Collection

https://www.baeldung.com/java-collections

Collection

- **Collection** nos proporciona un conjunto de clases con las que podemos acceder a “listas” de datos. *Collection* es un interfaz implementado por una serie de clases que permiten la gestión de diferentes tipos de colecciones.
- Entre las más interfaces herederas de Collection más utilizadas están :
  - **List** : lista secuencial en la que cada elemento tiene una posición y se permiten elementos duplicados. **Queue:** lista FIFO (First in first out) en la que el primer elemento en introducirse es el primero en salir cuando se vayan extrayendo elementos, UnmodifiableList : Listas no modificables
  - **Set** : Utilizada para los tradicionales conjuntos matemáticos. Los elementos no tienen ningún orden y no puede haber elementos repetidos
  - **Map** : asocia claves a valores y los almacena de forma que el acceso por clave sea rápido.

- Cada una de esas interfaces tiene una clase abstracta asociada que contiene la funcionalidad común: AbstractList<E>, AbstractSet<E>, AbstractQueue<E>,..

List

- Entre las clases que implementan la interfaz List <E> y extienden AbstractList están:
  - ArrayList <E> . No sincronizada. Utilizada cuando no se utiliza acceso concurrente (threads)
  - Vector <E>. Sincronizada. Utilizada cuando se utiliza acceso concurrente . Peor rendimiento que ArrayList.

  Ambas :
  - Extienden de AbstractList<E>
  - Utilizan un objeto de tipo Array<E> , que tienen un tamaño fijo, aunque tanto Vector como ArrayList nos dan la posibilidad de ampliar, durante la ejecución y de forma transparente, el número de elementos almacenados.
  - Tienen una interfaz similar.

  Otras:
  - LinkedList: lista enlazada en la que cada elemento tiene un enlace con el siguiente. La reserva de memoria no tiene que ser contigua para todos los elementos.

Clase Vector: Utilidad

- Si vamos a utilizar un **array** pero **no conocemos el número de elementos** que va a tener podemos utilizar la clase Vector que, conceptualmente, es similar a un array, aunque su uso es diferente y permite que el número de casillas pueda aumentar durante la ejecución de la aplicación
- La jerarquía de clases de la clase Vector es la siguiente:
  - java.lang.Object
    - java.util.AbstractCollection<E>
      - Java.util.AbstractList<E>
        - » java.util.Vector<E>
- Por tanto para utilizarla necesitamos incluir :

  import java.util.Vector;

Clase Vector y genéricos

- La clase Vector (y las collection en general) utiliza genéricos a la hora de definirse.
  - Los **genéricos** son una característica de java por la que se permite que existan clases cuyo contenido no está completamente definido. Por ejemplo definir la clase Vector sin indicar si va a manejar objetos de la clase Empleado o Integer o Alumnos, etc.
  - Permite que, al igual que los métodos son parametrizables (admiten parámetros), las clases también lo sean.
  - Cuando se utilizan genéricos se debe colocar, en la definición de la clase, después del nombre de la misma, el nombre del parámetro entre < y >.

```
class Vector <T> {..}
```

Vector :Creación de referencias

- A la hora de **crear una referencia** a un objeto de la clase Vector (que utiliza genéricos) se puede poner la clase que actuará como parámetro:

  Vector <Alumno> vectorAlumnos;

  Vector <Integer> vectorEnteros;

  Vector vectorObjetos;

  Si no se pone ninguna clase se creará un vector de Object
- No se puede definir que el parámetro de la clase sea un tipo básico:

  Vector <int> vectorEnterosMal; //Error

Vector :Creación de objetos

- Después hay **que instanciar el vector** y se puede indicar **el tamaño inicial** del vector (colocándolo el valor entre paréntesis) a la hora de su creación:

  vectorAlumnos= new Vector <Alumno> (15) ;

  Con lo que se asignarán, inicialmente, 15 casillas para referencias a alumnos aunque posteriormente, si es necesario, se reservarán automáticamente más casillas si se necesitan. Si no se indica nada el valor por defecto son 10 casillas.

- También se puede indicar **el tamaño inicial** y el **incremento** de casillas que se realizará cada vez que necesitemos casillas adicionales:

  Vector <Alumno> vectorAlumnos=new Vector<> (15 , 3) ;

  Cuando necesitemos 16 casillas se reservarán 3 más (18 casillas)

  Si se ha indicado el tipo E en la declaración se puede omitir en la creación del objeto : Vector <Alumno>va = new Vector<>();

Clase Vector: Métodos

- **void add ([int index,] E element):** si no se especifica *index* añade un objeto en la última posición del vector. Si se indica *index* se añade el objeto en la posición *index* desplazando los demás hacia el final del vector para dejar hueco. Tiene que ser 0<=index<vector.size();
- **E elementAt (int index):** devuelve el objeto que se encuentra en la posición indicada por **index.** El tipo retornado puede ser Object o la clase utilizada en la declaración del vector.
- **void setElementAt (E element, int index):** coloca a *element* en la posición *index* sustituyendo al elemento que se encontraba en esa posición. index debe estar en el rango 0..size()-1
- **int indexOf (E element [,int index]):** devuelve la posición de la primera aparición de *element* en el vector. Si se especifica index busca a partir de esa posición (incluyendo la posición index). Utiliza el método **equals** (si está implementado) para hacer la comparación.
- **boolean isEmpty()**: devuelve true si el vector está vacío
- **int size ():** devuelve la cantidad de elementos del vector.
- **E remove (int index)** : Elimina el objeto de posición index y lo devuelve como valor de retorno. Desplaza los elementos hacia el principio del vector para rellenar el hueco. Retorna un objeto de la clase utilizada en la declaración o de tipo Object sino se utilizó ninguna.

El tipo de element puede ser Object o la clase utilizada en la declaración del vector.

Clase Vector: Ejemplo

```
import java.util.Vector;

public class VectorAlumnos {

   public static void main (String arg[]){
     Vector <Alumno> va;
     va=new Vector <Alumno>(2);
     Alumno a= new Alumno ("Juan", 11);
     va.add(a);
     va.add(new Alumno ("Ana",12));
     va.add(new Alumno ("Oscar",13));
     va.add(new Alumno ("María",14));
     for (int i=0; i<va.size(); i++)
        System.out.println (va.elementAt(i) );
     System.out.println ("FIN VECTOR Alumno");

     Vector vo;
     vo=new Vector (5,2);
     vo.add(a);
     vo.add(new Integer(33));
     vo.add(12);  //Se permite añadir el entero directamente
     for (int i=0; i<vo.size(); i++)
         System.out.println (vo.elementAt(i) );
   }
}
```

```
public class Alumno {
  String nombre;
  int edad;

  Alumno (String n, int e){
  edad=e;
  nombre=n;
  }
  public String toString () {
  return ( nombre + " " + String.valueOf(edad));
  }
}
```

Resultado de la Ejecución:

Juan 11
Ana 12
Oscar 13
María 14
FIN VECTOR Alumno
Juan 11
33
12

Clonación con Vector: Ejemplo

```
public class Clonacion {
public static void main(String[] args) throws CloneNotSupportedException
{
        int []aEnteros={1,3,6,2,6};
        Vector <Integer> vEnteros= new Vector<Integer>();
        for (Integer i:aEnteros)
          vEnteros.add(i);
        Vector <Integer> vCopia;
        vCopia= (Vector<Integer>) vEnteros.clone();
        vCopia.add(111);
        vEnteros.remove(new Integer(6));
        System.out.println("vEnteros");
        for (Integer i:vEnteros )
          System.out.print(i+",");
        System.out.println();
        System.out.println("vCopia");
        for (Integer i:vCopia )
          System.out.print(i+",");
        System.out.println();
}
}
```

Resultado de la Ejecución:

vEnteros

1,3,2,6

vCopia

1,3,6,2,6,111,

Método *T[] toArray(T[] arg);* *de* Vector

```
import java.util.Vector;

public class convertirVectoraArray {
public static void main(String[] args) {
//Creamos el array
Numero an1[]={new Numero(1),new Numero(2),
      new Numero(3),new Numero(4)};
Vector <Numero>vn1= new Vector<Numero>(3);
//Copiamos el  array en el vector
for (Numero n:an1)
   vn1.add(n);
for (Numero n:vn1)
   System.out.println(n);

Numero an2[]=new Numero[2];
//Copiamos el vector en el array 2
an2=vn1.toArray(an2);  //an2 no puede ser null
// También valdría :
//an2=vn1.toArray( new Numero[1] );
System.out.println("Array de vector an2:");
for (Numero n:an2)
   System.out.println(n);
}
vn1.forEach(System.out::println);}
```

```
public class Numero {
 int valor;
 Numero( int n){
   valor=n;
 }
@Override
public String toString() {
   return "Numero [valor=" + valor + "]";
}

}
```

Resultado de la Ejecución:

```
Numero [valor=1]
Numero [valor=2]
Numero [valor=3]
Numero [valor=4]
Array de vector an2:
Numero [valor=1]
Numero [valor=2]
Numero [valor=3]
Numero [valor=4]
```

Clase Vector: Ejercicio

1. Añade una instrucción al ejemplo de la transparencia anterior para recuperar en la variable
   Alumno alumnoDelVector;
   El elemento de posición 2 de va

2. Añade una instrucción al ejemplo de la transparencia anterior para recuperar en la variable
   Alumno alumnoDelVector;
   El elemento de posición 0 de vo. ¿Hay alguna diferencia con el ejercicio 1?

3. Añade instrucciones al ejemplo de la transparencia anterior para recuperar en una variable *valorDevuelto* el elemento de posición 1 de *vo* suponiendo que no sabes si en la posición 1 hay un *Integer* o un *Alumno* ¿Hay alguna diferencia con el ejercicio 2?

4. Añade instrucciones al ejemplo para que se muestre por pantalla la posición de va en la que está creado el alumno “Oscar” de 13 años utilizando el método *indexOf*

5. Un pila es una estructura de datos en la que el primer elemento que entra es el último en salir. Utilizando la clase Vector crea una class Pila con las operaciones insertar (mete un elemento en la pila) y recuperar (saca un elemento de la pila, que desaparece de la misma, y lo devuelve).

class ArrayDeque

- Inserción al principio y al final: Puede ser usada como cola o como pila.
- Sin restricciones de capacidad
- No thread safe
- No se permiten elementos nulos (null)

Constructores ArrayDeque

- ***ArrayDeque****()* Crea un ArrayD*eque* vacío con una capacidad para 16 elementos
- ***ArrayDeque****(Collection<? extends E> c)* Crea un ArrayD*eque* que contiene los elementos de la colección indicada en el orden devuelto por el iterador asociado a la colección
- ***ArrayDeque****(int numElements)* Crea un ArrayD*eque* vacío con una capacidad inicial suficiente para almacenar *numElements*

- Un iterador es un patrón que se utiliza para recorrer una estructura de datos

Métodos ArrayDeque

- void **addFirst**(E e) Inserta el elemento al principio de la deque
- void **addLast**(E e) Inserta el elemento al final de la deque
- void **clear**() Borra todos los elementos de la deque
- ArrayDeque<E> **clone**() Devuelve una copia de la *deque*.
- boolean **contains**(Object obj) Devuelve true si la *deque* contiene el elemento obj
- E **getFirst**() Recupera, sin eliminarlo de la cola, el primer elemento de la misma
- E **getLast**() Recupera, sin eliminarlo de la cola, el último elemento de la misma.
- boolean **isEmpty**() Devuelve true si la *deque* está vacía.
- E **removeFirst**() Recupera y elimina de la cola, el primer elemento de la misma.
- E **removeLast**() Recupera y elimina de la cola, el último elemento de la misma.
- boolean **removeFirstOccurrence**(Object obj) Elimina la primera ocurrencia del elemento obj indicado en la cola recorriéndola de principio (first element) a fin
- int **size**() Devuelve el número de elementos de la *deque*.
- Object[] **toArray**() Devuelve un array que contiene todos los elementos de la *deque* en el mismo orden

Ejemplo uso ArrayDeque con genéricos

```
import java.util.ArrayDeque;
public class Cola <Tipo> {  // FIFO (First Input First Output)
ArrayDeque < Tipo > cola;
Cola (){
  cola = new ArrayDeque<Tipo>( );
}
void insertar (Tipo o){
       cola.addLast( o);  //add = addFirst
}
Tipo extraer (){
      return cola.removeFirst();
}
public static void main(String[] args){
 Cola <String>c = new Cola<String>();
c.insertar("Uno");
Cola <Persona> p=new Cola<Persona>();

 c.insertar("Dos");
 c.insertar("tres");
 System.out.println(c.extraer() );
 c.insertar ("Cuatro");
 System.out.println(c.extraer() );
 System.out.println(c.extraer() );
 System.out.println(c.extraer() );
 System.out.println(c.extraer() );  //Generará una excepción java.util.NoSuchElementException
   }
}
```

class Collections

Collections (con s) es una clase que consta de métodos estáticos que permiten operar con objetos de la clase Collection

Entre los métodos más utilizados estaría el utilizado para ordenaciones.

static <T extends Comparable<? super T>> void **sort**(List<T> list) que ordena la lista en orden ascendente teniendo en cuenta el orden natural de sus elementos.

- Si lo utilizo con Vector la clase almacenada en cada casilla del Vector debe implementar el interface Comparable<T> y codificar el método compareTo(T o) que debe devolver un número entero negativo, cero o un número positivo dependiendo de si el objeto this es menor, igual o mayor que el recibido como parámetro. Existen otras restricciones de comportamiento del método que se pueden ver en la definición del mismo

- Existen otras implementaciones como:
  - public static <T> void sort(List<T> list, Comparator<? super T> c)
  - Que permitirán ordenaciones por varios campos al no tener que codificar el método de comparación en la clase del objeto contenido en la colección sino en una clase cualquiera.

Collections: Ejemplo ordenación con Comparable

```
import java.util.Vector;
import java.util.Collections;

public class VectorAlumnos {
Static void mostrar(Vector va){
   for (int i=0; i<va.size(); i++)
      System.out.println (va.elementAt(i) );
   }
   public static void main (String arg[]){
    Vector <Alumno> va;
    va=new Vector <Alumno>(2);
    va.add(new Alumno ("Juan", 15));
    va.add(new Alumno ("Ana",12));
    va.add(new Alumno ("Juan",13));
    va.add(new Alumno ("María",14));
    VectorAlumnos.mostrar(va);
    System.out.println ("FIN VECTOR Alumno");
    Collections.sort(va);
    VectorAlumnos.mostrar(va);}
}
```

```
public class Alumno implements Comparable<Alumno> {
  String nombre;
  int edad;

  Alumno (String n, int e){
  edad=e;
  nombre=n;
  }
  public String toString () {
  return ( nombre + " " + String.valueOf(edad));
  }
  public int compareTo(Alumno a) {
   //Ordeno primero por nombre y luego por edad
    int compEdades=edad- a.edad;
    int compNombres = nombre.compareTo(a.nombre);
    return compEdades+ compNombres *1000;
  }
}
```

Resultado de la Ejecución:

Juan 15
Ana 12
Juan 13
María 14
FIN VECTOR Alumno
Ana 12
Juan 13
Juan 15
Maria14

Comparator: Ejemplo ordenación

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
//Ejemplo Comparator. También se podría hacer con una clase anónima definida el usar el sort.
public class CriterioComparacion implements Comparator<Persona> {
 @Override
 public int compare(Persona p1, Persona p2) {
  if (p1.getEdad() > p2.getEdad())
   return 1;
  else if (p1.getEdad() < p2.getEdad())
   return -1;
  else
   return 0;
 }
}
public class MainComparar {
 public static void main(String[] args) {
  List<Persona> vp = new ArrayList<Persona>();
  vp.add(new Persona("Ana", "Rodes",10));
  vp.add(new Persona("Hector", "Lopez",5));
  vp.add(new Persona("Ana", "Buigues",20));
  vp.add(new Persona("Carlitos", "Perez",15 ));
  Collections.sort(vp, new CriterioComparacion());
  for (Persona p : vp)
   System.out.println(p);
 }
}
```

```
public class Persona  {
 private String nombre;
 private String apellidos;
 private int edad;
 public Persona (String nombre, String apellidos, int e) {
  if (nombre == null || apellidos == null)
   throw new NullPointerException();
  this.nombre = nombre;
  this.apellidos = apellidos;
  edad=e;
 }
 @Override
 public String toString() {
  return String.format("%s, %s %d", nombre, apellidos, edad);
 }
public int getEdad() {
  return edad;
}
public void setEdad(int edad) {
  this.edad = edad;
}
}
```

Resultado de la Ejecución:

Hector, Lopez 5

Ana, Rodes 10

Carlitos, Perez 15

Ana, Buigues 20

Ordenación con Exp. Lambda

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
public class VectorAlumnos {
 public static void main (String arg[]){
  VectorAlumnos obj= new VectorAlumnos();
  Vector <Alumno> va;
  va=new Vector <Alumno>(2);

    Alumno a= new Alumno ("Juan", 13);
    va.add(a);
    va.add(new Alumno ("Ana",12));
    va.add(new Alumno ("Juan",11));
    va.add(new Alumno ("María",14));

    System.out.println("Ordeno por edad");
    //Convierto a Integer porque compareTo debe recibir
objetos.
    Collections.sort(va, (a1, a2) ->
(Integer.valueOf(a1.edad)).compareTo(Integer.valueOf(a2
.edad)));
    System.out.println(va);
 }
}
```

```
public class Alumno {
  String nombre;
  int edad;


  Alumno (String n, int e){
    edad=e;
    nombre=n;
  }
  public String toString () {
    return ( nombre + " " + String.valueOf(edad));
  }
}
```

Resultado de la Ejecución:

```
[Juan 11, Ana 12, Juan 13, María 14]
```

Hashtable: búsquedas rápidas

- Las Hashtable/Map proporcionan estructuras de almacenamiento de la información en memoria principal que se utiliza cuando se desea guardar, para posteriormente buscar de forma rápida, elementos que tienen asociada una clave.
- Se construyen en base a una pareja **Clave:Valor**.
  - La **clave** es el elemento identificativo por el que se buscará el dato. **No** puede estar **repetida** en la Hashtable.
  - El **valor** es el dato asociado a la clave.
- Las claves deben ser clases, al igual que los valores
- Las **claves** deben tener **codificado** el método **hashcode** (que devolverá un número entero) y el método **equals**. En otro caso se utilizarán los métodos hashcode y equals por defecto.
- Se utiliza el método *hashcode* como base para conocer en que posición se guardarán los elementos de una Hashtable y poder, cuando se indique esa clave, saber, utilizando el mismo método, en que posición se encuentra. De esa forma no hay que buscar en toda la estructura de almacenamiento el elemento a buscar.
- **Hashtable** es una clase sincronizada que no permite nulos ni en clave ni en valor. **HashMap** es similar pero no sincronizada y permite un nulo en clave y varios en valor. También tenemos **HashSet**
- **LinkedHashMap** permite orden predecible de iteracción (el de inserción)
- **Map** es el interfaz definido para el uso de pares clave-valor implementado por HashMap, Hashtable, LinkedHashMap, TreeMap, Properties, etc.

Hashtable: Constructor

– La forma básica de llamada al constructor es

Hashtable <TipoClave, TipoValor> ht= new Hashtable < TipoClave, TipoValor >();

Ejemplos:

Hashtable <String, Persona> *ht1= **new Hashtable <String, Persona>();***

Hashtable <**Integer, String**> *ht2= **new Hashtable <*** Integer, String ***>();***

Incorrectos:

static Hashtable <**int**, String> ht3= new Hashtable <**int**, String>();

static Hashtable <Integer, **int**> ht4= new Hashtable <Integer, **int**>();

– Aunque también existen otros constructores que permiten indicar el tamaño inicial de la tabla o cuando se debe incrementar ésta (porcentaje (entre 0 y 1) de cuando se considera lo suficientemente llena como para ampliarla)

static Hashtable <**Clave, Persona**> *ht3= new Hashtable <**Clave, Persona**>(**33**);*

static Hashtable <**Integer**, **Integer**> ht4= new Hashtable <Integer, Integer>(**33,(float)0.75**); //75% de sobrecarga máxima antes de ampliar.

Clase Hashtable: Métodos

- public V **put(K** key, V value): Asocia la clave al valor (ninguno de ellos puede ser nulo) y lo almacena en la Hashtable. Devuelve el valor anterior asociado a la clave o null si no existe. No puedes tener 2 valores asociados a la misma clave así que el segundo que insertes con una clave sustituirá al anterior con esa misma clave.
- public V **get(Object** key): Devuelve el elemento que tiene asociada esa clave o null si no existe.
- public V **remove(Object** key): Elimina la clave (y el valor asociado) de la Hashtable. Devuelve el elemento eliminado o null si no existe.
- public int **size**() : Devuelve el número de claves almacenadas en la Hashtable.
- public Collection<V> **values**() : Devuelve una colección con los valores almacenados en la Hashtable.
- También puedes **utilizar streams** a partir de la colección devuelta por values

Clase Hashtable: Ejemplo I

```
public class EjemHashTable {
static Hashtable <String, Persona> ht= new Hashtable <String, Persona>();
static Hashtable <Clave, Persona> ht2= new Hashtable <Clave, Persona>();
static Hashtable <Integer, Integer> ht4= new Hashtable <Integer, Integer>(33,(float)0.75);
static void crearHashTableEnteros(){
 ht4.put(3,30);
 ht4.put(1, 10);
 ht4.put(2, 2);
 System.out.println(ht4.size());
 for (int i=0; i<33; i++)
   ht4.put(i, i*10);
}
static void crearHashTableSimple(){
 ht.put("Juan",  new Persona ("Juan",1,100));
 ht.put("Ana",  new Persona ("Ana",2,200));
 ht.put("Jose",  new Persona ("Jose",4,400));
 Persona p=new Persona ("Oscar",3,300);
 ht.put(p.nombre,  p);
 Persona px= ht.get("Ana");
 System.out.println("nombre:"+px.nombre+"
edad:"+px.edad);
 }
```

```
public class Persona {
  String nombre;
  int edad;
  double sueldo;

public Persona(String nombre, int edad, double sueldo) {
  this.nombre = nombre;
  this.edad = edad;
  this.sueldo=sueldo;
}
}
```

Resultado de la Ejecución (1 y 2) :

3

nombre:Ana edad:2

```
public static void main(String[] args) {
   crearHashTableEnteros();
   crearHashTableSimple();

   listarHashTableSimple2();
   crearHashTable();
  añadirDuplicados();
}
```

Clase Hashtable: Ejemplo II

```
 static void listarHashTableSimple2(){
//No está ordenado por ningún criterio, tampoco por el orden de inserción
    Collection<Persona> listaPersonas;
    Persona p;
    listaPersonas = ht.values();
//ht.values().stream().forEach(System.out::println);
    Iterator<Persona> it=listaPersonas.iterator();
    while(it.hasNext()) {
       p =  it.next();
       System.out.println("Nombre " +p.nombre+ " Edad:"+ p.edad);
    }
  }
static void crearHashTable(){
  ht2.put( new Clave(19,"aa"),  new Persona ("Juan",1,100));
  ht2.put(new Clave(18,"b"),  new Persona ("Ana",2,200));
  Clave c=new Clave(17,"CCC");
  ht2.put(c,  new Persona ("Oscar",3,300));
  Clave c2=new Clave(17,"CCC");
  Persona p2= ht2.get(c2);    //devuelve Persona ("Oscar",3,300);
//no es válido poner ht2.get(new Clave(17)); porque referencia otra clave
//a no ser que para esa nueva clave equals y hashCode devuelvan lo mismo
  System.out.println(p2.edad);
 }
static void añadirDuplicados(){
  ht.put("Marta",  new Persona ("Marta",4,400));
  Persona px= ht.get("Marta");
  ht.put("Marta",  new Persona ("Marta",43,4300));
  px= ht.get("Marta");
  System.out.println("nombre:"+px.nombre+" edad:"+px.edad);
  px=ht.remove("Marta");
  System.out.println("eliminado nombre:"+px.nombre+" edad:"+px.edad);
  px= ht.get("Marta");
  if (px==null) System.out.println("Marta ya no está");
}
```

```
public class Clave {
  int codigo;
  String xxx;

public Clave(int codigo, String xxx) {
  this.codigo = codigo;
  this.xxx=xxx;
}
public int hashCode() {
//Si no se codifica hashCode se usa el por defecto
final int prime = 31;
int result = 1;
result = prime * result + codigo;
result = prime * result + ((xxx == null) ? 0 :
xxx.hashCode());
return result;
}
public boolean equals(Object obj) {
//Si no se codifica se usa el por defecto
  …
}
}
```

Resultado de la Ejecución (3,4y 5):

Nombre Jose Edad:4

Nombre Ana Edad:2

Nombre Juan Edad:1

Nombre Oscar Edad:3

3

nombre:Marta edad:43

eliminado nombre:Marta edad:43

Marta ya no está

Clase Properties: Map en ficheros

- La clase Properties nos permite gestionar el almacenamiento en fichero(stream) de estructuras de tipo Map. Pertenece al paquete java.util.
- Tanto la clave como el valor son String
- Es una clase Thread-safe.
- Hereda de Hashtable.

- `Properties p = new Properties();` constructor.
- **`p.setProperty(String clave ;String Objeto);`** Añade el par clave-Objeto a p.
- **`p.store(OutputStream fo, String coments);`** Añade al stream/fichero fo los comentarios (al inicio) y el contenido de p. coments se puede poner a null
- **`public String getProperty(String key, String defaultValue):`** devuelve el valor asociado a Key o defaultValue si no se encuentra.
- **`public Set<String> stringPropertyNames():`** Devuelve el conjunto de claves no modificable asociados al objeto Property.

Fichero properties

- En proyectos Maven se suele utilizar un fichero **config.properties,** que se puede colocar en el directorio src/main/resources para definir propiedades de mi proyecto.
- Para cargar el fichero de propiedades cuando se tiene la aplicación comprimida en un jar, se puede utilizar la forma

Properties p= new Properties();

p.load(miObj.getClass().getClassLoader().getResourceAsStream("config.properties");

Donde miObj es un objeto de la clase en la que está el main de mi aplicación.

- A veces se utiliza también el config.yaml (config.yml) colocado en el mismo directorio porque yaml porque permite subapartados, mientras que el config.properties no permite más que clave:valor y no permite listas ni estructuras subordinadas

Clase Hashtable: Ejemplo II

```
public class HashEnFichero {
static void escritura(){
try (OutputStream fo= new
FileOutputStream("fich.dat")){
 Properties p = new Properties();
 p.setProperty("clave1A","Juan");
 p.setProperty("clave2B","Pepe");
 p.setProperty("clave3C","Ana");
 p.store(fo, "escritura Realizada");
}catch (Exception e){
System.out.println("Error");
}
}
static void añadir(){
try (OutputStream fo= new FileOutputStream
    ("fich.dat", true)){
 Properties p = new Properties();
 p.setProperty("clave4D","Oscar" );
 p.setProperty("clave5E","Marta");
 p.store(fo, null);
}catch (Exception e) {
System.out.println("Error");
}
}
}
```

```
static void lecturaClaves(){
  try (InputStream fi= new
FileInputStream("fich.dat")){
   Properties p = new Properties();
   p.load(fi);
   String clavePepe=p.getProperty("Pepe");
   System.out.println("Clave de
Pepe:"+clavePepe);
   String claveNoExiste=p.getProperty("No
existe"); //null
   System.out.println("Clave de No
existe:"+claveNoExiste);
  System.out.println("TODOS los usuarios.");
  for (String s : p.stringPropertyNames())
    System.out.println(s);
 }catch (Exception e){
  System.out.println("Error");
}

public static void main(String[] args) {
 escritura();
 añadir();
 lecturaClaves();
}
```

Array de vectores

```
public class ArrayVectores {
  public static void main(String[] args) {
    Vector <Integer> pract= new Vector<Integer>(20);
    pract.add(10);
    Vector <Integer> vp[]= new Vector [2];
    vp[0]= pract;
    vp[1]= new Vector<Integer>(10);
    vp[1].add(33);
    vp[0].add(22);
    for (int i=0; i<vp.length; i++){
      for (Integer n:vp[i])
        System.out.print(n+",");
      System.out.println();
    }
  }
}
```

Resultado de la Ejecución :

```
10,22,
33,
```

Vector de vectores

```
public class VectorDeVectores {
static void mostrar(Vector<Vector<Integer>> vv){
  for (int i=0; i<vv.size(); i++){
    for (Integer n:vv.elementAt(i))
      System.out.print(n+",");
    System.out.println();
  }
}
public static void main(String[] args) {
  int numVectores=3;
  Vector<Vector<Integer>> vv;
  vv = new Vector<Vector<Integer>>(numVectores);
  for (int i=0; i<numVectores; i++)
    vv.add (new Vector<Integer>());
  añadir(vv);
  mostrar(vv);
}
}
```

```
static void añadir
(Vector<Vector<Integer>> vv){
  vv.elementAt(0).add(11);
  vv.elementAt(0).add(22);
  vv.elementAt(1).add(33);
  vv.elementAt(2).add(44);
  vv.elementAt(2).add(55);
}
```

Resultado de la Ejecución :

```
11,22,
33,
44,55,
```

Collectors

New Stream Collectors in Java 9 | Baeldung

Clase Collectors

- Implementan operaciones de reducción, como acumular elementos en colecciones, resumir elementos según varios criterios, etc.
- List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
- Set<String> set = people.stream().map(Person::getName).collect(Collectors.toCollection(TreeSet::new));
- String joined = things.stream().map(Object::toString).collect(Collectors.joining(", "));
- int total = employees.stream().collect(Collectors.summingInt(Employee::getSalary));
- Map<Department, List<Employee>> byDept = employees.stream()
- .collect(Collectors.groupingBy(Employee::getDepartment));
- Map<Department, Integer> totalByDept = employees.stream().collect(Collectors.groupingBy(Employee::getDepartment,
- Collectors.summingInt(Employee::getSalary)));
- Map<Boolean, List<Student>> passingFailing = students.stream()
- .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));