Universidad del Valle

Fundamentos de Programación Orientada a Eventos

Luis Yovany Romo Portilla, MsC.

Jueves | Edf. B-13 -> SALA 4 -- MG -- MELENDEZ

Universidad del Valle

Fundamentos de Programación Orientada a Eventos

Colecciones

Colecciones

- Una colección es un objeto que permite agrupar otros objetos.

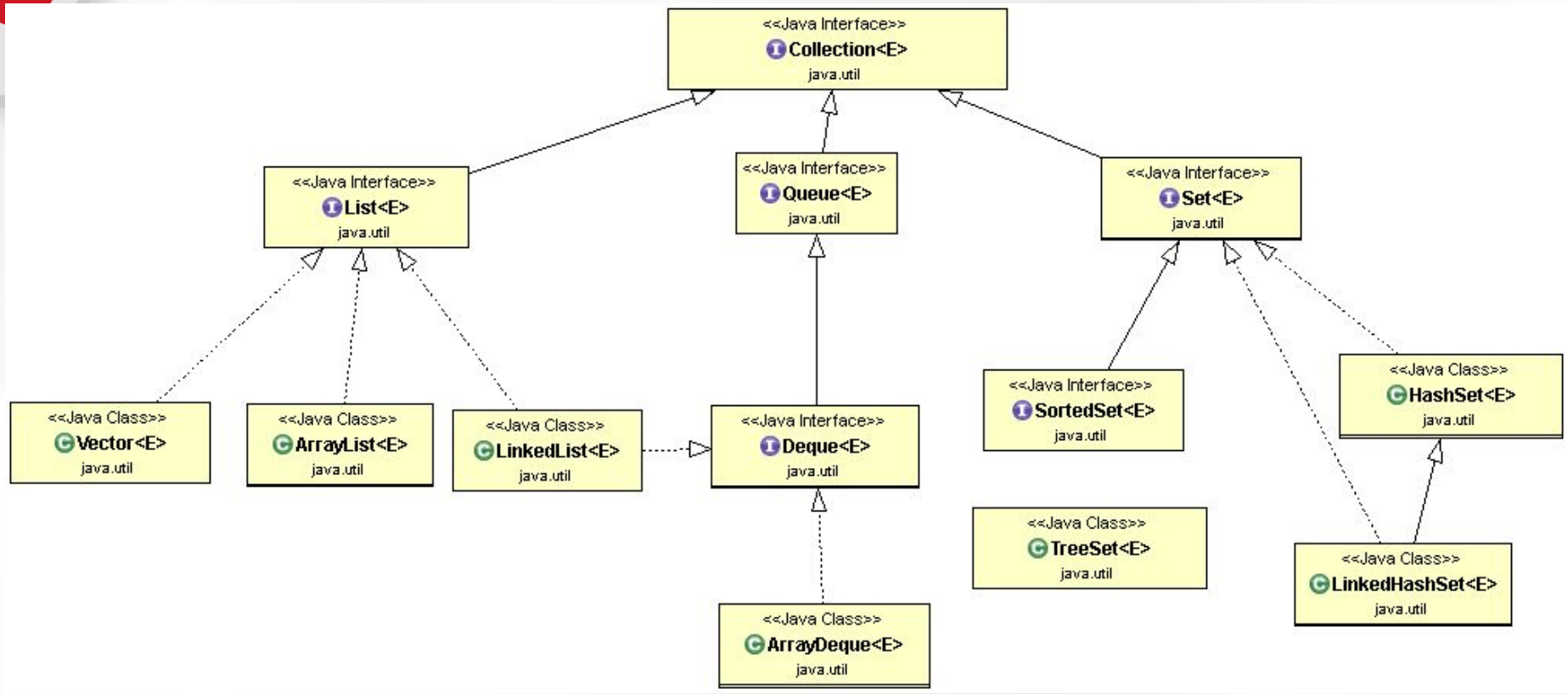
- Generalmente, los elementos en una colección representan datos de una agrupación específica de objetos, como una colección de personas, casas.

- En una colección se pueden realizar operaciones sobre los objetos que están almacenados en su interior, así podemos almacenar, manipular, obtener y comunicar entre estos objetos.

Arreglos y colecciones

Arreglos	Colecciones
Son de tamaño fijo	Son de tamaño dinámico
Almacenan datos del mismo tipo	Si se declara como genérico, se pueden almacenar información de diferentes tipos de datos
Almacenan datos primitivos / Objetos	Almacenan Objetos.
Para acceder a los datos, solo se necesita la posición	El casting es necesario para acceder a los objetos de la colección si se declaran como genérico

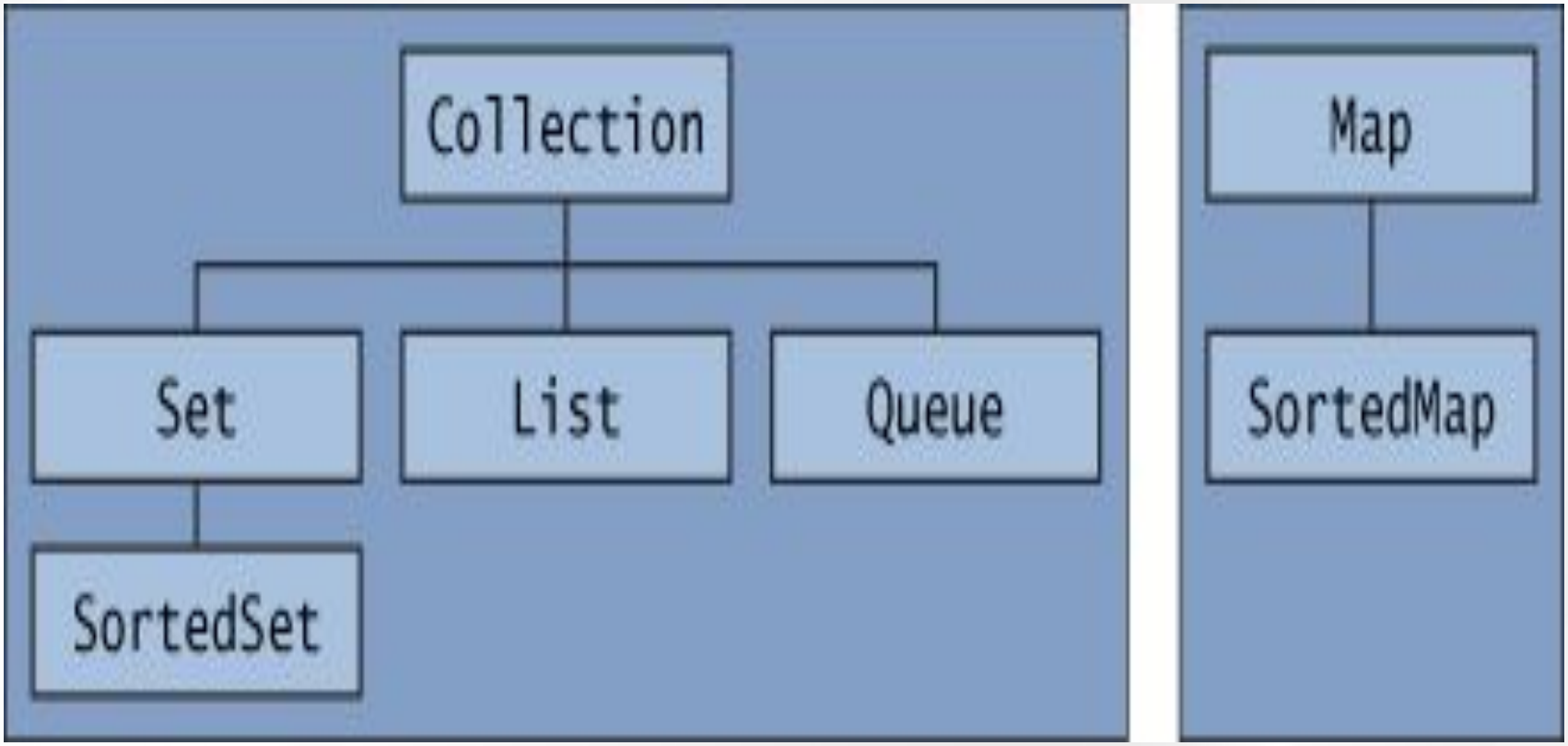
Jerarquía de las Colecciones



Colecciones en java

- Permiten almacenar y organizar objetos de manera útil para un acceso eficiente.
- Se encuentran en el paquete java.util
- Núcleo de abstracciones de colecciones de utilidad (interfaces) e implementaciones ampliamente útiles.
- Las interfaces proporcionan métodos para todas las operaciones comunes y las implementaciones concretas especifican la decisión de las operaciones no permitidas.

Interfaces



colecciones

- **Collection:** Es la raíz de la jerarquía. Representa un grupo de objetos. Esta interfaz es el último denominador común que todas las colecciones y es usada cuando se desea manipularlas con el máximo de generalidad deseado.

- **Set**: Una colección que no puede tener elementos duplicados. Un ejemplo son las cartas de un naipe.

- **List:** Una colección ordenada. Las listas pueden contener elementos duplicados. Generalmente se puede tener control sobre donde cada elemento es insertado y se puede acceder a cada elemento por su índice (posición).

colecciones

- **Queue**: (Cola) colección utilizada para guardar varios elementos por prioridad. Provee operaciones adicionales de inserción, extracción e inspección..

- **Map**: Un objeto que trabaja por parejas siendo sus componentes clave y valor. Map no permite tener claves duplicadas, cada clave debe corresponder al menos a un valor.

- **SortedSet:** Mantiene sus elementos en un orden ascendente.

- **SortedMap**: Mantiene sus mapeos en un orden ascendente por clave.

Colecciones java

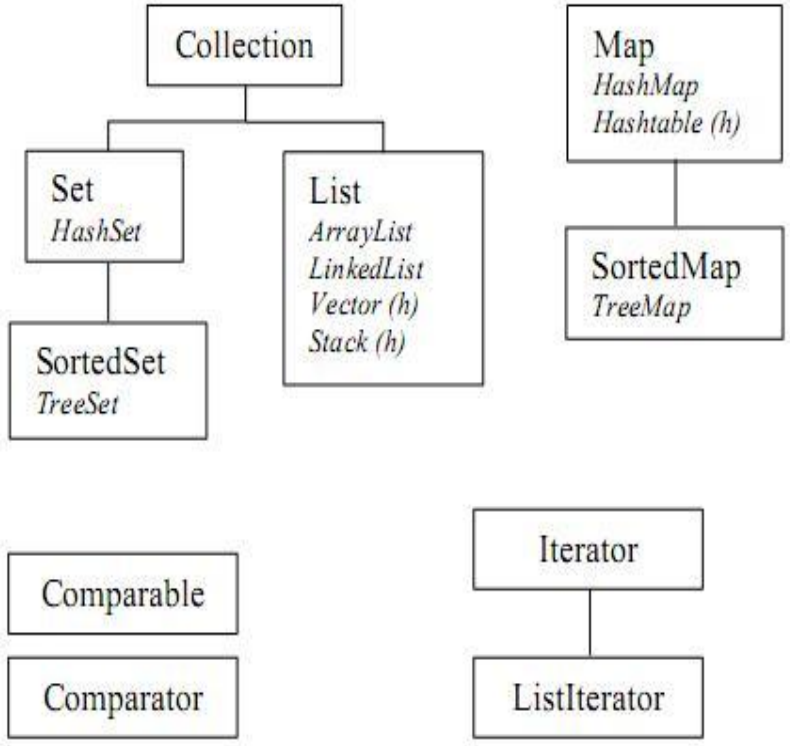


Figura 4.1. Interfaces de la Collection Framework.

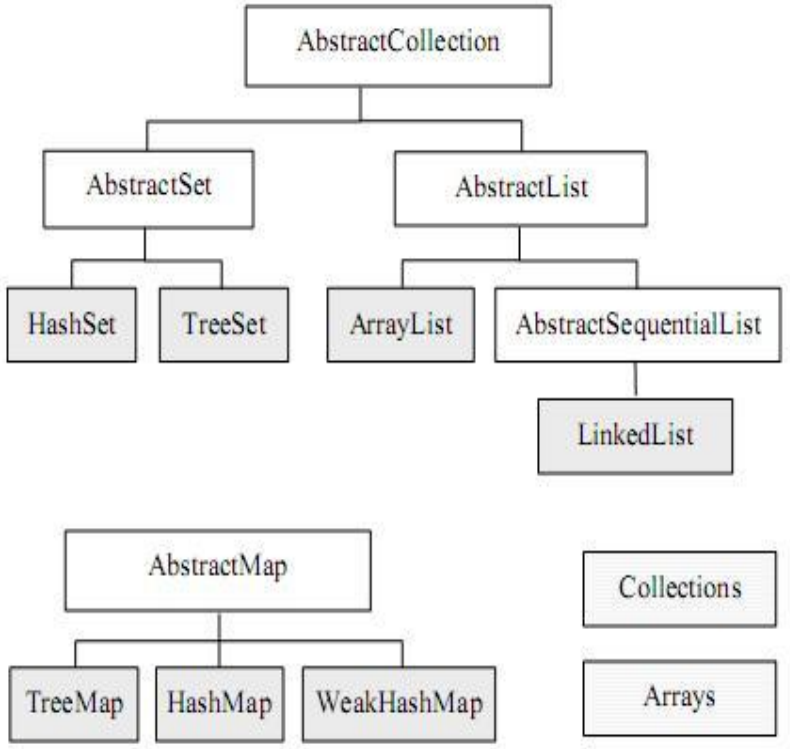


Figura 4.2. Jerarquía de clases de la Collection Framework.

Interfaz Collection

- **size():** permite saber cuántos elementos existen en la colección.
- **isEmpty()**: verifica si la colección se encuentra vacía.
- **contains()**: chequea si un elemento específico se encuentra en la colección.
- **add()**: para agregar un elemento a la colección.
- **remove()**: para eliminar un elemento de la colección.
- **iterator()**: para iterar sobre la colección.

Interface Collection

```
public interface  Collection  <E> extends  Iterable<E>{

    int size ();
    boolean isEmpty ();
    boolean contains(Object element);
    boolean add (E element);
    boolean remove (Object element);
    iterator <E> iterator();
}
```

Interface Iterator

next (): Permite acceder al siguiente elemento de una colección.

hasNext(): Evalúa si hay más elementos en una colección.

```
public interface Iterator<E> {

    boolean hasNext();
    E next();
    void remove(); //optional

}
```

Interface List

Define colecciones secuenciales:

Acceso por posición	**get(int)** **set(int, Object)** **add(int, Object)**
Búsquedas	indexOf(Object) lastIndexOf(Object)
Recorridos con iteradores	listIterator() listIterator(int)

Implementaciones de List

- **ArrayList**: Como los arrays pero con redimensión automática.
- **Vector**: Como ArrayList pero que permite acceder de forma fiable desde varios hilos de ejecución.
- **LinkedList**: Listas doblemente enlazadas con inserción por delante y por detrás (se puede usar a modo de cola).
- **Stack**: Pila implementada usando internamente la clase Vector

Implementaciones De List

- import java.util.ArrayList
- import java.util.LinkedList
- import java.util.Vector
- import java.util.Stack

```
List listaA = new ArrayList();
List listaB = new LinkedList();
List listaC = new Vector();
List listaD = new Stack();
```

Implementaciones de Collection

LinkedList

Una implementación de una lista doblemente enlazada. La modificación es poco costosa para cualquier tamaño, pero el acceso aleatorio es lento. Útil para implementar colas y pilas.

- getFirst, getLast, removeFirst, removeLast, addFirst, addLast

ArrayList

Una lista implementada utilizando un array de dimensión modificable. Es costoso añadir o borrar un elemento cerca del principio de la lista si ésta es grande, pero es relativamente poco costoso de crear y rápido para acceso aleatorio.

Interfaz list

Método	Descripción
add	Inserta un elemento al final de la lista
clear	Borra todos los elementos de una lista
contains	Retorna true si la lista contiene un elemento especificado o falso en caso contrario.
get	Retorna el elemento de una posición específica
indexOf	Retorna el índice de un objeto especificado
remove	Elimina un elemento de la lista
size	Retorna el número de elementos almacenados en la lista

Agregar elementos a una lista

Para agregar elementos a una lista se llama el método **add().** Este método es heredado de la interfaz Collection

```
ArrayList listaA = new ArrayList();

listaA.add("elemento 1");
listaA.add("elemento 2");
listaA.add("elemento 3");
```

Obtener elementos a una lista

1. Usando el índice

```
String elemento0 = listaA.get(0);
String elemento1 = listaA.get(1);
String elemento3 = listaA.get(2);
```

Obtener elementos de una lista

2. Usando un iterador

```
Iterator iterador =  listaA.iterator;

While (iterator.hasNext(){

    String elemento= (String)iterador.next();

}
```

Obtener elementos a una lista

3. Usando un ciclo for

```
for (Object ob : listaA){

   String  e = (String) ob;
}
```

Listas genéricas

Por defecto en una lista se puede insertar cualquier objeto, pero también se puede indicar el tipo de objeto que se quiere almacenar.

```
ArrayList <MyObject> lista = new ArrayList <MyObject>();
```

Ejemplo Arraylist

```
public class  Empleado {
     String id;
     String nombre;
     String cargo;
     int salario;

}
```

Ejemplo Arraylist

```
ArrayList<Empleado> lista = new
ArrayList<Empleado>();

Empleado emp1= new Empleado();
```

Ejemplo Arraylist

```
ArrayList<Empleado> lista = new ArrayList <Empleado>();

Empleado emp1= new Empleado();

emp1.setId("e1");
emp1.setNombre("Pedro Perez");
emp1.setCargo("Operador");
emp1.setSalario(1000);
```

Ejemplo Arraylist

```
Empleado emp2= new Empleado();

emp2.setId("e2");
emp2.setNombre("Juan Torres");
emp2.setCargo("Auxiliar");
emp2.setSalario(500);
```

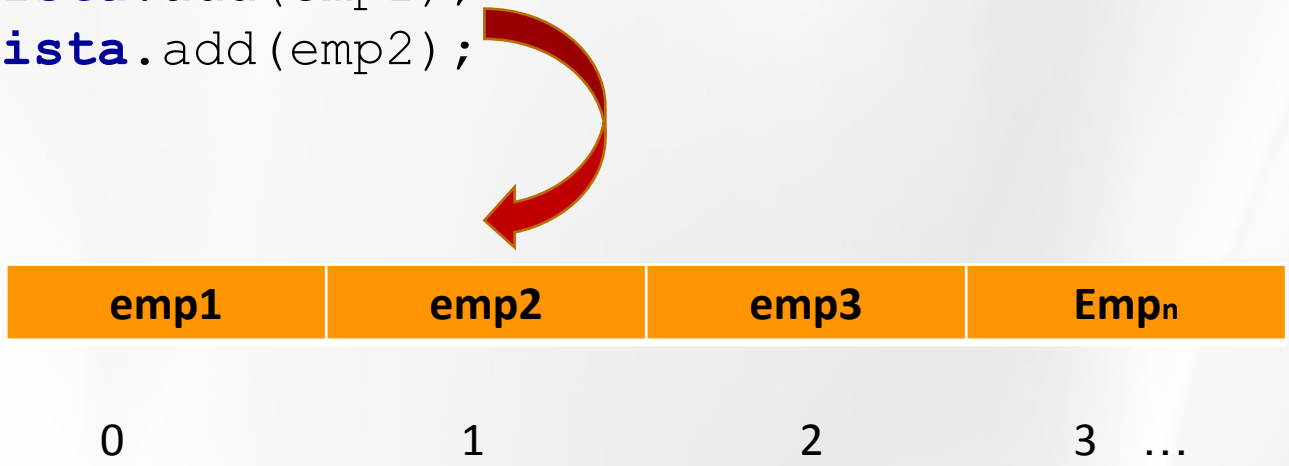
```
lista.add(emp1);
lista.add(emp2);
```

Ejemplo Arraylist: Agregar

```
Empleado emp2= new Empleado();

emp2.setId("e2");
emp2.setNombre("Juan Torres");
emp2.setCargo("Auxiliar");
emp2.setSalario(500);

lista.add(emp1);
lista.add(emp2);
```



Ejemplo Arraylist: Recorrido

```
lista.add(emp1);
lista.add(emp2);
```


```
Iterator iterador= lista.iterator();

while (iterador.hasNext()){
   Empleado e= (Empleado) iterador.next();

   System.out.println(e.getId() + " " + e.getNombre()
        + "\n " );
 }
```

Ejemplo Arraylist: Recorrido

```
lista.add(emp1);
lista.add(emp2);
```


```
for(int i=0; i<lista.size(); i++){

    Empleado e= (Empleado) lista.get(i);
    System.out.println(e.getId() + " " + e.getNombre() + "\n "
    );
}
```

Clase vector

java.util

Class Vector<E>

java.lang.Object
  └ java.util.AbstractCollection<E>
    └ java.util.AbstractList<E>
      └ java.util.Vector<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

La clase Vector implementa un *Array* de objetos

Clase vector

Vector(): Tamaño inicial de 10 y se incrementa duplicando el tamaño

Vector (int size): Señala el tamaño inicial y se incrementa duplicando el tamaño

Vector (int size, int increment): Señala el tamaño del incremento

Clase vector

addElement (Object)	Adiciona un elemento al final del vector.
elementAt(int index)	Retorna el elemento de la posición especificado en el índice.
equals(Object o)	Compara si el objeto actual es igual al especificado
get(int index)	Retorna el elemento de la posición especificado en el índice.
remove(int index)	Elimina el elemento de la posición especifica.

http://docs.oracle.com/javase/6/docs/api/java/util/Vector.html

Clase vector

```
public class  Empleado {
     String id;
     String nombre;
     String cargo;
     int salario;

}
```

Vector: Ejemplo

```
Vector empleados = new Vector ();

empleados.addElement(emp1);
empleados.addElement(emp2);
```

Recorrer un Vector

```
Iterator it= empleados.iterator();
```

```
while(it.hasNext()){
        Empleado e = (Empleado)it.next();
}
```

```
for(int i=0; i<empleados.size(); i++){

    Empleado e = (Empleado) empleados.get(i);

    System.out.println(e.getId() + " " +
        e.getNombre() + "\n " );

}
```

Ejercicio

Suponga que una empresa tiene almacenada la información de sus empleados en una colección de objetos. Implemente métodos en java para:

- Calcular el salario promedio de los empleados.
- Calcular el salario más alto y quien lo tiene
- Imprimir el nombre de aquellos empleados que ganan más del promedio.
- Mostrar el salario de los empleados cuyo cargo es “auxiliar”

Ejercicio

Ahora suponga que todos los empleados trabajan para un departamento: Ventas, producción o informática, esta información se encuentra en el empleado.

- Mostrar el nombre de los empleados agrupados por departamentos.

Colecciones

Gracias!!!

Luis Yovany Romo Portilla, MsC.

Fundamentos de Programación Orientada a Eventos