public static void main(String[] args) {
        test2(new TreeMap<Person, String>()); // estructura con orden
}
```

Error !!

Ejemplo Sol1

```
public static void test2(Map<Person, String> m) {
        System.out.println("------");
        m.put(new Person("Mike", 23), "Prof.");
        m.put(new Person("John", 23), "Student");
        m.put(new Person("Andres", 5), "Admin");
        System.out.println(m);
}
```

```
public class Person {
        private int id;
        private String name;
        public Person(String name, int id) {...}
        public int getId() { return id; }
        @Override
        public String toString() { return name +":"+id; }
}
```

```
public class Orden implements Comparator<Person> {
        @Override
        public int compare(Person o1, Person o2) {
                return o1.getId() > o2.getId() ? 1 : o1.getId() < o2.getId() ? -1 : 0;
        }
}
```

```
public static void main(String[] args) {
        test2(new TreeMap<Person, String>(new orden())); // estructura con orden
}
```

```
------
{Andres:5=Admin, Mike:23=Student}
```

Ejemplo Sol2

```
public static void test2(Map<Person, String> m) {
        System.out.println("------");
        m.put(new Person("Mike", 23), "Prof.");
        m.put(new Person("John", 23), "Student");
        m.put(new Person("Andres", 5), "Admin");
        System.out.println(m);
}
```

```
public class Person implements
                        Comparable<Person> {
    private int id;
    private String name;
    public Person(String name, int id) {...}
    @Override
    public int compareTo( Person o2) {
        return this.getId() > o2.getId() ? 1 :
            dis.getId() < o2.getId() ? -1 : 0;
    }

    @Override
    public String toString() { return name +":"+id; }
}
```

```
public static void main(String[] args) {
        test2(new TreeMap<Person, String>()); // estructura con orden
}
```

```
------
{Andres:5=Admin, Mike:23=Student}
```