



Técnico en
< DESARROLLO DE SOFTWARE >

***Programación Orientada a
Objetos II***

(CC BY-NC-ND 4.0)
International

Attribution-NonCommercial-NoDerivatives 4.0



Atribución

Usted debe reconocer el crédito de una obra de manera adecuada, proporcionar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que tiene el apoyo del licenciante o lo recibe por el uso que hace.



No Comercial

Usted no puede hacer uso del material con fines comerciales.



Sin obra derivada

Si usted mezcla, transforma o crea un nuevo material a partir de esta obra, no puede distribuir el material modificado.

No hay restricciones adicionales - Usted no puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otros hacer cualquier uso permitido por la licencia.

<http://creativecommons.org/licenses/by-nc-nd/4.0/>



Programación Orientada a Objetos II

Unidad II

1. Paquetes

Un paquete es un contenedor donde se almacenan un conjunto de clases, interfaces y subpaquetes. Los paquetes permiten diferenciar el contenido y evitar que surjan conflictos, por ejemplo al utilizar el mismo nombre de una clase. Las clases o interfaces deben estar relacionadas en un paquete por lo que hacen o para lo que sirven.

Por ejemplo si tenemos las siguientes clases e interfaces:

```
public interface vehiculo {  
    ...  
}  
  
public class motocicleta implements vehiculo {  
    ...  
}  
  
public class carro implements vehiculo {  
    ...  
}
```

Podemos agrupar las clases e interfaces anteriores en un paquete denominado vehículos, para ello debemos agregar en el inicio de cada clase la palabra reservada package seguida por el nombre del paquete. Por ejemplo:

```
package vehiculos;

public interface vehiculo {

    ...

}

package vehiculos;

public class motocicleta implements vehiculo {

    ...

}

package vehiculos;

public class carro implements vehiculo {

    ...

}
```

El nombre del paquete siempre debe ser escrito en minúsculas para evitar conflictos con los nombres de las clases o interfaces. El nombre de la clase debe ser único dentro del mismo paquete.

Subpaquete

Un paquete puede contener subpaquetes, para definir un subpaquete se debe escribir al inicio de la clase:

package nombre_del_paquete.nombre_del_subpaquete

Por ejemplo el paquete `geometria` contiene los subpaquete `geometria.triangulos`, `geometria.rectangulos` y `geometria.circulos`.

Encapsulamiento

Se puede utilizar el encapsulamiento de clases dentro de los paquetes. Las clases e interfaces que se declaren públicas (`public`) podrán ser utilizadas desde otros paquetes, las clases o interfaces que se declaren protegidas (`protected`) sólo podrán ser utilizadas dentro del mismo paquete. Por ejemplo:

```
package banco;  
  
protected class cuenta {  
    ...  
}
```

Importar paquetes

Para poder utilizar clases de otros paquetes se deben importar los paquetes donde se encuentran estas clases. Para poder hacer esto se utiliza la palabra reservada `import` seguido de `nombre_del_paquete.nombre_de_la_clase`. Para importar todas las clases de un paquete se utiliza la notación `nombre_del_paquete.*`.

A continuación vamos a mostrar algunos ejemplos:

```
/* Importar clase Random */
```

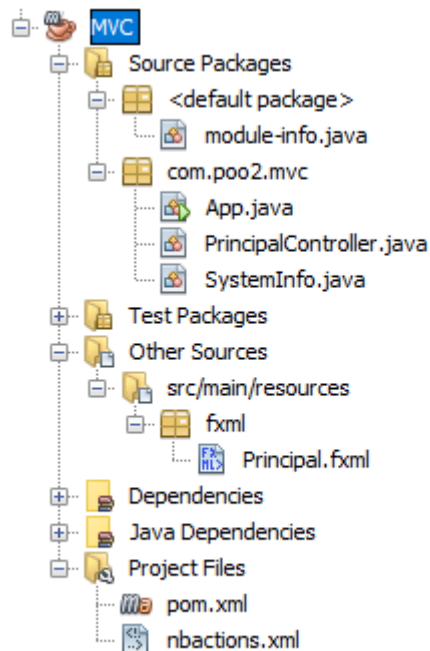
```
import java.util.Random;
```

```
/* Importar todas las clases del paquete java y subpaquete util */
```

```
import java.util.*;
```

Paquetes en IDE Netbeans

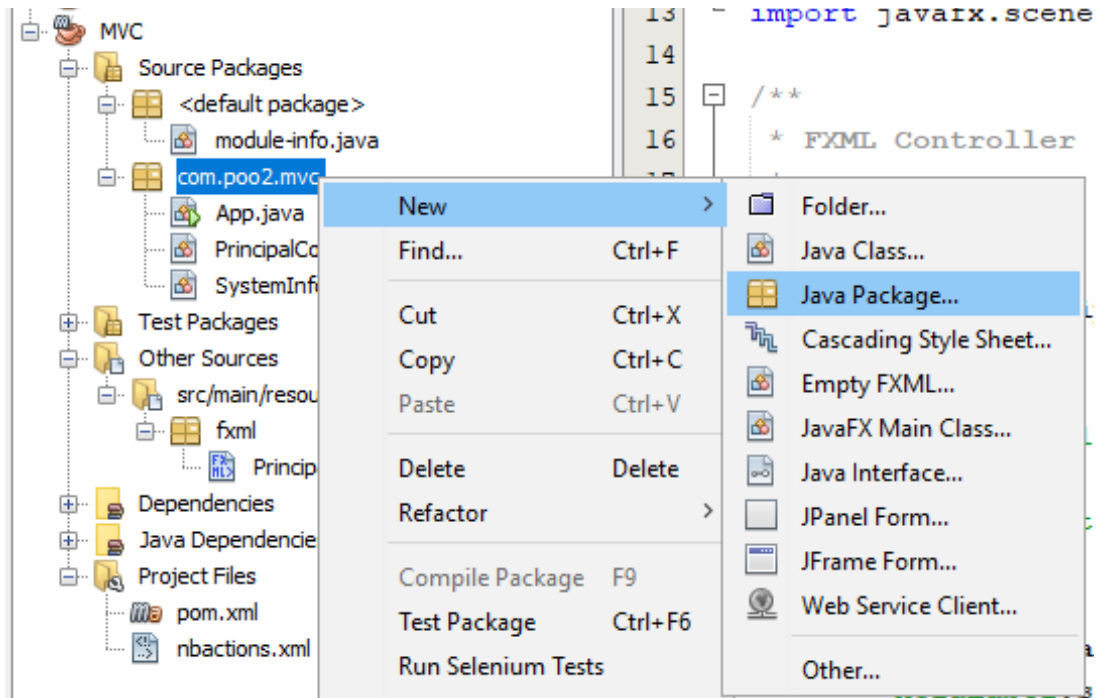
Hablemos de paquetes, la estructura básica de nuestros proyectos se ve así



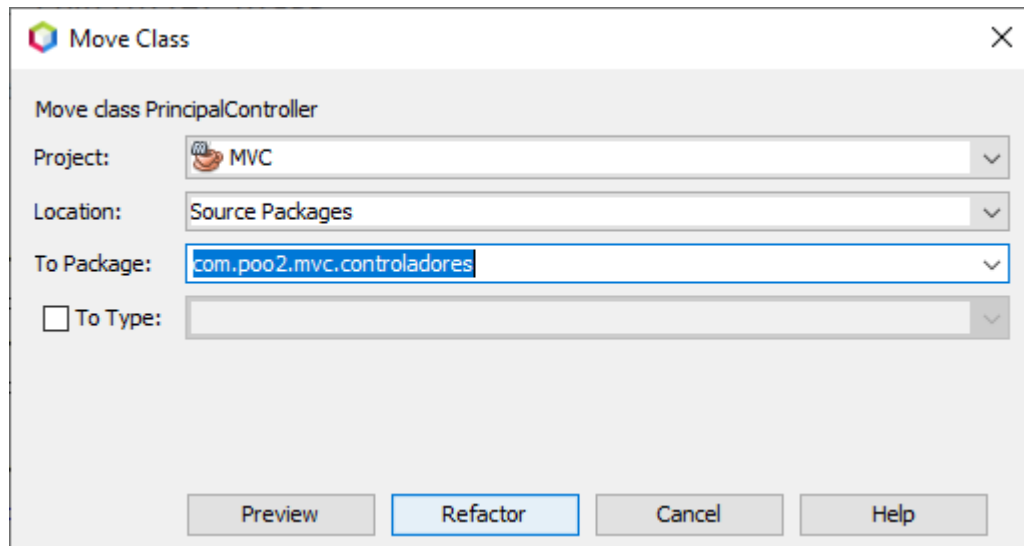
Un paquete es un conjunto de clases o recursos relacionados. Hemos trabajado en dos archivos, el Controlador y el archivo FXML. Estos dos pertenecen a dos paquetes. El primero, como lo podemos ver, pertenece al paquete principal de la aplicación. Esto es, el paquete que contiene todas las clases del proyecto. El archivo FXML pertenece al

paquete de recursos fxmL. Es decir, en este paquete vamos a encontrar solamente archivos FXML.

Pero podemos organizarlos un poco más, creemos un paquete que contenga solamente a los controladores, para separarlos de las clases, con el nombre controladores.



Ahora movemos el archivo de Principal Controller a nuestro nuevo paquete. Pero no basta con moverlo, tenemos que mantener las referencias a las clases. Por eso vamos a *Refactorizar*. Para esto hacemos clic derecho > Refactor > Move. Nos mostrará la siguiente ventana



En la tercera opción, seleccionamos el paquete nuevo: controladores. Luego hacemos clic en Refactor.

Agregando paquetes a un proyecto existente

Si tenemos un proyecto, el cual ya hemos creado archivos fxml y controladores, pero en el que no usamos paquetes, tenemos que verificar algunas cosas antes de poder correr el programa.

Recordemos el archivo de **modules-info.java**. Este representa las dependencias del proyecto, pero también representa a qué archivos podemos acceder. En particular recordemos la línea `opens com.poo2.mvc`; esta es usada por la propiedad de Reflexión, que si recordamos es la propiedad que nos permite vincular los componentes al código con la etiqueta `@FXML`. Usando Maven la propiedad de reflexión se puede usar únicamente en los paquetes especificados, a pesar de ser subpaquete del paquete mvc, no se tiene acceso al paquete controladores. Entonces hay dos opciones, se puede agregar la línea `opens com.poo2.mvc.controladores`; y dejar la que ya teníamos, o podemos reemplazar la línea por la segunda. Quitar el acceso a lugares que no utilizamos es la mejor práctica, pero si necesitamos acceder a algún componente (por

ejemplo para inicializarlo en el archivo App.java) vamos a tener que volver a agregar la línea `opens com.poo2.mvc;`

Nota: Si en este momento no se puede ejecutar el proyecto, el error más probable es que no se actualizó correctamente la referencia en el FXML relacionado, le damos a clic derecho > Edit, y en el archivo FXML buscamos la propiedad **fx:controller**, Esta propiedad es la que asocia al controlador con el fxml. Si no se actualizó, se vería así

```
fx:controller="com.poo2.mvc.PrincipalController"
```

Actualizamos la dirección del paquete así

```
fx:controller="com.poo2.mvc.controladores.PrincipalController"
```

2. Patrón MVC

El patrón MVC está conformado por tres capas: modelo, vista y controlador. Este patrón permite separar y organizar el código de acuerdo a su función (gestionar datos, interfaz gráfica y lógica de la aplicación) para que el software desarrollado sea escalable y mantible.

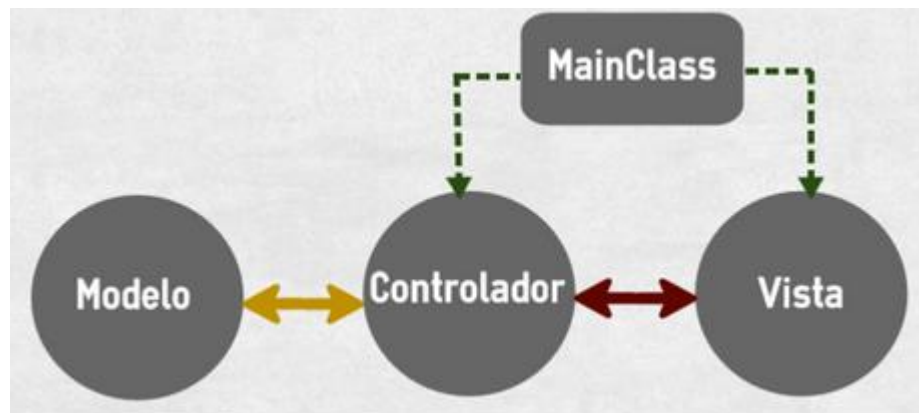
Uno de los mayores beneficios de utilizar MVC es que cada componente se puede modificar individualmente, además esto permite rastrear fácilmente el origen de un error.

Las capas están conformadas de la siguiente manera:

- **Modelo:** Contiene la definición de la clase del objeto. Además en esta capa se puede hacer integración de datos, parseo de datos o persistencia de datos. El modelo responde a las peticiones del controlador.

- **Vista:** Cuando un usuario interactúa con la aplicación, interactúa con la capa de vista. La vista contiene todos los componentes que conforman la interfaz gráfica de usuario, el diseño de la vista debe ser independiente del manejo de datos. En JavaFX las vistas son los archivos con extensión .fxml
- **Controlador:** Es responsable la lógica de la aplicación y procesa los datos ingresados por el usuario a través de periféricos de entrada (teclado, mouse, etc.). Mantiene comunicación con la vista y el modelo, permite actualizar los datos en el modelo así como también los componentes en la vista.

En JavaFX el patrón MVC se representa en el siguiente gráfico:



Adicionalmente al patrón MVC en JavaFX se puede utilizar una clase principal (MainClass) para poder gestionar el inicio del programa, conectar los controladores y las vistas, permitir la comunicación entre controladores y organizar el manejo de ventanas.

Ejemplo patrón MVC

Vamos a prepararnos para la parte de MVC, entonces vamos a crear otro paquete, que le llamaremos Modelos

En ese paquete agreguemos una clase Estudiante, que representará a un estudiante. Se verá así

```
package com.poo2.mvc.modelos;

public class Estudiante {

    String id;
    String nombre;
    String edad;

    public Estudiante(){}

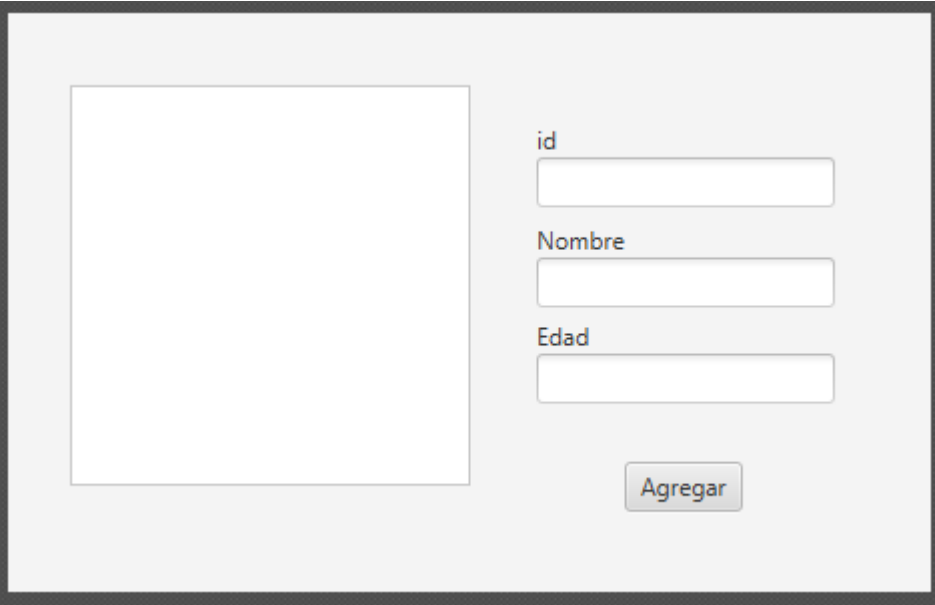
    public Estudiante(String id, String nombre, String edad){
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
}
```

```
public String getNombre() {  
    return nombre;  
}  
  
public void setNombre(String nombre) {  
    this.nombre = nombre;  
}  
  
public String getEdad() {  
    return edad;  
}  
  
public void setEdad(String edad) {  
    this.edad = edad;  
}  
  
@Override  
public String toString() {  
    return "Estudiante{" + "id=" + id + ", nombre=" + nombre + ", edad=" + edad + '}';  
}  
}
```

Ahora, vamos a crear la ventana, con un campo de texto para cada propiedad de la clase Estudiante, un botón y un ListView. Se debería ver similar a esto



A screenshot of a JavaFX application window. On the left is a large empty rectangular area. On the right, there are three text input fields stacked vertically, labeled 'id', 'Nombre', and 'Edad'. Below these fields is a button labeled 'Agregar'.

Recordemos darles ids a los campos, al botón y al listview, y también la acción del botón. Ahora movámonos al controlador, y vinculamos nuestros campos, luego en el método del botón, leemos el contenido de los campos, creamos un estudiante y luego lo mostramos en la consola. El controlador debería verse de la siguiente forma

```
public class PrincipalController implements Initializable {  
  
    @FXML  
    TextField campold;  
  
    @FXML  
    TextField campoNombre;  
  
    @FXML  
    TextField campoEdad;  
  
    @FXML  
    Button botonAgregar;  
  
    @FXML
```

```
ListView listaEstudiantes;
```

```
@FXML
```

```
public void agregarEstudiante(){
```

```
    Estudiante nuevoEstudiante = new Estudiante();
```

```
    nuevoEstudiante.setId(campold.getText());
```

```
    nuevoEstudiante.setNombre(campoNombre.getText());
```

```
    nuevoEstudiante.setEdad(campoEdad.getText());
```

```
    System.out.println(nuevoEstudiante);
```

```
}
```

```
@Override
```

```
public void initialize(URL url, ResourceBundle rb) {
```

```
    // TODO
```

```
}
```

```
}
```

Al ejecutar podemos comprobar el funcionamiento del botón. El cual crea un objeto de tipo estudiante y lo muestra en la consola.

Aquí podemos ver cómo funciona el patrón MVC, la Vista es la parte con la que interactúa el usuario, el Modelo es la parte que contiene la información y el Controlador es el intermediario que comunica a la vista y al modelo.

3. Estructuras de datos List y Map

Colecciones

Las colecciones están basadas en código reutilizable utilizando lenguaje orientado a objetos. Los tipos abstractos de datos se definen como interfaces. Estas interfaces permiten generalizar el comportamiento en varias implementaciones. Un mismo tipo abstracto de datos puede ser implementado por varias clases.

Estructuras de datos List y Map

Las estructuras de datos List y Map son interfaces de la clase `java.util.Collections`. Esta librería puede procesar estructuras de datos de tamaño fijo o variable.

List

Una estructura de datos List almacena de forma ordenada una colección de objetos de tamaño variable. La interfaz List define un conjunto de métodos útiles para manipular los elementos, por ejemplo agregar un elemento al final de la lista, buscar un elemento dentro de la lista.

Existen dos implementaciones generales de List: `ArrayList` y `LinkedList`. La implementación `ArrayList` mantiene un sistema basado en índices conjunto con la estructura de arreglo, esto permite que se puedan obtener rápidamente los resultados al hacer una búsqueda. La implementación `LinkedList` utiliza una lista doble encadenada con punteros lo que permite que las operaciones de agregar y eliminar sean muy rápidas.

Si el programa que se está desarrollando requiere más operaciones de búsqueda que de inserción o eliminación se debe utilizar `ArrayList`. Pero si se requieren menos

operaciones de búsqueda y más operaciones de inserción o eliminación se debe utilizar LinkedList.

A continuación veremos un ejemplo con las operaciones básicas:

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

public class ColeccionList {

    public static void main(String[] args) {
        List<String> list1 = new ArrayList<String>();

        List<String> list2 = new LinkedList<String>();
        list1.add("Elemento 1");

        list1.add("Elemento 2");

        list2.add("Elemento 1");

        list2.add("Elemento 2");

        for (Object o : list1) {
            System.out.println(o.toString());
        }

        for (Object o : list2) {
            System.out.println(o.toString());
        }

    }
}
```

Map

El mapa es una estructura de datos que relaciona una llave con un valor. En un mapa las llaves deben ser únicas y cada llave solo puede estar relacionada a un valor. Las llaves y los valores pueden ser definidos al momento de ejecutar el programa.

La clase más eficiente de Map es HashMap. HashMap permite almacenar valores nulos.

A continuación veremos un ejemplo de HashMap con las operaciones básicas:

```
import java.util.Map;
import java.util.HashMap;

public class ColeccionMap {
    public static void main(String[] args) {

        Map<String,Integer> map = new HashMap<String,Integer>();

        System.out.println("Vacío " + map.isEmpty());
        map.put("GT", 502);
        map.put("SV", 503);

        System.out.println("Vacío " + map.isEmpty());
        System.out.println("Tamaño " + map.size());

        System.out.println("Contiene Guatemala?" + map.containsKey("GT"));
        map.remove("GT");

        System.out.println("Contiene Guatemala?" + map.containsKey("GT"));
        map.clear();
    }
}
```

4. Colecciones en JavaFX

Las colecciones en JavaFX están definidas por el paquete `javafx.collections`, que consta de las siguientes interfaces y clases:

Interfaces

ObservableList: Lista que permite a los oyentes realizar un seguimiento de los cambios cuando se producen

ListChangeListener: Una interfaz que recibe notificaciones de cambios a un ObservableList

ObservableMap: un mapa que permite a los observadores realizar un seguimiento de los cambios cuando ocurren

MapChangeListener: Una interfaz que recibe notificaciones de cambios a un ObservableMap

Clases

FXCollections: Una clase de utilidad que consiste en métodos estáticos que son copias uno a uno de los métodos java.util.Collections

ListChangeListener.Change: Representa un cambio realizado en un ObservableList

MapChangeListener.Change: Representa un cambio realizado a un ObservableMap

Ejemplo:

Al implementar interfaces una de las tareas más comunes es refrescar datos en una ventana al modificar datos en otra. Para esto es necesario utilizar las colecciones de JavaFX. A continuación veremos un ejemplo:

```
import java.util.List;
import java.util.ArrayList;
import javafx.collections.ObservableList;
import javafx.collections.ListChangeListener;
import javafx.collections.FXCollections;

public class Colecciones {
```

```
public static void main(String[] args) {
    List<String> list = new ArrayList<String>();
    ObservableList<String> observableList =
FXCollections.observableList(list);
    observableList.addListener(new ListChangeListener() {
        @Override
        public void onChanged(ListChangeListener.Change change) {
            System.out.println("Cambio detectado");
        }
    });

    observableList.add("elemento");
}
```

Es importante notar que la estructura de datos que se debe manipular tiene que ser del tipo ObservableList para detectar cambios. Si en el ejemplo anterior se hacen cambios directamente en el list, el programa no va a mostrar ningún mensaje de cambio detectado.

Ejemplo de ListView

Continuando con el proyecto de MVC, ya habíamos creado un modelo y podíamos obtener la información de los campos, se crea un objeto y se muestra la información en la consola. Ahora vamos a mostrar esa información en la ListView que habíamos colocado anteriormente

Primero vamos a guardar los datos ingresados en una lista.

Para esto vamos a crear un ArrayList, y luego a guardar cada objeto creado en ella.

Eso se vería de esta forma

```
ArrayList<Estudiante> listaNuevosEstudiantes = new ArrayList();
```

@FXML

```
public void agregarEstudiante(){  
    Estudiante nuevoEstudiante = new Estudiante();  
    nuevoEstudiante.setId(campold.getText());  
    nuevoEstudiante.setNombre(campoNombre.getText());  
    nuevoEstudiante.setEdad(campoEdad.getText());  
  
    listaNuevosEstudiantes.add(nuevoEstudiante);  
    System.out.println(listaNuevosEstudiantes);  
}
```

Esto nos va a almacenar cada estudiante en una lista.

Ahora vamos a colocar esa lista en la listView.

Para eso, la listView necesita una lista de datos, pero no un ArrayList, sino una ObservableList.

La diferencia entre una ArrayList y una ObservableList es que la segunda es una ArrayList que nos puede informar cuando ha recibido un cambio, por ejemplo un ingreso de datos, una eliminación, o incluso un cambio en alguno de sus datos.

Entonces vamos a cambiar la ArrayList por una ObservableList, de esta forma.

```
ObservableList listaNuevosEstudiantes = FXCollections.observableArrayList();
```

Ya con esta lista, podemos conectarla a la ListView. En el método initialize colocamos lo siguiente

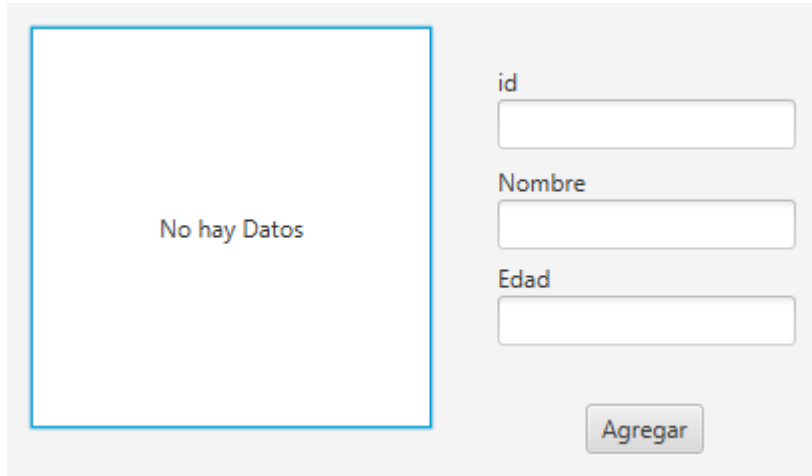
```
listaEstudiantes.setItems(listaNuevosEstudiantes);
```

Al vincular la lista observable con el ListView, cada vez que ingresemos un objeto a la lista, se va a actualizar el ListView automáticamente.

Para finalizar, podemos ver que al inicio del programa, la lista se encuentra vacía. Vamos a colocar un texto que se muestre para indicar que no tenemos información. Seguimos en el método initialize, coloquemos lo siguiente.

```
listaEstudiantes.setPlaceholder(new Label("No hay Datos"));
```

Ahora cuando ejecutamos el programa, se ve de esta forma.



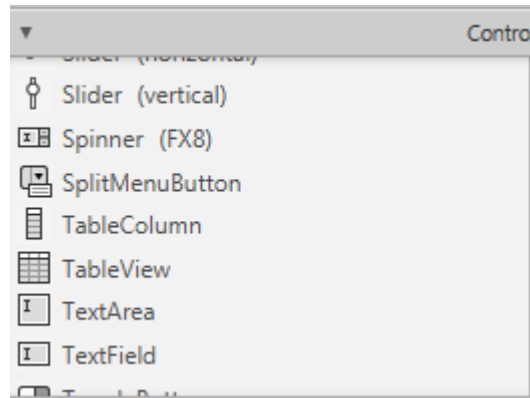
Ese texto va a desaparecer luego de ingresar un dato.

Table View

Esta vista nos ayuda a separar la información en columnas, lo que la hace mas legible y útil.

El proyecto de mvc funciona, pero en la lista se muestra la información de cada estudiante con el metodo toString, lo que no es la mejor idea. Vamos a cambiarlo para mostrar la información en una tabla. Para eso vamos a borrar el componente ListView en SceneBuilder y lo reemplazamos por un TableView. (Recordemos borrar el código relacionado al ListView de nuestro controlador)

Nota: Si necesitamos más columnas para nuestra tabla las podemos agregar con el componente TableColumn



La nueva ventana se debería ver así

Id	Nombre	Edad
Tabla sin contenido		

id

Nombre

Edad

Hay que tener cuidado, cuando usamos tablas le tenemos que colocar un id único a cada columna, ya veremos porque más tarde.

Le colocamos de Id a la tabla: `tablaEstudiantes`, y a las columnas, `columnaId`, `columnaNombre` y `columnaEdad` respectivamente.

Ahora tenemos que hacer algunas modificaciones a la clase `Estudiante`.

```
package com.poo2.mvc.modelos;
```

```
import javafx.beans.property.SimpleStringProperty;

public class Estudiante {

    SimpleStringProperty id;
    SimpleStringProperty nombre;
    SimpleStringProperty edad;

    public Estudiante(){
        id = new SimpleStringProperty();
        nombre = new SimpleStringProperty();
        edad = new SimpleStringProperty();
    }

    public Estudiante(String id, String nombre, String edad){
        this.id = new SimpleStringProperty(id);
        this.nombre = new SimpleStringProperty(nombre);
        this.edad = new SimpleStringProperty(edad);
    }

    public String getId() {
        return id.get();
    }

    public void setId(String id) {
        this.id.set(id);
    }
}
```



```
}

public String getNombre() {
    return nombre.get();
}

public void setNombre(String nombre) {
    this.nombre.set(nombre);
}

public String getEdad() {
    return edad.get();
}

public void setEdad(String edad) {
    this.edad.set(edad);
}

@Override
public String toString() {
    return "Estudiante{" + "id=" + id.get() + ", nombre=" + nombre.get() + ", edad=" +
        edad.get() + '}';
}
}
```

Lo que hacemos es cambiar los tipos de datos de String a SimpleStringProperty.



SimpleStringProperty es una implementación de Property, que es un tipo de dato Observable, es decir que podemos ver el estado y responder a los cambios.

Necesitamos esto por la forma en la que funciona el TableView.

Antes de seguir, también tenemos que agregar el paquete de modelos a nuestro archivo de **module-info.java** en estos momentos se verá así

```
module com.poo2.mvc {  
    requires javafx.controls;  
    requires javafx.fxml;  
    requires java.base;  
    opens com.poo2.mvc.controladores;  
    opens com.poo2.mvc.modelos;  
    exports com.poo2.mvc;  
}
```

Recordemos vincular la tabla y las columnas a nuestro controlador

```
@FXML  
TableView tablaEstudiantes;  
  
@FXML  
TableColumn columnaId;  
  
@FXML  
TableColumn columnaNombre;  
  
@FXML  
TableColumn columnaEdad;
```

Y ahora en el método initialize colocamos lo siguiente en lugar del código de la ListView

```
columnald.setCellValueFactory(new PropertyValueFactory<>("id"));  
  
columnaNombre.setCellValueFactory(new PropertyValueFactory<>("nombre"));  
  
columnaEdad.setCellValueFactory(new PropertyValueFactory<>("edad"));  
  
tablaEstudiantes.setItems(listaNuevosEstudiantes);
```

Lo que hacemos aquí es, a cada columna le colocamos un valor, que será de tipo Property (recordemos que son observables) asociado a un campo de la clase Estudiante. Es decir, a la columna Id va a estar asociado el id de un Estudiante.

Ahora al agregar un valor con el botón, nos va a mostrar el objeto Estudiante, pero separado, cada propiedad en una columna.

5. Múltiples Ventanas

JavaFX permite mostrar múltiples ventanas estas son de mucha utilidad para mejorar la experiencia del usuario y evitar saturar de componentes una sola ventana. Para crear una nueva ventana se debe definir un nuevo escenario, cargar los nodos a una nueva escena, asociar la nueva escena al escenario y mostrar el nuevo escenario. Se implementa de forma similar a cómo se crea la ventana principal. Por ejemplo:

```
private Stage secondStage;  
...  
public void openWindow() {  
    secondStage = new Stage();  
    FXMLLoader loader = new  
FXMLLoader(Main.class.getResource("/vista/Vista2.fxml"));  
    try {  
        Parent root = loader.load();  
        Scene scene = new Scene(root);  
        secondStage.setScene(scene);  
    }  
}
```

```
        secondStage.setTitle("Nueva ventana");
        secondStage.initModality(Modality.APPLICATION_MODAL);
        secondStage.initOwner(primaryStage);
        secondStage.show();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Para colocar un título a la ventana se utiliza el método setTitle del objeto escenario.

```
secondStage.setTitle("Agregar gasto");
```

Al crear el nuevo escenario se le debe indicar a qué escenario padre pertenece. Para ello se utiliza el método initOwner:

```
secondStage.initOwner(primaryStage);
```

Existen tres modalidades para las ventanas adicionales:

- **APPLICATION_MODAL**: Define una ventana modal que bloquea eventos en cualquier otra ventana abierta de la aplicación. Esto permite que el usuario solo se enfoque en la ventana actual.
- **NONE**: Define una ventana que es modal y no bloquea ninguna otra ventana.
- **WINDOW_MODAL**: Define una ventana modal que bloquea la entrega de eventos a toda la jerarquía de ventanas del propietario.

La modalidad se define con el método initModality del objeto escenario, por ejemplo:

```
secondStage.initModality(Modality.APPLICATION_MODAL);
```

6. Invocar métodos entre ventanas

Para poder invocar métodos entre ventanas se necesita colocar referencias a los controladores en la ventana principal y referencias a la ventana principal en cada controlador.

En cada controlador se debe agregar un atributo de la clase Main y el método setMain para hacer referencia a la ventana principal.

```
private Main main;
```

```
...
```

```
public void setMain(Main main) {  
    this.main = main;  
}
```

En la ventana principal se deben agregar como atributos las instancias de los controladores:

```
protected Controlador controlador;
```

```
protected Controlador2 controlador2;
```

En la clase Main después de cargar los nodos a root en cada ventana, se debe obtener el controlador de la vista y agregarle la referencia a Main.

```
Parent root = loader.load();
```

```
controlador = loader.getController();
```

```
controlador.setMain(this);
```

Ahora que se tiene la referencia a main, se puede abrir la ventana secundaria desde el controlador principal invocando el método OpenWindow.

```
public class Controlador {  
  
    private Main main;  
  
    public void nuevaVentana() {  
  
        this.main.openWindow();  
  
    }  
  
    public void setMain(Main main) {  
  
        this.main = main;  
  
    }  
  
}
```

7. Ejemplo de Múltiples Ventanas

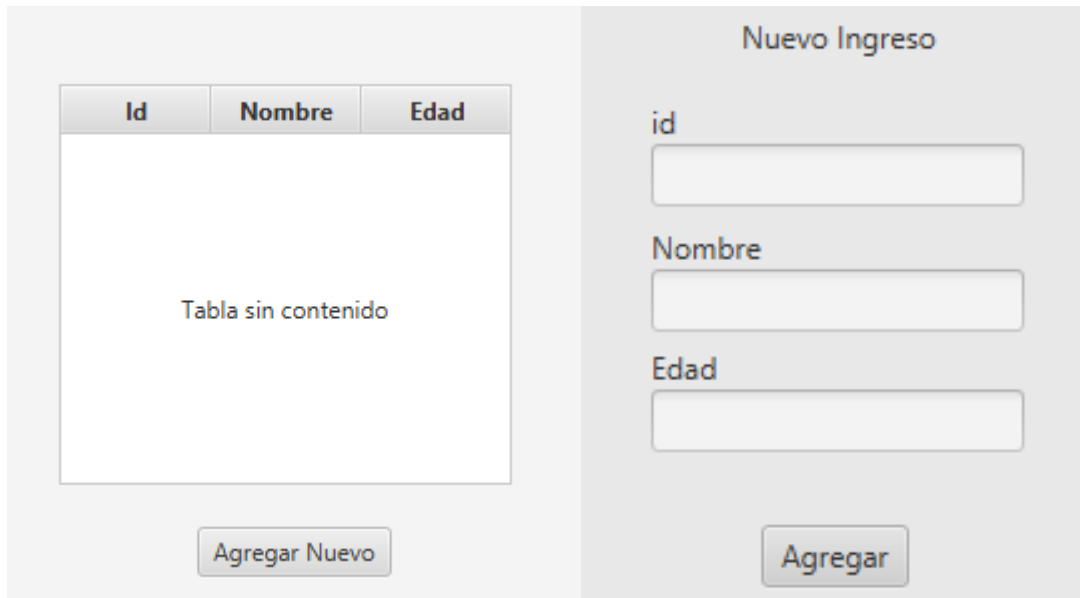
Finalmente, vamos a ver cómo trabajar con múltiples ventanas y cómo interactuar entre las ventanas.

Sigamos con el proyecto del mvc.

Ahora vamos a crear un archivo nuevo de fxml, le llamamos NuevoIngreso y agregamos un Controlador (siempre con cuidado de colocarlo en el paquete de controladores)

Lo que vamos a hacer es separar el formulario de ingreso de la vista de datos, entonces lo vamos a quitar del archivo de la ventana principal y lo vamos a colocar en el archivo de NuevoIngreso

Al final vamos a tener dos archivos que se verán así



Una ventana que contiene los datos, y una ventana que servirá para el ingreso, colocamos un botón en la primera para poder abrir la segunda.

Ya que cambiamos los componentes que hay en la ventana no olvidemos actualizar los campos vinculados en los controladores. Al botón nuevo en la ventana principal le colocamos de id botonAbrirAgregar y de On Action abrirAgregar.

Ahora vamos a modificar los controladores, el del controlador principal se vera asi

```
public class PrincipalController implements Initializable {
```

@FXML

Button botonAbrirAgregar;

@FXML

TableView tablaEstudiantes;

@FXML

TableColumn columnald;

@FXML

TableColumn columnaNombre;

@FXML

TableColumn columnaEdad;

ObservableList<Estudiante> listaNuevosEstudiantes =

FXCollections.observableArrayList();

@FXML

public void abrirAgregar(){

}

@FXML

public void agregarEstudiante(String id, String nombre, String edad){

Estudiante nuevoEstudiante = new Estudiante();

nuevoEstudiante.setId(id);

nuevoEstudiante.setNombre(nombre);


```
nuevoEstudiante.setEdad(edad);

listaNuevosEstudiantes.add(nuevoEstudiante);

System.out.println(listaNuevosEstudiantes);
}

@Override

public void initialize(URL url, ResourceBundle rb) {

    columnald.setCellValueFactory(new PropertyValueFactory<>("id"));
    columnaNombre.setCellValueFactory(new PropertyValueFactory<>("nombre"));
    columnaEdad.setCellValueFactory(new PropertyValueFactory<>("edad"));
    tablaEstudiantes.setItems(listaNuevosEstudiantes);

}

}
```

Lo que se cambió es que removimos los componentes que ya no están, y colocamos el botón nuevo así como el método asociado. Finalmente modificamos el método para agregar, de manera que recibe de parámetros los datos del estudiante. Esto es importante porque ahora va a recibir los datos, entonces el método debe de trabajar de esta manera.

Y el controlador de NuevoIngreso se verá así

```
public class NuevoIngresoController implements Initializable {
```

@FXML

TextField campold;

@FXML

TextField campoNombre;

@FXML

TextField campoEdad;

@FXML

Button botonAgregar;

@FXML

public void agregarEstudiante(){

}

@Override

public void initialize(URL url, ResourceBundle rb) {

// TODO

}

}

Con los controles que hemos quitado de la ventana anterior.

Ahora, para manejar dos controladores diferentes lo que tenemos que hacer es utilizar la clase principal del proyecto para llamar a los controladores. Vamos a trabajar sobre el archivo App.java, se vería de esta forma

```
public class App extends Application {  
  
    private Stage primaryStage;  
  
    @Override  
    public void start(Stage stage) throws IOException {  
        this.primaryStage = stage;  
        cargarPrincipal();  
    }  
  
    public void cargarPrincipal() {  
        FXMLLoader loader = new  
        FXMLLoader(getClass().getResource("/fxml/Principal.fxml"));  
  
        Parent root;  
        try {  
            root = loader.load();  
  
            Scene scene = new Scene(root);  
            primaryStage.setScene(scene);  
            primaryStage.show();  
        } catch (IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

```
public void cargarNuevoIngreso(){

    FXMLLoader loader = new
FXMLLoader(App.class.getResource("/fxml/NuevoIngreso.fxml"));

    Parent root;

    try {

        root = loader.load();

        Scene scene = new Scene(root);

        Stage stage2 = new Stage();

        stage2.setScene(scene);

        stage2.initOwner(primaryStage);

        stage2.initModality(Modality.WINDOW_MODAL);

        stage2.show();

    } catch (IOException ex) {

        ex.printStackTrace();

    }

}

public static void main(String[] args) {

    launch();

}

}
```

Creamos una propiedad primaryStage, que va a representar el escenario principal, es decir la primera ventana.

Luego creamos dos métodos en los cuales estamos creando las escenas de ambas ventanas. La primera es casi igual al método start que teníamos antes, con la excepción que ahora tenemos que colocarlo dentro de un try-catch porque ahora puede que se cause una excepción al buscar el archivo fxm1.

El segundo método es más interesante. Tenemos dos instrucciones nuevas. **stage2.initOwner(primaryStage);** esta nos coloca la segunda ventana como dependiente de la primera, es decir si en algún momento, la primera ventana se cierra, también se cerrará la segunda. **stage2.initModality(Modality.WINDOW_MODAL);** nos dice el comportamiento que ha de tener la ventana **WINDOW_MODAL** es un comportamiento que, cuando esta ventana este visible, no se puede interactuar con otras ventanas de la aplicación si pertenecen a la jerarquía de la misma, como colocamos initOwner, no podemos interactuar con la ventana principal.

Ahora coloquemos en los dos controladores una referencia a nuestra clase principal 'App' de esta forma

```
App main;  
  
public void setMain(App main){  
    this.main = main;  
}
```

Una variable de Clase App, y un método que nos permita obtener un valor.

Esto lo hacemos para llamar los métodos que acabamos de definir desde los controladores.

Pero tenemos que llamar esta función desde nuestra clase App. Lo hacemos antes de mostrar las ventanas, así.

```
public void cargarPrincipal() {  
    FXMLLoader loader = new  
        FXMLLoader(getClass().getResource("/fxml/Principal.fxml"));  
    Parent root;  
    try {  
        root = loader.load();  
        Scene scene = new Scene(root);  
        primaryStage.setScene(scene);  
  
        PrincipalController pc = loader.getController();  
        pc.setMain(this);  
  
        primaryStage.show();  
    } catch (IOException ex) {  
        ex.printStackTrace();  
    }  
}
```

Y en el controlador principal modificamos el método abrirAgregar así

```
@FXML  
public void abrirAgregar(){  
    main.cargarNuevoIngreso();  
}
```

Ahora podemos abrir la ventana de ingreso desde nuestra ventana principal.

Pero necesitamos enviar los ingresos desde la ventana de ingreso a la ventana principal, para eso declaremos una variable de tipo `PrincipalController` y le vamos a asignar valor cuando creemos la primera ventana

```
private PrincipalController pc;
```

Y luego

```
public void cargarPrincipal() {  
    FXMLLoader loader = new  
    FXMLLoader(getClass().getResource("/fxml/Principal.fxml"));  
  
    Parent root;  
  
    try {  
        root = loader.load();  
  
        Scene scene = new Scene(root);  
        primaryStage.setScene(scene);  
  
        pc = loader.getController();  
        pc.setMain(this);  
  
        primaryStage.show();  
    } catch (IOException ex) {  
        ex.printStackTrace();  
    }  
}
```

```
}
```

Finalmente agregamos un método que nos servirá para realizar el nuevo ingreso

```
public void agregarNuevoEstudiante(String id, String nombre, String edad){  
    pc.agregarEstudiante(id, nombre, edad);  
}
```

Ahora desde el controlador de nuevo ingreso, modificamos el método de agregarEstudiante así

```
@FXML  
  
public void agregarEstudiante(){  
    String id = campold.getText();  
    String nombre = campoNombre.getText();  
    String edad = campoEdad.getText();  
  
    main.agregarNuevoEstudiante(id, nombre, edad);  
}
```

Y ahora al ejecutar podemos ver que los ingresos se pasan de una ventana a la otra

8. Algoritmos de ordenamiento

Muchas veces tratamos con conjuntos de datos no ordenados, por ejemplo cuando estamos recibiendo entradas de datos de los usuarios.

Los conjuntos de datos no ordenados son bastante comunes, pero algunas veces queremos que los datos estén ordenados, como para mostrar los valores más altos, o para mostrarlos por fecha. Entonces lo que tenemos que hacer es ordenarlos, para esto hay muchas formas de hacerlo.

Un algoritmo de ordenamiento es aquel que nos permite, dado un conjunto de datos desordenados, obtener un conjunto ordenado de datos.




Vamos a ver 2 algoritmos de uso general. El primero, y uno de los más sencillos es el Bubble Sort, o Ordenamiento de Burbuja. Se le llama así porque para ordenar los datos lo que hace es ir “subiendo” un dato hasta que llega a la posición que le corresponde.

Un Bubble Sort, inicia desde la primera y la segunda posición de un arreglo, y verifica si el segundo valor es menor al primer valor, si este fuera el caso, los cambia de posición, si no, no hace nada. Luego se mueve a la segunda y la tercera posición y verifica nuevamente, si el tercer valor es menor al segundo, los cambia de lugar. Y así recorre todo el arreglo.

Este algoritmo no se detiene allí, ya que no hay garantía que quede ordenado. El algoritmo vuelve a recorrer el arreglo hasta que no haya realizado ningún cambio, no importando las veces que tenga que recorrerlo. Pero podemos cambiar el comportamiento manteniendo una variable que nos diga si hubo un cambio o no.

Dado el siguiente arreglo [2,7,3,10,6] veamos cómo lo arreglaría un Bubble Sort.

Primera pasada

 = Primer Valor  = Segundo Valor  = Valores cambiados de Posición

2	7	3	10	6
2	7	3	10	6
2	7	3	10	6
2	3	7	10	6
2	3	7	10	6
2	3	7	10	6
2	3	7	10	6
2	3	7	6	10

Luego de la primera pasada, vemos que aún no está ordenado. Entonces el algoritmo recorre el arreglo nuevamente

2	3	7	6	10
2	3	7	6	10
2	3	7	6	10
2	3	7	6	10
2	3	7	6	10
2	3	6	7	10

Cuando tratamos este algoritmo, el valor más alto siempre acaba en la posición de abajo, por lo que no hay necesidad de revisar este valor después de la primera pasada. Veamos qué pasaría si el 10 se encuentra en la primera posición

10	2	7	3	6
2	10	7	3	6
2	10	7	3	6
2	7	10	3	6
2	7	10	3	6
2	7	3	10	6
2	7	3	10	6
2	7	3	6	10

En todos los casos se movió el valor hacia atrás.

Este algoritmo es muy fácil de implementar, pero es muy lento, por lo que no se usa mucho.

La implementación en Java es de lo más sencilla

```
public class BubbleSort{

    public static void main(String[] args){

        int[] arreglo = {2,7,3,10,6};

        BubbleSort sort = new BubbleSort();
```

```
sort.bubbleSort(arreglo);

}

public void bubbleSort(int[] valores){
    int temp;
    Boolean cambio;
    mostrarArreglo(valores);
    for(int i = 0; i<valores.length-1; i++){
        cambio = false;
        for(int j = 0; j<valores.length-i-1; j++){
            if(valores[j+1] < valores[j]){
                cambio = true;
                temp = valores[j+1];
                valores[j+1] = valores[j];
                valores[j] = temp;
            }
        }
        if(!cambio){
            break;
        }
        mostrarArreglo(valores);
    }
}

// metodo auxiliar para mostrar el contenido del arreglo
```

```
public void mostrarArreglo(int[] arreglo){  
    for(int i = 0; i< arreglo.length; i++){  
        System.out.print(arreglo[i] + " ");  
    }  
    System.out.println("\n");  
}  
}
```

Este es un ordenamiento ascendente, es decir, los valores van del menor al mayor, si quisiéramos realizar un ordenamiento descendente, modificamos la condición del cambio, para que los números más pequeños queden hasta abajo.

El segundo algoritmo que vamos a ver se llama MergeSort, este algoritmo lo que hace es separar un arreglo en 2 partes, luego cada parte se vuelve a separar en 2 partes. Esto se hace hasta encontrar 2 elementos sueltos. Estos elementos se comparan y se colocan en la posición correcta. Esto se hace para todos los pares. Luego cada par, se compara con otro par, y se colocan en orden los elementos y así sucesivamente hasta volver a unir todos los valores.

Este es un algoritmo recursivo, es decir, vamos a volver a aplicar el algoritmo sobre cada mitad hasta que lleguemos a un caso base.

Para un arreglo [5,2,8,7,1,11,10,23,6] veamos cómo se vería el Merge Sort

5	2	8	7	1	11	10	6
5	2	8	7	1	11	10	6
5	2	8	7	1	11	10	6

En este punto, ya están separados, entonces podemos empezar a comparar y a unir los valores

Lo que hacemos en la parte de unir es: vamos revisando los valores de cada uno de los dos arreglos, y vamos colocando el menor valor de entre los dos arreglos. Así hasta unir los dos arreglos en 1 solo arreglo

2	5	7	8	1	11	10	6
2	5	7	8	1	11	6	23
2	5	7	8	1	6	11	23
1	2	5	6	7	8	10	23

El código se vería algo así

```
public class MergeSort {
    public static void main(String[] args){
        int[] valores = {5,2,8,7,1,11,10,6};
        MergeSort mergeSort = new MergeSort();
        mergeSort.sort(valores, 0, valores.length-1);
    }
    //la parte de merge une los arreglos y los ordena
    void merge(int[] valores, int izq, int mitad, int der){
        //calculamos el largo de los subarreglos
        int largolzquierdo = mitad - izq + 1;
```

```
int largoDerecho = der - mitad;

//arreglos derecho e izquierdo en los que se separa el arreglo original

int[] izquierdo = new int[largolzquierdo];

int[] derecho = new int[largoDerecho];

//copiamos valores a los arreglos

for (int i = 0; i < largolzquierdo; ++i){

    izquierdo[i] = valores[izq + i];

}

for (int j = 0; j < largoDerecho; ++j){

    derecho[j] = valores[mitad + 1 + j];

}

//valores iniciales de los valores

int i = 0, j = 0, k = izq;

/*ordenamos los valores, comparando los valores de los arreglos izquierdo y
derecho

y colocamos el menor en el arreglo original */

while (i < largolzquierdo && j < largoDerecho){

    if (izquierdo[i] <= derecho[j]) {

        valores[k] = izquierdo[i];

        i++;

    }

    else {

        valores[k] = derecho[j];

        j++;

    }

    k++;

}
```

```
}
```

//si los arreglos no son del mismo tamaño van a sobrar valores en alguno de los arreglos

```
while (i < largolzquierdo) {
```

```
    valores[k] = izquierdo[i];
```

```
    i++;
```

```
    k++;
```

```
}
```

```
while (j < largoDerecho) {
```

```
    valores[k] = derecho[j];
```

```
    j++;
```

```
    k++;
```

```
}
```

```
}
```

//El metodo principal, como es recursivo, lo llamamos con el valor de incio y de final del arreglo

```
void sort(int arr[], int izquierdo, int derecho) {
```

//Si se diera el caso que izquierdo >= derecho es que ya llegamos a un solo valor

```
if (izquierdo < derecho) {
```

```
    // se separa el arreglo en dos mitades
```

```
    int mitad = (izquierdo + derecho) / 2;
```

```
    //se ordenan las mitades
```

```
    sort(arr, izquierdo, mitad);
```

```
    printArray(arr);
```



```
        sort(arr, mitad + 1, derecho);

        printArray(arr);

        //al final se unen nuevamente los valores

        merge(arr, izquierdo, mitad, derecho);
    }
}

//Funcion para imprimir los arreglos
static void printArray(int[] valores) {
    for (int i = 0; i < valores.length; ++i) {
        System.out.print(valores[i] + " ");
    }
    System.out.println();
}
}
```

La interfaz Comparable

En Java existe una interfaz que podemos usar, que se llama Comparable, nos permite definir los criterios de comparación para una clase, por ejemplo

```
class Nota implements Comparable{

    String nombre;

    int valor;

    public Nota(String n, int v){
```

```
    this.nombre = n;

    this.valor = v;
}

@Override

public int compareTo(Object o){

    if(this == o){

        return 0;

    }

    Nota objeto = (Nota)o;

    return this.valor - objeto.valor;

}

public String toString(){

    return this.nombre + " "+ this.valor;

}

}
```

Esta clase representa la nota obtenida en una asignación, y lo que nos interesa es saber que nota es mayor o menor.

Podemos definir la comparación como nosotros queramos, en este caso solo nos interesa el valor de la nota.

Una comparación usando la interfaz comparable se llamara de la siguiente forma

```
objeto1.compareTo(objeto2)
```

compareTo nos va a devolver un entero con las siguientes condiciones

- 0 si el valor de ambos objetos es igual
- -1 si el valor de objeto1 es menor al de objeto2
- 1 si el valor de objeto1 es mayor al de objeto2

A la hora de implementar la interfaz, los valores de las últimas dos condiciones no tienen que ser estrictamente 1 y -1, basta con que sea un número positivo y uno negativo.

Una de las cosas que nos permite implementar la interfaz es hacer uso de las comparaciones para ordenar nuestros objetos. Por ejemplo, con algunas modificaciones al Bubble Sort, podemos hacer que pueda arreglar nuestros objetos Nota

```
public class BubbleSort{  
    public static void main(String[] args){  
  
        Nota[] notas = {new Nota("A", 100), new Nota("D", 60), new Nota("C", 70), new  
        Nota("B", 85), new Nota("F",50)};  
  
        BubbleSort sort = new BubbleSort();  
        sort.bubbleSort(notas);  
  
    }  
}
```

```
public void bubbleSort(Nota[] valores){  
    Nota temp;  
    Boolean cambio;  
    mostrarArreglo(valores);  
    for(int i = 0; i<valores.length-1; i++){  
        cambio = false;  
        for(int j = 0; j<valores.length-i-1; j++){  
            if(valores[j+1].compareTo(valores[j]) < 0){  
                cambio = true;  
                temp = valores[j+1];  
                valores[j+1] = valores[j];  
                valores[j] = temp;  
            }  
        }  
        if(!cambio){  
            break;  
        }  
        mostrarArreglo(valores);  
    }  
}
```

```
public void mostrarArreglo(Nota[] arreglo){  
    for(int i = 0; i< arreglo.length; i++){  
        System.out.print(arreglo[i] + " ");  
    }  
}
```

```
    }  
    System.out.println("\n");  
}  
}  
  
class Nota implements Comparable{  
    String nombre;  
    int valor;  
  
    public Nota(String n, int v){  
        this.nombre = n;  
        this.valor = v;  
    }  
  
    @Override  
    public int compareTo(Object o){  
        if(this == o){  
            return 0;  
        }  
  
        Nota objeto = (Nota)o;  
        return this.valor - objeto.valor;  
    }  
    public String toString(){  
        return this.nombre + " " + this.valor;  
    }  
}
```

```
}  
  
}
```

Descargo de responsabilidad

La información contenida en este documento descargable en formato PDF o PPT es un reflejo del material virtual presentado en la versión online del curso. Por lo tanto, su contenido, gráficos, links de consulta, acotaciones y comentarios son responsabilidad exclusiva de su(s) respectivo(s) autor(es) por lo que su contenido no compromete al área de e-Learning del Departamento GES o al programa académico al que pertenece.

El área de e-Learning no asume ninguna responsabilidad por la actualidad, exactitud, obligaciones de derechos de autor, integridad o calidad de los contenidos proporcionados y se aclara que la utilización de este descargable se encuentra limitada de manera expresa para los propósitos educativos del curso.

