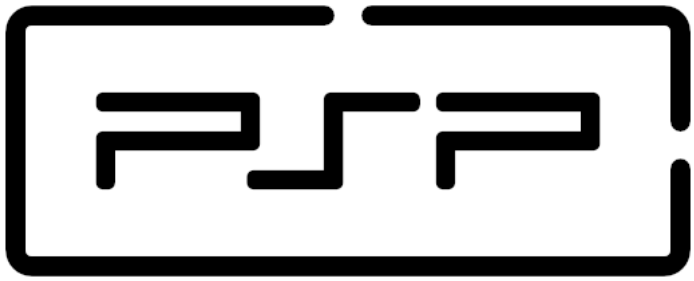


Programación de Servicios y Procesos

3.7 Anexo III - HashMap hoja de referencia de los alumnos

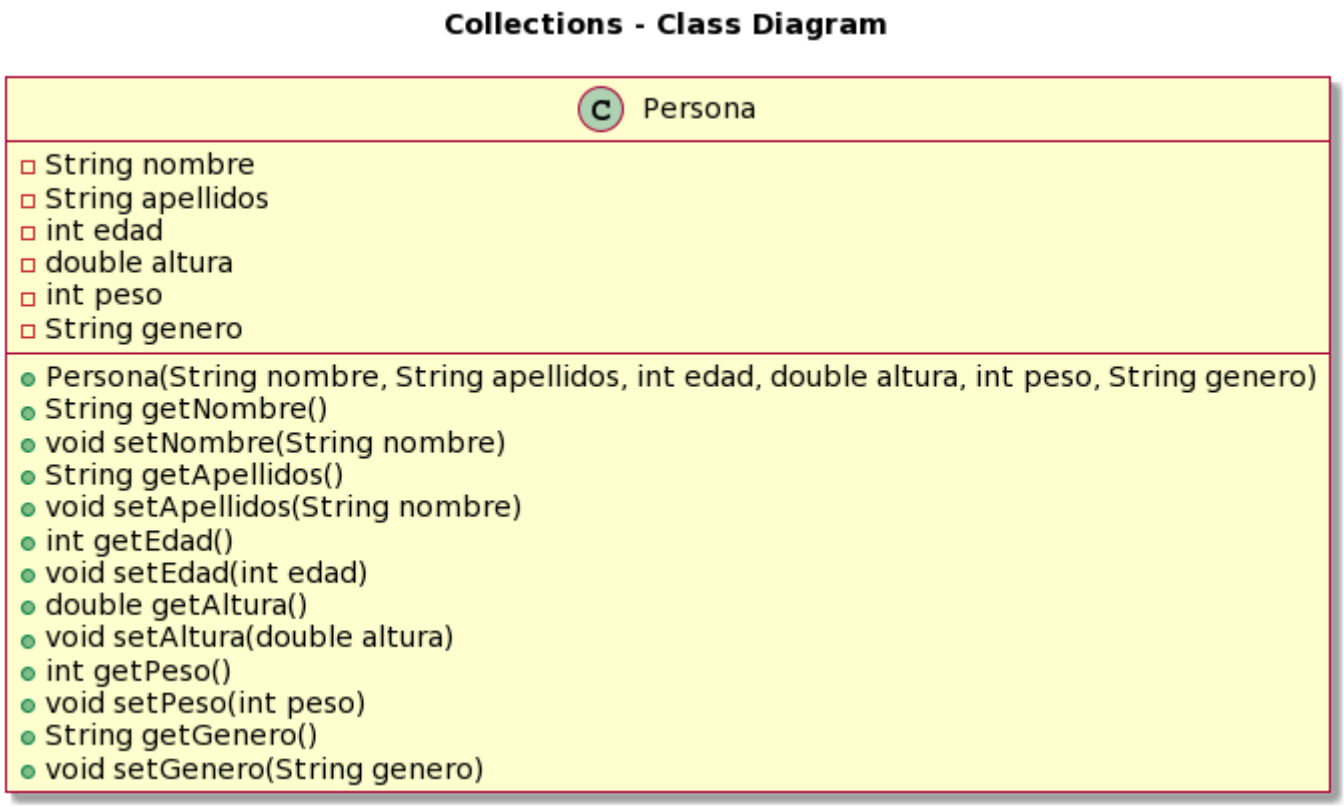
3.7 Anexo III - HashMap hoja de referencia de los alumnos

Autoría

Esto es un extracto del trabajo *Reto I (Challenge I)* realizado por mis alumnos como parte del módulo de PSP. He tomado partes de los diferentes trabajos entregados para complementar la información a la que podréis acceder durante los exámenes.

Gracias a todos.

Para los ejemplos vamos a trabajar con la siguiente clase



A. Definición y creación

Una colección representa un grupo de objetos. Esto objetos son conocidos como elementos. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos. En Java, se emplea la interfaz genérica `Collection` para este propósito. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos 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.

Un HashMap básicamente designa claves únicas para los valores correspondientes que se pueden recuperar en cualquier punto dado, es decir, nos permite almacenar elementos asociando a cada clave un valor.

Para cada clave tenemos un valor asociado. Podemos después buscar fácilmente un valor para una determinada clave. Las claves en el diccionario no pueden repetirse.

A.1. Constructores de HashMap

HashMap proporciona 4 constructores que definen la capacidad inicial de la colección y en qué momento debe redimensionarse. Son parámetros para mejorar el rendimiento en el uso del HashMap.

```java
// Crea una instancia de HashMap con una capacidad inicial de 16 y un factor de carga de 0,75.
HashMap<Integer, String> hm1 = new HashMap<>();

// HashMap(int initialCapacity). Crea una instancia de HashMap con una capacidad inicial especificada y un factor de carga de
HashMap<Integer, String> hm1 = new HashMap<>(10);

// HashMap(int initialCapacity, float loadFactor). Crea una instancia de HashMap con una capacidad inicial especificada y un
HashMap<Integer, String> hm1 = new HashMap<>(5, 0.75f);

//HashMap(Mapa de mapas) . Crea una instancia de HashMap con las mismas asignaciones que el mapa especificado.
HashMap<Integer, String> hm1 = new HashMap<>();
```

B. Métodos y propiedades generales

Partiendo de una serie de objetos, vamos a ver el resultado que obtendríamos con la ejecución de estos métodos

```java
Persona p1 = new Persona("Manuel", "García", 44, 1.74d, 80, "Hombre");
Persona p2 = new Persona("Juan", "Martínez", 65, 1.84d, 82, "Hombre");
```

B.1. Creación de un HashMap

```java
// Crear un HashMap con claves de tipo String y valores de tipo Persona
HashMap<String,Persona> DNIs = new HashMap<>();
```

B.2. Añadir y eliminar elementos

```java
// Element.put(k,v) - Añade un par clave-valor al mapa.
DNIs.put("390543M",p);
// Element.remove(Object key) - Removes the key and his value.
DNIs.remove("298423Z");// Will remove (“390543M”, p1) from the map.
```

> **Clave ya existente**
>
> Si ya existe un elemento con la misma clave en el mapa, el método put reemplaza el valor existente por el nuevo. Podemos evitarlo comprobando previamente si ya existe esa clave o con el método **DNIs.putIfAbsent(k,v)**.

B.3. Comprobar si una clave o un valor existen

```java
// Element.containsKey(Key) - Comprueba si existe la clave dada
DNIs.containsKey("390543M");
// Element.containsValue(Value) - Comprueba si existe el valor dado asociado a alguna clave
DNIs.containsValue(p2);
```

B.4. Acceder a las partes del mapa

```java
// Element.keySet() - Obtiene el conjunto de claves del mapa
Set<String> claves = DNIs.keySet();
// Element.values() - Obtiene el conjunto de valores almacenados en el mapa
Collection<Persona> valores = DNIs.values();
// Element.entrySet() - Obtiene el conjunto de pares clave-valor del mapa
Set<Entry<String,Persona>> tuplas = DNIs.entrySet();
```

B.5. Acceder a un elemento del mapa

```java
// Element.get(Key) - Obtiene el valor asociado a la clave
Persona p = DNIs.get("390543")
```

B.6. Otras funciones de utilidad

```java
// Element.size() - Devuelve el número de elementos en el mapa
int size = DNIs.size();
//Element.clear() - Elimina todas las asignaciones y vacía el mapa
DNIs.clear();
```

C. Añadir datos a un HashMap

> **Orden de los elementos en el HashMap**
>
> Cuando añadimos elementos a una HashMap, el orden de inserción no se conserva. Internamente, para cada elemento, se genera un hash separado y los elementos se indexan en función de este hash para hacerlo más eficiente. Antes de añadir un elemento, como se ha avisado antes, es recomendable comprobar si ya existe para no sustituirlo

C.1. Añadir elementos desde el constructor

A la hora de crear el Hashmap, podemos añadirle datos, usando la sintaxis del doble corchete o bien con la construcción Map.of

```java
// Crea un nuevo mapa y a la vez lo inicializa con valores
HashMap<String, Persona> map1 = new HashMap<>() {{
  put("390543M", p1);
  put("298423Z", p2);
}};
// En este caso indicamos los pares clave valor como si de un array se tratase
// De esta forma podemos añadir hasta un máximo de 10 elementos
HashMap<String, Persona> map2 = new HashMap<>(
  Map.of("390543M", p1, "298423Z", p2)
);
```

C.2. Añadir elementos desde otras colecciones

Al ser una colección compuesta por una clave y un valor, la inicialización se limita a los tipos de colecciones que tienen una estructura similar.

```java
// Partiendo del código anterior, creamos un nuevo mapa a partir de map2
HashMap<String, Persona> map3 = new HashMap<>(map2);
// O bien copiando todos los pares clave-valor
map3.putAll(map2);
```

C.3. Añadir / eliminar elementos desde código

```java
// Añadir y si existe la clave reemplazarlo
DNIs.put("390543M", p1);
// Añadir sólo si la clave no existe
DNIs.putIfAbsent("390543M", p1);
// Eliminar un elemento. Si la clave existe devuelve el valor asociado, si no devuelve null
Persona eliminada = map3.remove("390543M");
// Eliminar un par clave-valor. Si existe la tupla, la elimina y devuelve true, si no devuelve false
boolean existe = map3.remove("390543M", p1);
```

D. Recorrer la colección

Vamos a preparar un HashMap para recorrerlo y usarlo en los siguientes apartados

```java
// HashMap creation
HashMap<String, Persona> grupoPersonas = new HashMap<>() {{
        put("12345678A", new Persona("Nombre1", "Apellido1", 35, 1.66d, 71, "Mujer"));
        put("23456789B", new Persona("Nombre2", "Apellido2", 40, 1.84d, 88, "Mujer"));
        put("34567890C", new Persona("Nombre3", "Apellido3", 52, 1.70d, 66, "Hombre"));
        put("45678901D", new Persona("Nombre4", "Apellido4", 23, 1.96d, 98, "Mujer"));
        put("56789012E", new Persona("Nombre5", "Apellido5", 16, 1.55d, 60, "Hombre"));
        put("67890123F", new Persona("Nombre6", "Apellido6", 20, 1.75d, 74, "Hombre"));
}};
```

Tipos de valores

Vamos a mostrar ejemplos de cómo recorrer cada uno de los elementos que componen el HashMap, claves, valores y pares clave-valor.

.entrySet() - devuelve el conjunto de pares clave-valor

.keySet() - devuelve el conjunto de claves

.values() - devuelve la colección de valores

D.1. Usando un bucle for

Con el bucle for iteramos de forma natural accediendo a los elementos por índice, por lo que vamos a necesitar una forma de obtener el índice (i) de cada elemento para recorrer el HashMap.

En este caso, vamos a usar el índice de la clave, a través del método toArray.

Orden de los elementos en el HashMap

Es importante recordar y tener en cuenta que los elementos en un HashMap se ordenan automáticamente en base a una función Hash (resumen) que permite realizar una búsqueda muy rápida sobre la clave. Por lo tanto, no podemos esperar que el recorrido por índice coincida con el orden en el que los elementos se añaden al mapa.

```java
for (int i = 0; i < grupoPersonas.size(); i++) {
  System.out.println(grupoPersonas.get(map.keySet().toArray()[i]));
}
```

D.2. Usando un bucle foreach de Java

Otra forma de recorrer el HashMap es con un bucle similar al foreach de C#, aunque con el formato de un bucle for, pero en este caso indicando for(elemento : colección)

```java
// Recorremos la estructura obteniendo la tupla (k,v) en cada iteración
for (Map.Entry<String, Persona> entry : grupoPersonas.entrySet()) {
      System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
}
```

o bien para ir mostrando las claves

```java
for (String key : grupoPersonas.keySet()) {
  System.out.println("DNI = " + key);
}
```

o los valores

```java
for (Person value : grupoPersonas.values()) {
  System.out.println("Value = " + value);
```

D.3. Usando Iterator

El interface Iterator de Java permite movernos por una colección y acceder a sus elementos

java.lang.Iterator

Todas las colecciones de Java incluyen un método iterator() que devuelve una instancia de Iterator para recorrer la colección.

Iterator tiene 4 métodos:

- **hasNext()** - devuelve true si hay un elemento más en la lista
- **next()** - devuelve el siguiente elemento de la lista
- **remove()** - elimina el último elemento de la lista que hemos obtenido con next()
- **forEachRemaining()** - realiza la acción indicada con cada uno de los elementos que quedan por recorrer de la lista

Vamos a ver un ejemplo con los valores de la colección

```java
Iterator<Persona> iterator = grupoPersonas.values().iterator();
while (iterator.hasNext()) {
  System.out.println(iterator.next());
}
```

D.4. Usando el método forEach con expresiones lambda

En este caso aprovechamos el método foreach de las colecciones para poder realizar una acción concreta sobre cada uno de los elementos de la misma

```java
// Pares (clave,valor)
grupoPersonas.forEach((k,v) -> System.out.println("Clave: " + k + ", Values: " + v));
// Claves
grupoPersonas.keySet().forEach(k -> System.out.println("Clave: " + k));
```

```
// Valores
grupoPersonas.values().forEach(v -> System.out.println("Values: " + v));
```

D.5 Eliminando / Modificando elementos mientras se itera sobre la colección

Mientras se está recorriendo una colección, no con todos los tipos de bucles se puede modificar (añadir/eliminar elementos) de la colección. Vamos a ver el comportamiento de cada uno de ellos.

D.5.1 Con un bucle for

En este caso no tendríamos problemas. Al acceder por índice, podemos añadir o eliminar elementos mientras se recorre la colección.

java
```
for (int i = 0; i < grupoPersonas.size(); i++) {
  if(grupoPersonas.keySet().toArray()[i].equals("23456789B")) {
    grupoPersonas.remove(grupoPersonas.entrySet().toArray()[i]);
  }
}
```

D.5.2 Con un bucle foreach de Java

No modificable mientras se recorre

Si intentamos eliminar un elemento mientras lo recorremos con foreach, provocaremos una `java.util.ConcurrentModificationException`. Por lo tanto, sólo debemos usar este bucle si queremos leer sus elementos sin modificar la estructura de la colección.

java
```
for (Map.Entry<String, Persona> p : grupoPersonas.entrySet()) {
  if (p.getKey().equals("34567890C")) {
    personas.remove(p.getKey());
  }
  System.out.println("Clave: " + p.getKey() + " Valor: " + p.getValue().toString());
}
```

D.5.3 Con Iterator

Si usamos el método remove para eliminar elementos de la colección mientras la recorremos, podremos hacerlo sin que se genere ninguna excepción.

java
```
Iterator<Entry<String, Persona>> iterator = grupoPersonas.entrySet().iterator();
while (iterator .hasNext()) {
  Map.Entry<String, Persona> p = (Map.Entry<String, Persona>) iterator.next();
  if (p.getKey().equals("12345678A")) {
    // Si borramos usando iterator.remove o el método remove(key) de HashMap, funciona
    iterator.remove();
  }
}
```

D.5.4 Con el método forEach y expresiones lambda

> **No modificable mientras se recorre**
>
> Si intentamos eliminar un elemento mientras lo recorremos con foreach, provocaremos una `java.util.ConcurrentModificationException`. Por lo tanto, sólo debemos usar este bucle si queremos leer sus elementos sin modificar la estructura de la colección.

```java
// Elimina si encuentra un elemento concreto
grupoPersonas.keySet().forEach((k) -> {if (k.equals("34567890C")) grupoPersonas.remove(k);});
```

Otra cosa es que intentemos hacer cambios en los valores de la colección, por ejemplo, intercambiar los apellidos

```java
// Intercambia los apellidos de todas las personas
grupoPersonas.values().forEach((p) -> {
    String aux = p.getApellidos();
    p.setApellidos(p.getNombre());
    p.setNombre(aux);
  }
);
```

E. Búsqueda de elementos

Para buscar elementos en un HashMap hay distintas formas de hacerlo. Desde los propios métodos que nos ofrece la clase hasta el uso de API Stream. Vamos a describir cada uno de ellos

E.1. Búsqueda por clave o usando los métodos de la clase

La clase HashMap nos ofrece diferentes alternativas para buscar y/o saber si un elemento está presente en la colección. Así, podemos usar los métodos

```java
// A partir de una clave, obtener el valor
grupoPersonas.get("34567890C");
// O simplemente comprobar si existe esa clave o ese valor antes de buscarlo
grupoPersonas.containsKey("34567890C");
grupoPersonas.containsValue(person);
```

E.2. Búsqueda por el valor de una propiedad

A diferencia del caso anterior, si queremos buscar un objeto que contenga un valor concreto en un campo, debemos recorrer la colección hasta encontrarlo. Para eso, una de las alternativas es usar alguno de los bucles vistos anteriormente.

```java
// Puede haber más de un elemento que cumpla el criterio de búsqueda
Iterator<Map.Entry<String, Persona>> it = grupoPersonas.entrySet().iterator();
while (it.hasNext()) {
  Map.Entry<String, Persona> entry = it.next();
  if (((Persona)entry.getValue()).getNombre().equals("Jorge")) {
    // Se puede obtener el elemento o bien modificarlo
    System.out.println("Key: " + entry.getKey() + ", Value: " +
    entry.getValue());
  break;
  }
}
```

E.3. Búsqueda usando expresiones lambda

Mediante expresiones lambda, podemos incluir una condicional que nos haga el filtrado de elementos que deseemos

```java
// Puede haber más de un elemento que cumpla el criterio de búsqueda
grupoPersonas.forEach((k, v) -> {
  if (v.getNombre().equals("Jorge")) {
    // Se puede obtener el elemento o bien modificarlo
    System.out.println("Key: " + k + ", Value: " + v);
  }
});
```

E.4. Búsqueda usando API Stream

En este tipo de acciones es donde ya podemos empezar a ver la potencia que ofrece el API Stream para el manejo y gestión de las colecciones. Podemos emplear varios métodos, como filter, findAny, findFirst, allMatch, anyMatch, count, distinct. Como veremos en el siguiente apartado, esos resultados los podemos guardar en forma de subcolección

```java
// Puede haber más de un elemento que cumpla el criterio de búsqueda

// Obtener un submapa con los elementos que cumplan el criterio
grupoPersonas.entrySet().stream()
  .filter(k -> k.getKey().equals("abc"))
  .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

// O bien recorrer la lista de entradas obtenidas
for (Entry<String, Persona> p : grupoPersonas.entrySet().stream()
    .filter(k -> k.getKey().equals("abc"))
    .collect(Collectors.toList())) {
        System.out.println(p.getValue());   // Sacamos el valor de la tupla <clave,valor>
}

for (Persona p : grupoPersonas.entrySet().stream()
    .filter((k) -> k.getKey().equals("98761234D"))
    .map(Map.Entry::getValue)         // Cogemos sólo los valores del entryMap
    .collect(Collectors.toList())) {
        System.out.println(p);        // Muestra solo la persona
}

// Saber cuántos cumplen el criterio de búsqueda
grupoPersonas.entrySet().stream().filter(k -> k.getKey().equals("abc")).count();

// Obtener el primero que cumpla el criterio, si es que hay alguno
Optional<Entry<String,Persona>> s = grupoPersonas.entrySet().stream()
  .filter(k -> k.getKey().equals("6780123F"))
  .findFirst();
```

F. Obtención de subcolecciones

Lo podemos considerar un tipo especial de búsqueda en el que el objetivo es conseguir una colección con los elementos que cumplan un determinado criterio.

Así, la forma de buscar es idéntica a la del apartado anterior, pero en este caso lo que obtendremos de esa búsqueda será un nuevo tipo de colección, no necesariamente otro HashMap.

F.1. Subcolecciones usando bucles

```java
// Personas cuyo DNI acaba en "F"
HashMap<String,Persona> personas2 = new HashMap<String,Persona>();
for (Entry<String, Persona> e : grupoPersonas.entrySet()) {
  if(e.getKey().endsWith("F")) {
    personas2.put(e.getKey(), e.getValue());
  }
}
```

F.2. Subcolecciones usando expresiones lambda

En este ejemplo obtenemos un nuevo HashMap, pero también podríamos guardar la información en un ArrayList o en cualquier otro tipo de estructura.

```java
// Personas cuyo DNI acaba en "F"
HashMap<String, Persona> grupoPersonas2 = new HashMap<String, Persona>();
grupoPersonas.forEach((k, v) -> {
  if (k.endsWith("F")) {
    grupoPersonas2.put(k, v);
  }
});
```

F.3. Subcolecciones usando API Stream

Podemos obtener diferentes tipos de subcolecciones. Con API Stream podemos filtrar, hacer subconjuntos y guardar el resultado usando diferentes formas de .collect, que darán como resultados distintos tipos de colecciones.

```java
// HashMap de Personas cuyo DNI acaba en "F"
HashMap<String, Persona> grupoPersonas3 = (HashMap<String, Persona>) grupoPersonas.entrySet().stream()
    .filter(x -> x.getKey().endsWith("F"))
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

// Lista de Personas cuyo DNI acaba en "F"
List<Persona> listaPersonas3 = (List<Persona>) grupoPersonas.entrySet().stream()
    .filter(x -> x.getKey().endsWith("F"))
    .collect(Collectors.toList());
```

G. Ordenación de elementos

HashMap no garantiza el orden de los elementos

Como ya hemos comentado anteriormente, un HashMap es una estructura de datos en la que el orden de los elementos no está garantizado. Es por eso que si queremos mantener una copia ordenada de los elementos, deberemos recurrir a otros tipos de colecciones que sí garantizan el orden.

G.1. Ordenar por clave

Hay un tipo especial de map, TreeMap, en el que los elementos se ordenan siguiendo el orden natural de las claves. Por lo tanto es la opción ideal si queremos tener los elementos ordenados por clave.

```java
// Así podemos recorrer el TreeMap y mostrar los elementos ordenados por clave
TreeMap<String, Persona> grupoPersonasOrdenado = new TreeMap<>(grupoPersonas);

Iterator it=grupoPersonasOrdenado.keySet().iterator();
while(it.hasNext())
{
  int key=(int)itr.next();
  System.out.println("Key: "+key+" Element: "+hashMap.get(key));
}
```

G.2. Ordenar usando métodos de Collection

Si lo que queremos es tener el conjunto de claves o valores ordenados por separado, la forma más fácil es obtener una lista y ordenarla usando el método sort() de Collection.

```java
// Para las claves
List<String> clavesGrupoPersonas = new ArrayList<>(grupoPersonas.keySet());
Collections.sort(clavesGrupoPersonas);

// Para los valores
List<Persona> valoresGrupoPersonas = new ArrayList<>(grupoPersonas.values());
// Ordena según el método compareTo sobrescrito al implementar el interfaz Comparable
Collections.sort(valoresGrupoPersonas);
// Ordena por un campo cualquiera que indiquemos
Collections.sort(valoresGrupoPersonas, Comparator.comparing(Persona::getApellidos));
// Con Comparator tenemos disponibles varios comparadores (naturalOrder, reverseOrder, nullsFirst, ....)
```

Interfaz Comparable

Para que la primera forma de sort funcione, la clase Persona debe implementar el interfaz Comparable y sobrescribir su método `compareTo` para definir la forma de ordenar los objetos de tipo Persona.

Veremos más adelante que tenemos formas de definir el comparador usando expresiones lambda o APi Stream, permitiendo mayor flexibilidad a la hora de comparar elementos.

```java
// Un ejemplo, si queremos ordenar a las personas por edad
  @Override
  public int compareTo(Object o) {
      return ((Integer)this.getEdad()).compareTo((Integer)((Persona)o).getEdad());
  }
```

G.3. Ordenar con expresiones lambda

Si usamos una lista, no es necesario implementar el interfaz Comparable, ya que podemos indicar la comparación que queremos hacer como parámetro del método sort

```java
// Para los valores
List<Persona> valoresGrupoPersonas2 = new ArrayList<>(grupoPersonas.values());
// Indicamos la comparación que queremos hacer. Podemos usar compareTo o definir
// nuestro propio comparador
Collections.sort(valoresGrupoPersonas2, (e1, e2) -> ((Integer)e1.getEdad()).compareTo(e2.getEdad()));
System.out.println(valoresGrupoPersonas2);
```

G.4. Ordenar con API Stream

Con API Stream usamos también el método sorted para indicar qué comparación se debe realizar. Tenemos varias opciones en función del tipo de dato que pasemos.

Como en el caso anterior, el más flexible es aquel en el que indicamos, mediante una expresión lambda qué comparación realizar.

```java
// Guardamos el resultado en un LinkedHashMap que sí garantiza el orden
Map<String, Persona> sortedMap = grupoPersonas.entrySet().stream()
    // En este caso ordenamos por apellido, ascendente.
    .sorted((e1, e2) -> e1.getValue().getApellidos().compareTo(e2.getValue().getApellidos()))
    // Si queremos hacerlo descendente, ponemos .sorted((e2,e1) -> ......
    //.sorted((e2, e1) -> e1.getValue().getApellidos().compareTo(e2.getValue().getApellidos()))
    .collect(Collectors.toMap(Entry::getKey, Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));
```

Encadenar métodos

Al final, el uso de API Stream nos permite en una misma sentencia, buscar los elementos que queramos, ordenarlos y generar una subcolección con los resultados.

Es lo más parecido que vamos a encontrar a una consulta SQL para los datos de una colección cualquiera.

Aunque su sintaxis no es muy clara, si aprendemos a usarla correctamente, podremos realizar operaciones instantáneas, sin lugar a bugs, con muy poco código.