UD 03 – FXML e informes

1. El patrón Modelo-Vista-Controlador (MVC)

El desarrollo de aplicaciones con interfaz gráfica requiere una buena organización del código para facilitar su mantenimiento, reutilización y escalabilidad. Cuando se diseña una aplicación aplicando el patrón de diseño *modelo-vista-controlador*, se persigue separar la aplicación en 3 capas diferenciadas:

- **VISTA**

  Hace referencia a la ventana principal de la aplicación, la que contiene la mayoría de los elementos sobre los que el usuario va a interactuar.

  En MVC, la vista solo contiene el código referente a la construcción y organización de los componentes gráficos. Esta clase no contiene ningún *manejador de eventos* ni ningún código que haga alguna operación. Tampoco tiene ninguna interacción con el modelo, y es independiente del modelo y del controlador; de este modo es un componente reutilizable.

  Todos los componentes gráficos (botones que se puedan pulsar, campos de texto de los que obtener datos, cajas de texto, etc) que quiera gestionar desde el controlador deben tener visibilidad **default** también llamada *package-private*.

- **MODELO**

  Es la capa que contiene los datos de la aplicación y los gestiona. Es la que se encarga de satisfacer las peticiones del usuario que se indican en la vista. Recordemos que el usuario solo interactúa con la vista.

  El modelo contiene todos los métodos para realizar las operaciones de nuestra aplicación. Dar de alta elementos, eliminar, buscar, guardar, cargar, etc. Es completamente independiente de la vista y del controlador.

  El modelo es la clase que maneja los datos de la aplicación y permite las operaciones. Los puede obtener desde un fichero o desde una base de datos. Puede cargar esos datos en una estructura como el ArrayList o leerlos cada vez. No tiene ningún conocimiento ni interacción con la vista, y devuelve y recibe los datos tal cual se los pide el controlador.

- **CONTROLADOR**

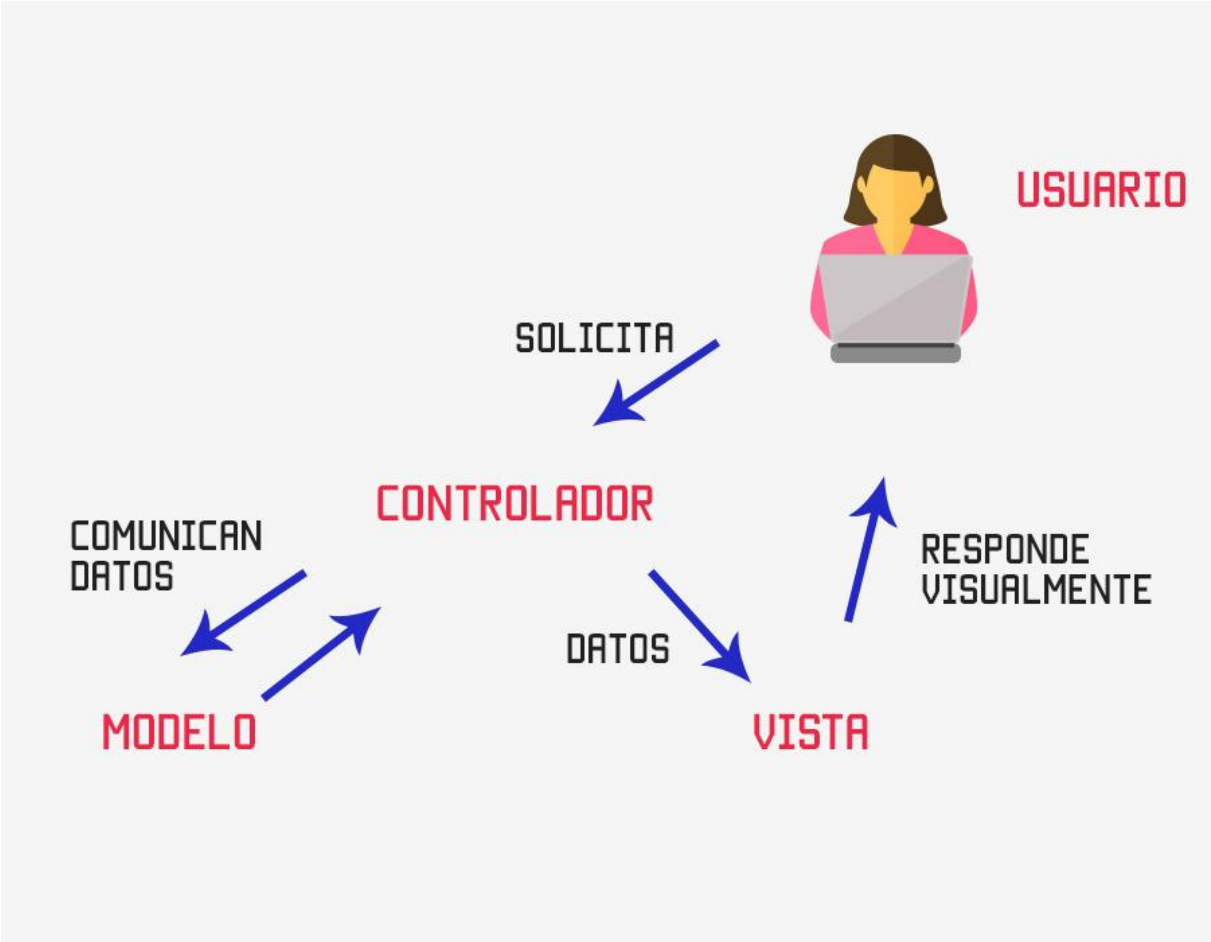
  Es la capa que comunica a las otras dos. Al llamar a su constructor, se le pasa como parámetro una instancia de la vista, y otra del modelo.

  El controlador es quien tiene implementados los manejadores de eventos y se los añade a los componentes de la vista indicados, y también es quien ejecuta las operaciones del modelo en respuesta a esos eventos.

- **APLICACIÓN PRINCIPAL**

  Desde una clase con un método main se lanza la aplicación, creando una instancia de cada clase.

Resulta aconsejable que al menos las clases que representan la vista y el controlador **estén dentro del mismo paquete** (package). De este modo podemos hacer uso del modificador package-private, también conocido como *default*.



1.1. Ejemplo del patrón MVC

MiVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.VBox?>
```

```
<VBox spacing="10" xmlns:fx="javafx.com"
fx:controller="com.ejemplo.controlador.MiControlador">
    <TextField fx:id="textoEntrada" promptText="Escribe algo" />
    <Button text="Saludar" onAction="# saludarClick" />
    <Label fx:id="etiquetaSalida" text="Esperando entrada..." />
</VBox>
```

MiControlador.java

```
package com.ejemplo.controlador;
import com.ejemplo.modelo.MiModelo; // Asumimos un modelo
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;

public class MiControlador {

    @FXML
    private TextField textoEntrada;

    @FXML
    private Label etiquetaSalida;

    private MiModelo modelo; // Referencia al modelo

    public MiControlador() {
        modelo = new MiModelo(); // Inicializa el modelo
    }

    @FXML
    public void saludarClick() {
        String texto = textoEntrada.getText();
        modelo.setMensaje(texto); // Actualiza el modelo
        etiquetaSalida.setText("Hola, " + modelo.getMensaje()); // Actualiza
la vista desde el modelo
    }
}
```

MiModelo.java

```
package com.ejemplo.modelo;

public class MiModelo {
    private String mensaje;
```

```
    public String getMensaje() {
        return mensaje;
    }

    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }
}
```

AplicacionPrincipal.java

```
package com.ejemplo;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.stage.Stage;
import java.io.IOException;

public class AplicacionPrincipal extends Application {
    @Override
    public void start(Stage stage) throws IOException {
        FXMLLoader fxmlLoader = new
FXMLLoader(AplicacionPrincipal.class.getResource("MiVista.fxml"));
// Carga FXML
        Scene scene = new Scene(fxmlLoader.load(), 300, 200);
        stage.setTitle("MVC Simple JavaFX");
        stage.setScene(scene);
        stage.show();
    }

    public static void main (String[] args){
        launch(args);
    }
}
```

2. Interfaces de usuario descritas en XML

FXML es un lenguaje basado en XML que permite describir la estructura de una interfaz gráfica JavaFX de forma declarativa. En lugar de crear la interfaz completamente mediante código Java, los componentes visuales, su jerarquía y muchas de sus propiedades se definen en un archivo FXML.

El uso de interfaces descritas en XML presenta múltiples ventajas:

- Separación clara entre diseño y lógica.
- Facilita el mantenimiento y la modificación de la interfaz.
- Permite que diseñadores y programadores trabajen de forma independiente.
- Mejora la legibilidad del proyecto y reduce la complejidad del código Java.

Una vez creada la interfaz, es fundamental comprender la estructura del archivo FXML generado. Este documento refleja la jerarquía de nodos de la escena y define propiedades como tamaños, estilos y eventos.

2.1. Creación visual de interfaces con Scene Builder

Scene Builder es una herramienta gráfica que permite diseñar interfaces JavaFX de forma visual, generando automáticamente el archivo FXML correspondiente.

A través de esta herramienta se pueden:

- Seleccionar contenedores de diseño (VBox, HBox, GridPane, BorderPane, etc.).
- Añadir controles como botones, etiquetas, campos de texto, tablas o menús.
- Configurar propiedades visuales y de comportamiento sin escribir código.
- Asignar identificadores (`fx:id`) y métodos de eventos a los componentes.

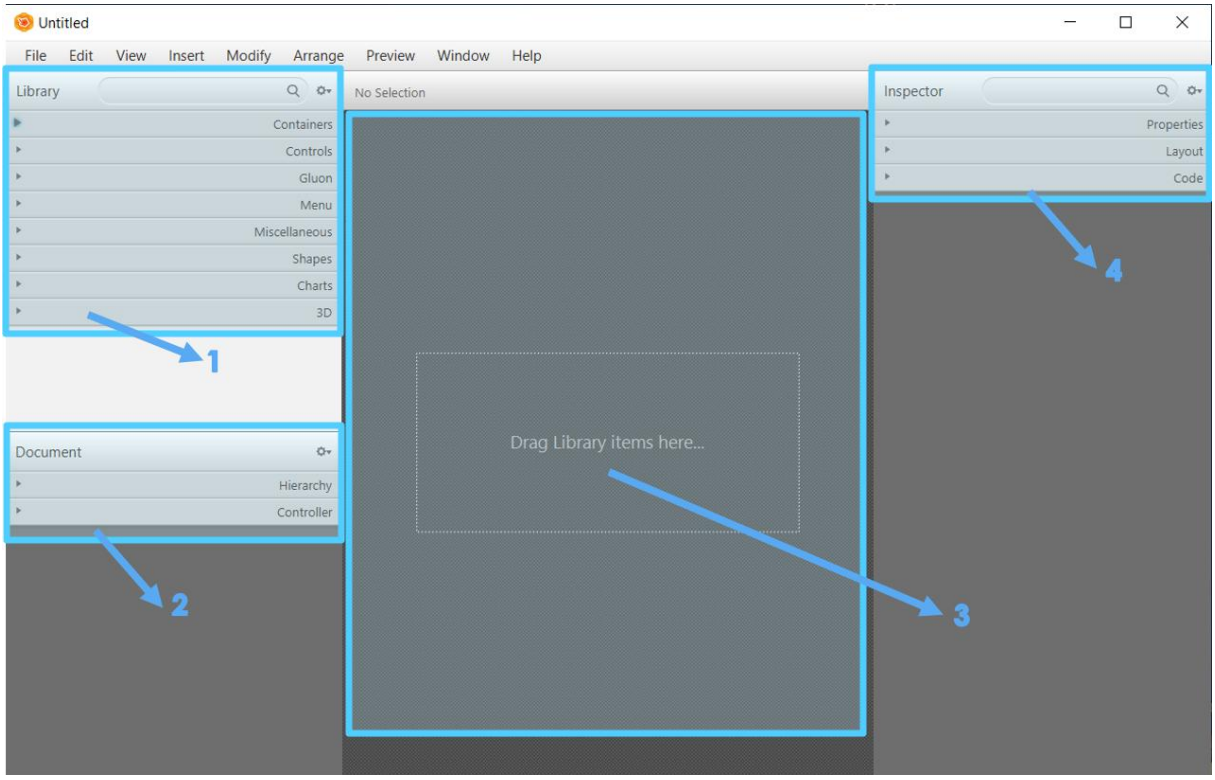
Scene Builder facilita la creación rápida de prototipos y reduce errores en la definición inicial de la interfaz.

a. Instalación

- Descargar Scene Builder desde la web oficial de Gluon.
- Instalar la herramienta según el sistema operativo.
- Integrarlo con el IDE:
  - Asociar los archivos `.fxml` para que se abran con Scene Builder.
  - En NetBeans: *Tools → Options → Java → JavaFX → Scene Builder Home*.
- Una vez configurado, al hacer doble clic sobre un archivo FXML, este se abrirá directamente en Scene Builder.

b. Interfaz de Scene Builder: partes y función de cada una

Scene Builder se divide en varias zonas claramente diferenciadas, cada una con una función concreta en el diseño de la vista.



- **1. Library (Biblioteca):**
  - Contiene todos los controles y contenedores JavaFX.
  - Se divide, principalmente, en:
    - *Containers*: AnchorPane, VBox, HBox, GridPane, BorderPane…
    - *Controls*: Button, Label, TextField, TableView, etc.
  - Se utilizan mediante *arrastrar y soltar* sobre el área central.
- **2. Document (Documento):**
  - **Content (Contenido):**
    - Muestra el árbol jerárquico de nodos de la escena.
    - Permite ver claramente qué elementos contienen a otros.
    - Muy útil para reorganizar la interfaz y detectar errores de diseño.
  - **Controller (Controlador):**
    - Permite administrar la clase controladora que se desea usar con el documento FXML.
    - Esta clase proporciona la lógica para gestionar el comportamiento de los objetos en el diseño FXML.
- **3. Design Area (Área de diseño):**
  - Zona central donde se construye visualmente la interfaz.

- Representa el aspecto final de la escena.

- **4. Inspector:**
  - Panel clave para la configuración.
  - Se divide en pestañas:
    - **Properties**: propiedades visuales y funcionales.
    - **Layout**: alineación, márgenes, restricciones del contenedor.
    - **Code**: conexión con el controlador (fx:id, eventos, controlador).

c. Configuración clave para la relación Vista–Controlador

La conexión entre la vista (FXML) y el controlador Java se establece mediante una serie de **atributos fundamentales**, configurados principalmente en el panel **Code** y desde **Controller**. Estos elementos esenciales son:

- **fx:controller**
  - Indica la clase controladora asociada a la vista.
- **fx:id**
  - Identificador que permite acceder a un componente desde el controlador.
- **Eventos**
  - Métodos del controlador que se ejecutan ante acciones del usuario (por ejemplo, onAction).

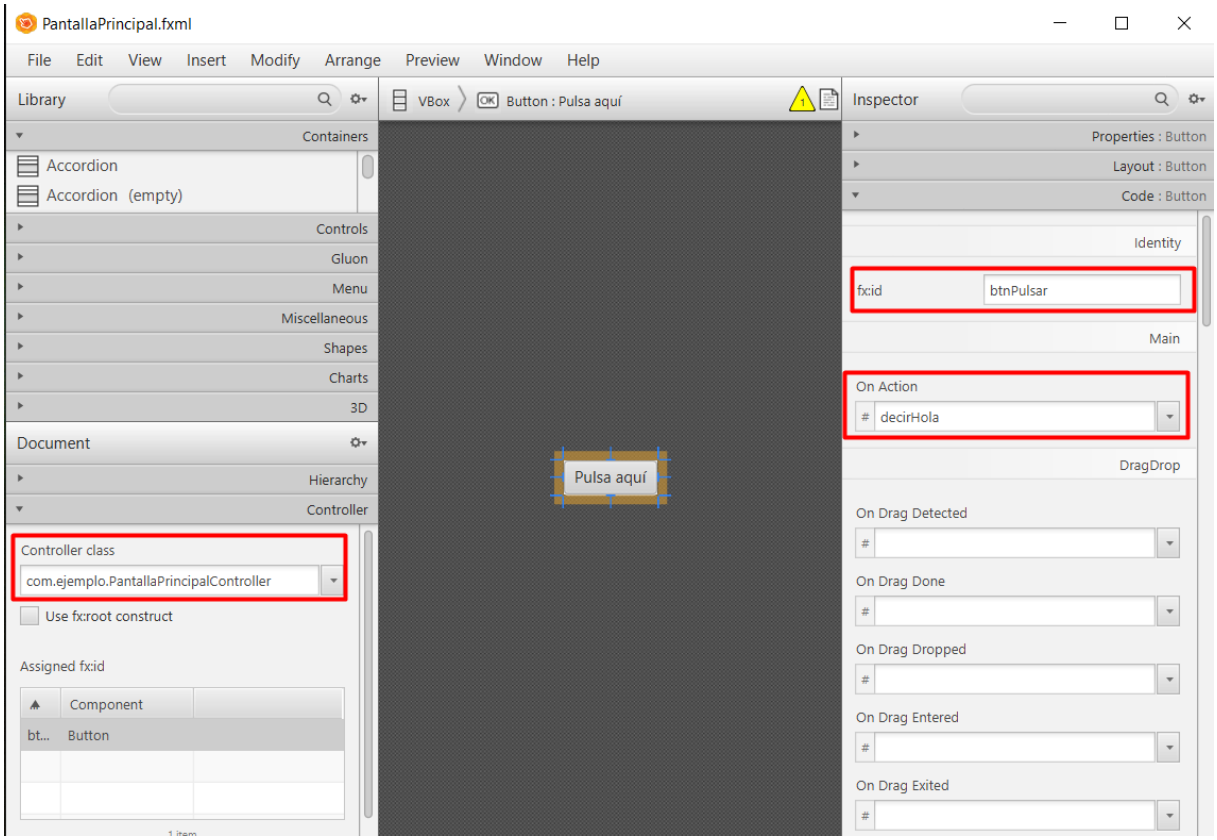
2.2. De Scene Builder a FXML

Scene Builder actúa como una herramienta de edición gráfica que permite construir interfaces JavaFX de forma intuitiva mediante el uso de contenedores y controles, mientras que **FXML es la representación textual en XML de esa interfaz**.

Cada elemento que se añade en el editor visual (contenedores, botones, propiedades, eventos, alineaciones, etc.) se traduce automáticamente en **etiquetas XML y atributos** dentro del archivo FXML. Comprender esta relación es fundamental para:

- Interpretar el código generado por Scene Builder.
- Modificar manualmente el FXML cuando sea necesario.
- Entender cómo la vista se conecta con el controlador dentro del patrón MVC.

En la siguiente imagen se puede observar **la interfaz diseñada gráficamente en Scene Builder** y, por otro, **el archivo FXML resultante**, lo que permite identificar claramente la correspondencia entre ambos.



Código FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.VBox?>

<VBox alignment="CENTER"
      spacing="10"
      xmlns="http://javafx.com/javafx/25"
      xmlns:fx="http://javafx.com/fxml/1"
      fx:controller="com.ejemplo.PantallaPrincipalController">

    <Button fx:id="btnPulsar"
            onAction="#decirHola"
            text="Pulsa aquí" />
</VBox>
```

a. Análisis de las partes importantes del archivo FXML

A continuación, se describen los elementos más relevantes del siguiente código FXML y su función dentro de la aplicación JavaFX:

Código FXML

```
<?xml version="1.0" encoding="UTF-8"?>
```

- Indica que el documento es un archivo XML.
- Especifica la codificación de caracteres utilizada.

Código FXML

```
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.VBox?>
```

- Importan las clases JavaFX necesarias para los componentes utilizados en la vista.
- Scene Builder las añade automáticamente según los controles usados.

Código FXML

```
<VBox alignment="CENTER"
      spacing="10"
      xmlns="http://javafx.com/javafx/25"
      xmlns:fx="http://javafx.com/fxml/1"
      fx:controller="com.ejemplo.PantallaPrincipalController">
```

Este bloque define el **nodo raíz de la interfaz** y contiene varios aspectos clave:

- **VBox**
  Es el contenedor principal. Organiza los elementos hijos de forma vertical.
  - **Propiedades de diseño**
    - **`alignment="CENTER"`**: centra los elementos dentro del contenedor.
    - **`spacing="10"`**: establece un espacio de 10 píxeles entre los componentes.
  - **Espacios de nombres (xmlns)**
    - **`xmlns`**: define el espacio de nombres de JavaFX.
    - **`xmlns:fx`**: habilita el uso de atributos específicos de FXML como fx:id o fx:controller.

- **fx:controller**
  - Indica la clase controladora asociada a esta vista. Es el punto clave de conexión entre la **vista (FXML)** y el **controlador (Java)**.
  - En este caso es la clase **`PantallaPrincipalController.java`** del paquete **`com.ejemplo`**

Código FXML

```
<Button fx:id="btnPulsar"
        onAction="#decirHola"
        text="Pulsa aquí" />
```

- **Button**
  Control de interfaz que permite al usuario lanzar una acción.
- **fx:id="btnPulsar"**
  - Identificador del botón dentro del FXML.
  - Permite acceder a este componente desde el controlador:

Java

```
@FXML
private Button btnPulsar;
```

- **onAction="#decirHola"**
  - Asocia el evento de pulsación del botón con un método del controlador.
  - El método `decirHola()` debe existir en la clase controladora y estar anotado con `@FXML`.
- **text="Pulsa aquí"**
  - Texto que se muestra en el botón.
  - Se corresponde con la propiedad configurada visualmente en Scene Builder.

2.3. Carga del archivo FXML desde el punto de entrada de la aplicación

Cuando se trabaja con **JavaFX sin FXML**, la interfaz gráfica se construye íntegramente dentro del método `start(Stage stage)`. En este método se crean los nodos de la interfaz, se configuran sus propiedades y se añaden manualmente a la escena. Este enfoque concentra en un mismo lugar la definición visual, el comportamiento y la inicialización de la aplicación.

Con la introducción de **FXML**, este planteamiento cambia. La definición de la interfaz gráfica se externaliza a un archivo FXML y la **clase principal reduce su responsabilidad**. El método `start(Stage stage)`, que JavaFX ejecuta automáticamente al iniciar la aplicación, pasa a encargarse únicamente de **cargar la vista y mostrarla**, delegando la creación de la interfaz en el cargador FXML.

a. Enfoque tradicional: JavaFX sin FXML:

- El método `start()` construye toda la jerarquía de nodos.
- En el mismo código se mezclan:
  - la estructura visual de la interfaz,
  - la configuración de propiedades,
  - la gestión de eventos.
- Cualquier cambio en el diseño obliga a modificar código Java y recompilar la aplicación.

b. Nuevo enfoque: JavaFX con FXML

Al trabajar con FXML:

- El método `start()` no crea nodos manualmente.
- La jerarquía visual se define completamente en un archivo FXML.
- El `start()` se limita a:
  - cargar la vista,
  - crear la escena,
  - mostrarla en el `Stage`.
- La lógica de interacción se traslada al controlador asociado a la vista.

c. Papel de `FXMLLoader`

Con FXML, la responsabilidad de construir la interfaz pasa a la clase `FXMLLoader`, que se encarga de:

- Leer el archivo FXML.
- Crear la jerarquía completa de nodos JavaFX.
- Aplicar propiedades visuales y eventos definidos en el XML.
- Instanciar automáticamente el controlador indicado mediante `fx:controller`.

Desde la clase principal, el desarrollador solo necesita **recuperar el nodo raíz ya construido**.

d. Flujo de carga de una vista FXML

Aunque internamente el proceso es más abstracto, el flujo sigue siendo equivalente al modelo tradicional:

- Localizar el archivo FXML como un recurso del proyecto.
- Cargar la vista mediante `FXMLLoader.load()`.
- Obtener el nodo raíz de la interfaz.
- Crear la escena (`Scene`) a partir de ese nodo.
- Asignar la escena al `Stage` y mostrar la ventana.

Este flujo mantiene la lógica habitual de JavaFX, pero con una **separación clara entre vista y código**, que facilita el mantenimiento y la evolución de la aplicación.

e. Ejemplo comparativo

JavaFX sin FXML

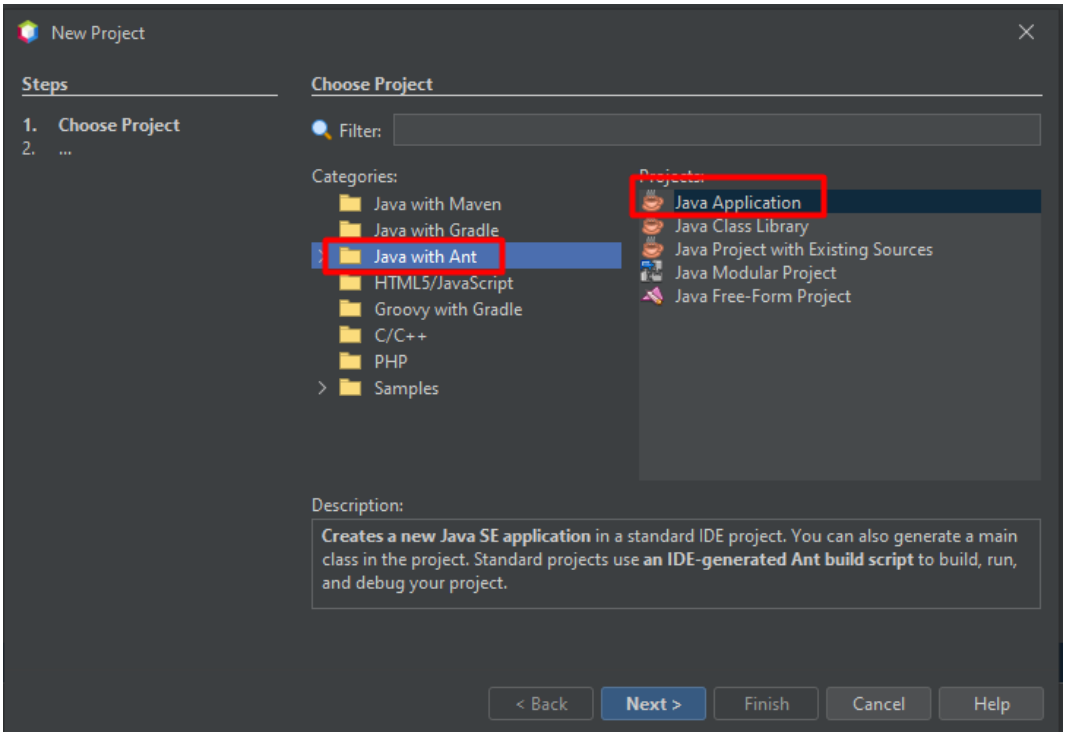
```
@Override
public void start(Stage stage) {
    VBox root = new VBox(10);
    Button btn = new Button("Hola");

    btn.setOnAction(e -> System.out.println("Hola"));
    root.getChildren().add(btn);
    stage.setScene(new Scene(root, 300, 200));
    stage.show();
}
```

JavaFX con FXML

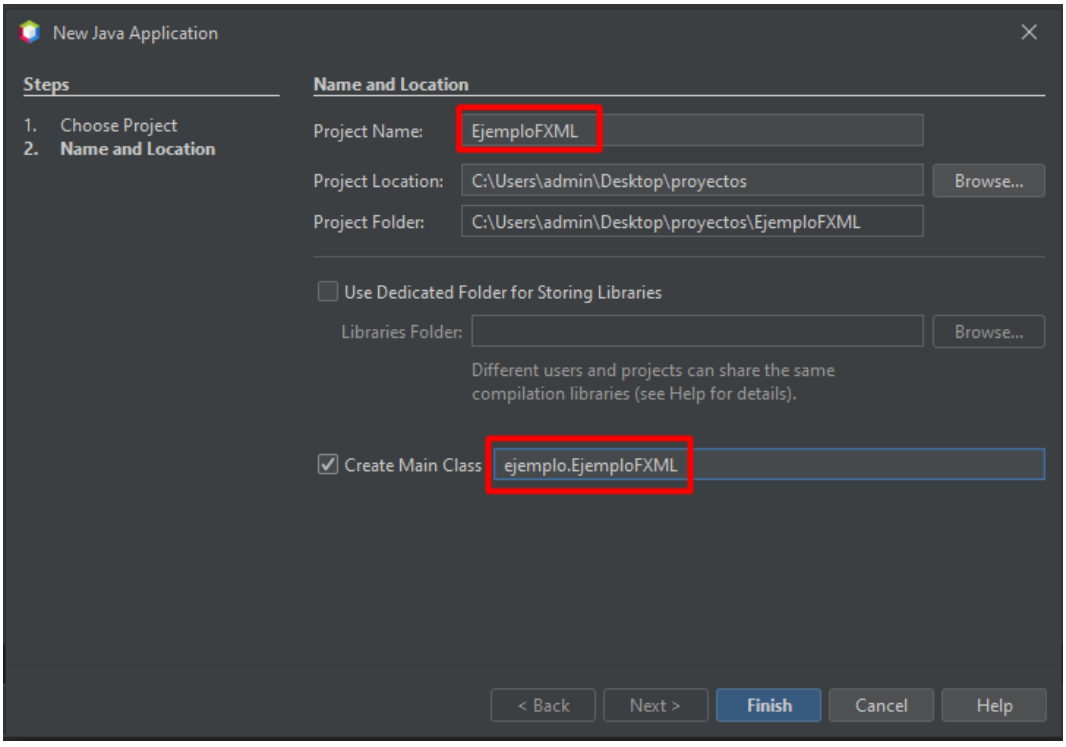
```
@Override
public void start(Stage stage) throws Exception {

    Parent root = FXMLLoader.load(
            getClass().getResource("/vista/PantallaPrincipal.fxml")
    );

    stage.setScene(new Scene(root, 300, 200));
    stage.show();
}
```

2.4. Crear un proyecto FXML

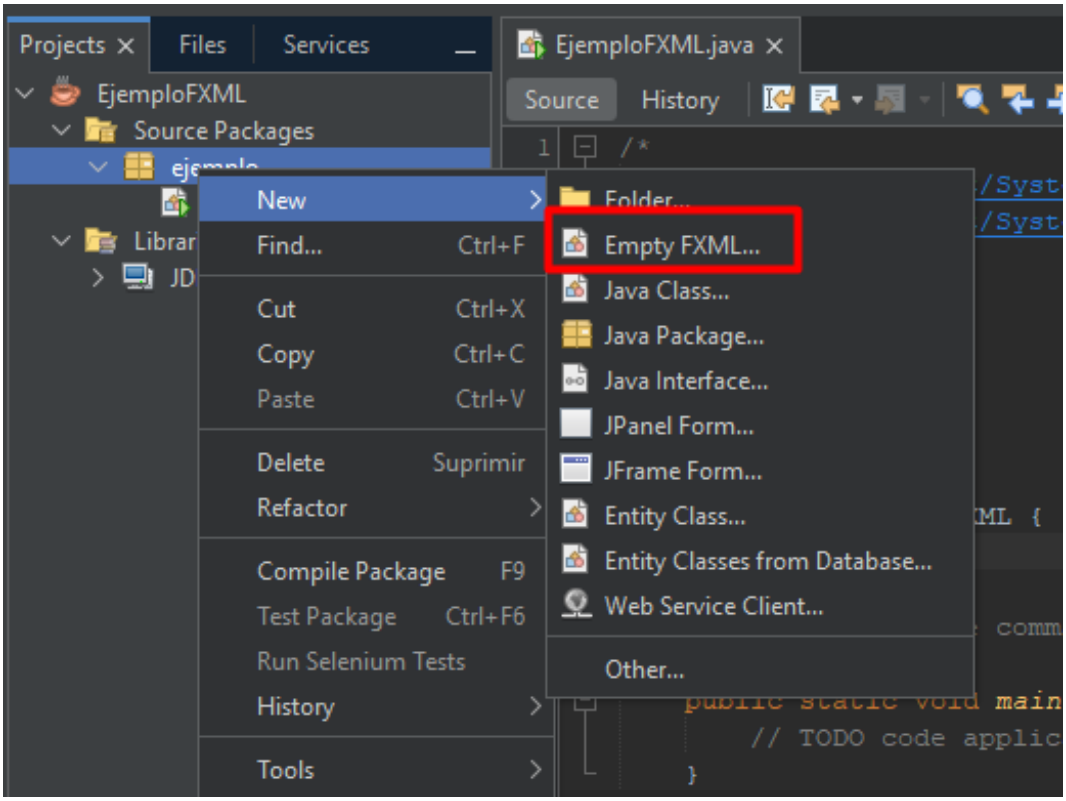
a. Creación del proyecto Java



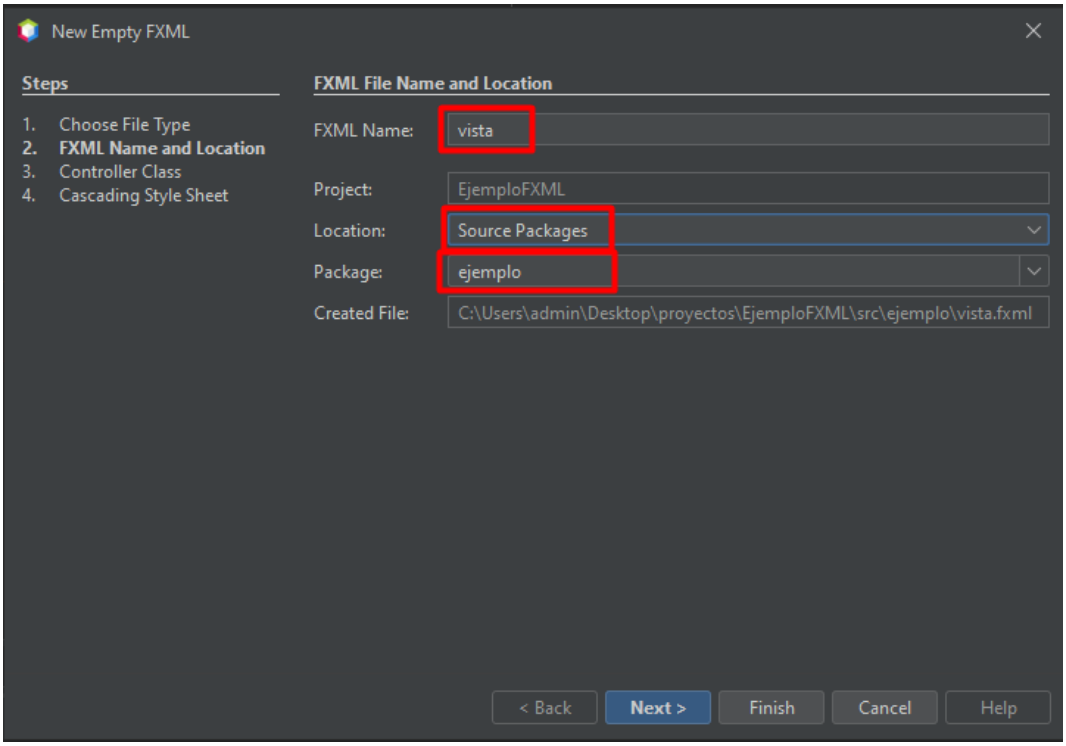
b. Se asigna un nombre al proyecto y un nombre al pequete



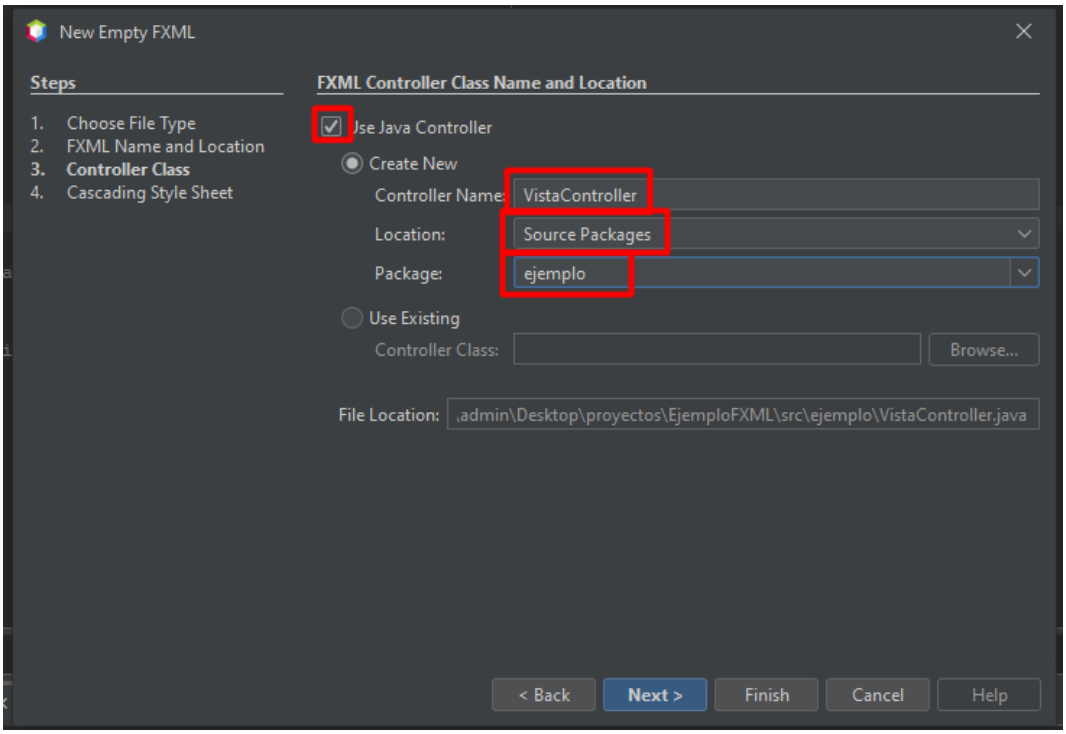
c. Creación del archivo FXML



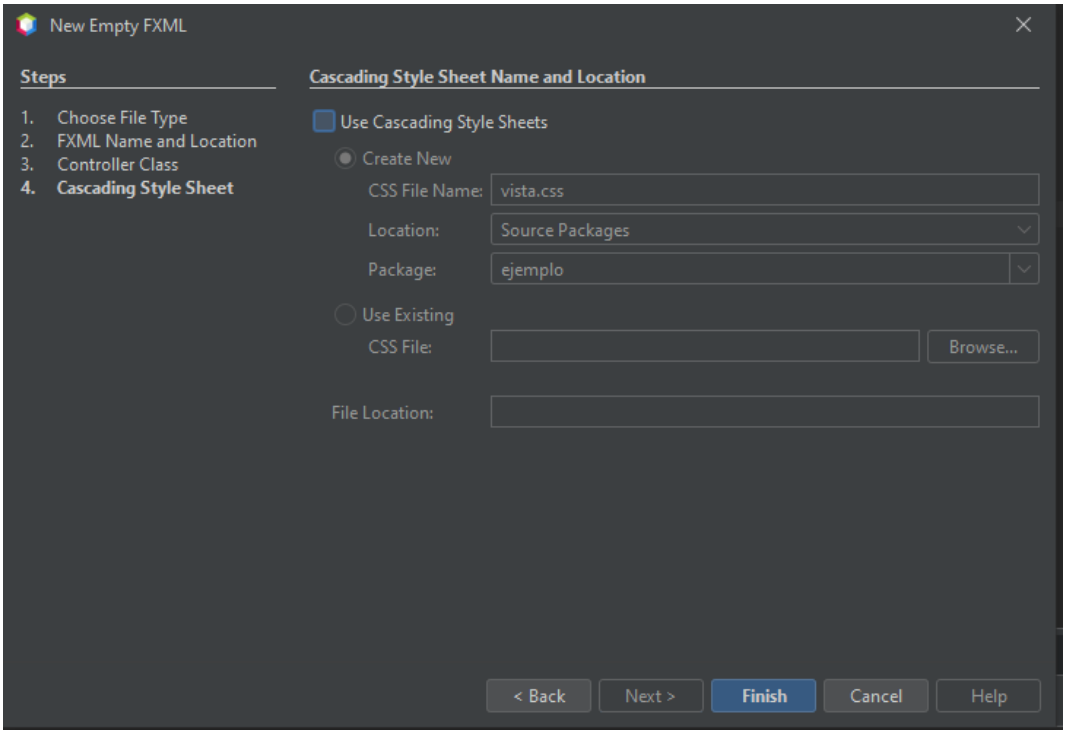
d. Se asigna un nombre a la vista, la ubicación y el paquete en el que se encuentra



e. Se asigna una clase que actúe de controlador (en caso de que ya exista una clase, se marca la opción de "Use Existing")



f. En caso de ser necesario se crea/asigna un archivo CSS para los estilos



3. Informes con gráficos (Charts)

En una aplicación de gestión, un **informe** no es solo una tabla con datos: es una **presentación estructurada** de información que permite **entender, comparar y tomar decisiones**. En JavaFX, una forma muy directa de "dar valor" a un informe es incorporar **gráficos** que visualicen tendencias, comparaciones o distribuciones.

3.1. ¿Qué es un gráfico en JavaFX y para qué se usa en informes?

Un **gráfico** es una representación visual de datos. Los gráficos proporcionan una forma más sencilla de analizar grandes volúmenes de información de manera visual. Habitualmente se utilizan con fines de **seguimiento (monitorización)** y **elaboración de informes**.

Existen distintos tipos de gráficos, que se diferencian en la forma en la que representan los datos. Por ejemplo:

- Un **gráfico de líneas** es apropiado para comprender la evolución o tendencia comparativa de los datos.
- Un **gráfico de barras** resulta más adecuado para comparar valores entre diferentes categorías.

JavaFX incluye el paquete **`javafx.scene.chart`**, que proporciona componentes para **visualización de datos**, con creación sencilla y un alto nivel de personalización.

En la jerarquía de clases, la clase abstracta **`Chart`** es la clase base de todos los gráficos. Esta clase hereda de **`Node`**, lo que significa que los gráficos:

- Pueden añadirse directamente al grafo de escena.
- Pueden estilizarse mediante CSS, igual que cualquier otro nodo JavaFX.

La clase Chart contiene propiedades y métodos comunes a todos los tipos de gráficos. Todo gráfico en JavaFX está compuesto por tres elementos fundamentales:

- Título
- Leyenda
- Contenido (o datos representados)

Aunque cada tipo de gráfico define sus datos de forma específica, todos comparten una serie de propiedades comunes definidas en la clase Chart.

a. Propiedades comunes de la clase Chart

Las principales propiedades comunes a todos los gráficos son:

- **`title`**: establece el título del gráfico.
- **`titleSide`**: indica la posición del título.
- **`legend`**: define el nodo que representa la leyenda.
- **`legendSide`**: indica la posición de la leyenda.
- **`legendVisible`**: determina si la leyenda es visible.
- **`animated`**: indica si los cambios en el gráfico se muestran con animación.

b. Título del gráfico

La propiedad **`title`** define el texto del título del gráfico.
La propiedad **`titleSide`** especifica la posición del título. Por defecto, el título se sitúa en la parte superior del gráfico.

Los valores posibles pertenecen al enumerado **`Side`**:

- TOP (valor por defecto)
- RIGHT
- BOTTOM
- LEFT

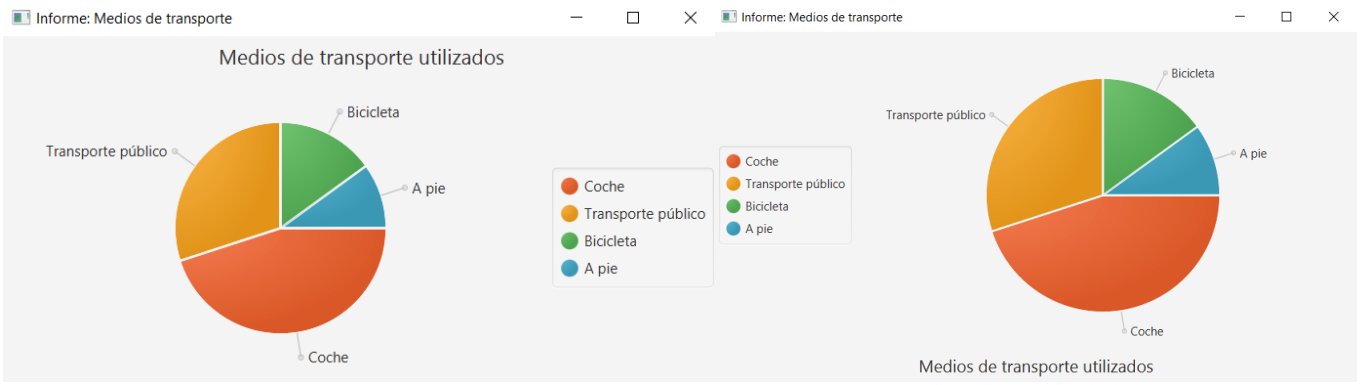
c. Leyenda del gráfico

Normalmente, un gráfico utiliza diferentes símbolos o colores para representar distintas categorías de datos. La **leyenda** muestra estos símbolos junto con su descripción.

- La propiedad **`legend`** es un Node que representa la leyenda del gráfico.
- La propiedad **`legendSide`** permite cambiar su posición (TOP, RIGHT, BOTTOM, LEFT). Por defecto, la leyenda se muestra debajo del contenido del gráfico.
- La propiedad **`legendVisible`** indica si la leyenda se muestra o no. Por defecto, es visible.

d. Animaciones en los gráficos

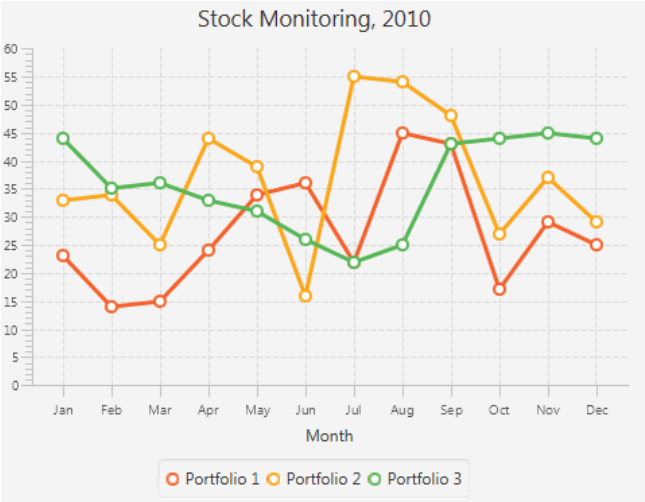
La propiedad **`animated`** indica si los cambios en los datos del gráfico se muestran mediante animaciones. Por defecto, esta propiedad está activada (true), lo que hace que la aparición o actualización de los datos sea más visual y atractiva.



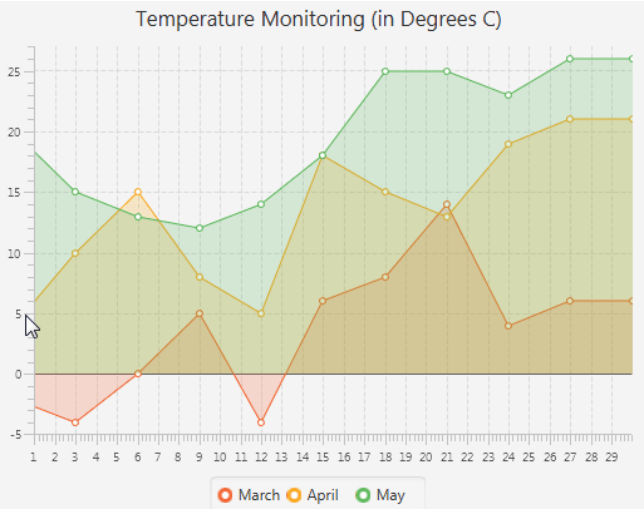
3.2. Tipos de gráficos principales y cuándo usarlos

JavaFX incluye una serie de gráficos estándar que cubren la mayoría de necesidades de **visualización de datos en informes**. Elegir el tipo de gráfico adecuado es fundamental para que la información sea clara y fácil de interpretar:

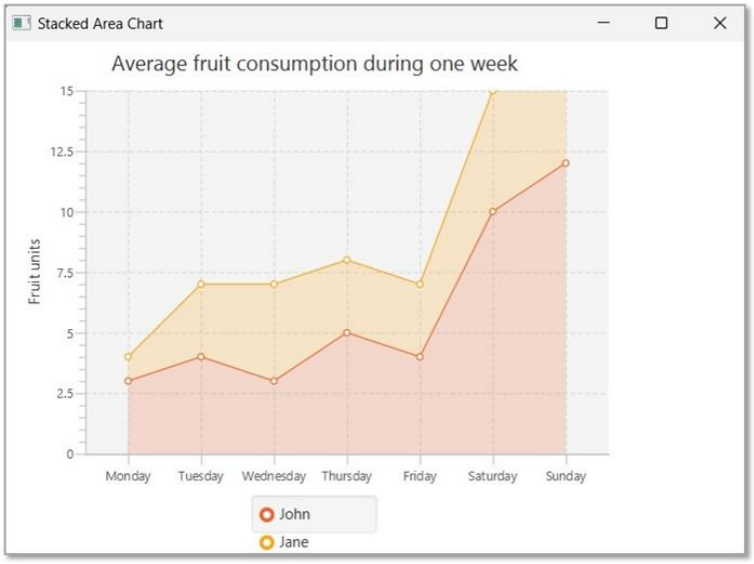
- **LineChart**
  - Representa los datos mediante líneas que unen puntos.
  - Ideal para **analizar tendencias en el tiempo** o la evolución de una variable.



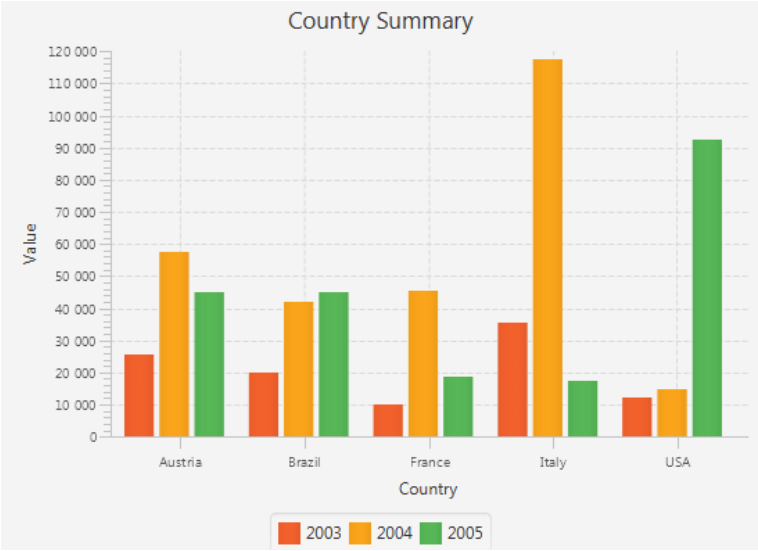
- **AreaChart**
  - Similar al LineChart, pero rellena el área bajo la curva.
  - Útil para **mostrar acumulados** o comparar el impacto total de varias series a lo largo del tiempo.



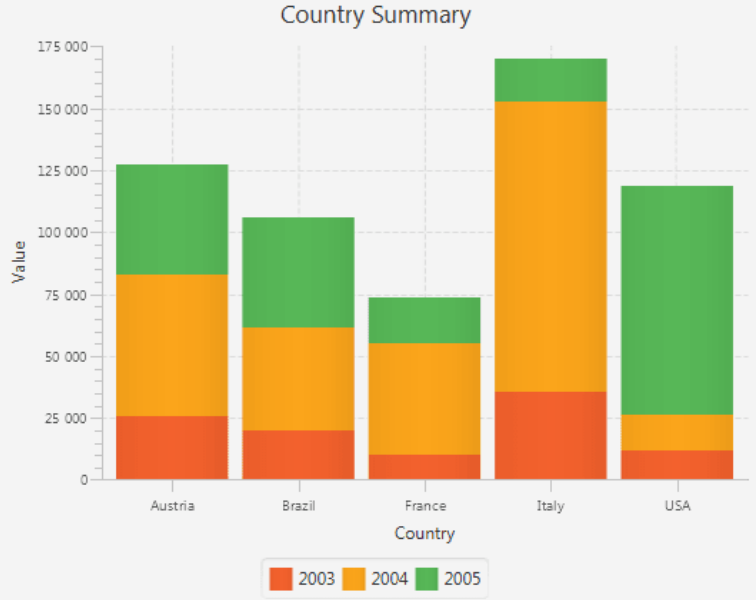
- **StackedAreaChart**
  - Variante del AreaChart donde las áreas **se apilan**.
  - Ideal para visualizar cómo contribuyen distintas series al total acumulado a lo largo del tiempo.



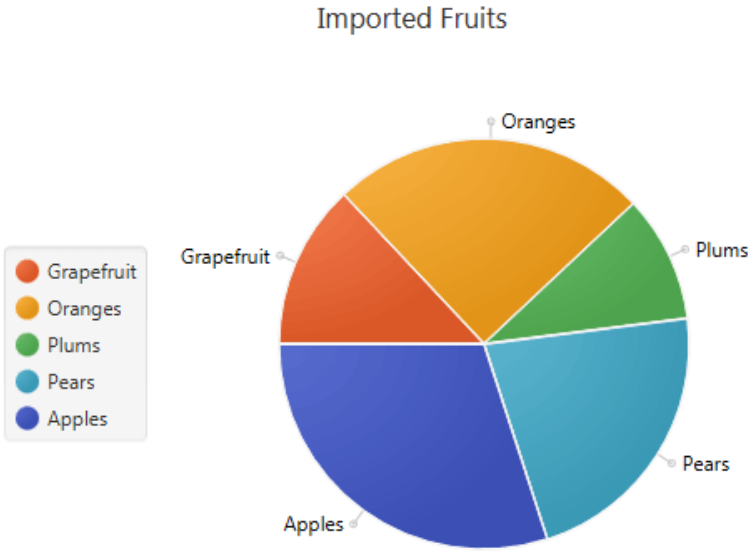
- **BarChart**
  - Representa los datos mediante barras separadas por categorías.
  - Especialmente indicado para **datos discretos** y comparaciones directas entre categorías.



- **StackedBarChart**
  - Variante del BarChart donde las barras **se apilan unas sobre otras.**
  - Permite ver el **total por categoría y la contribución de cada subcategoría**.

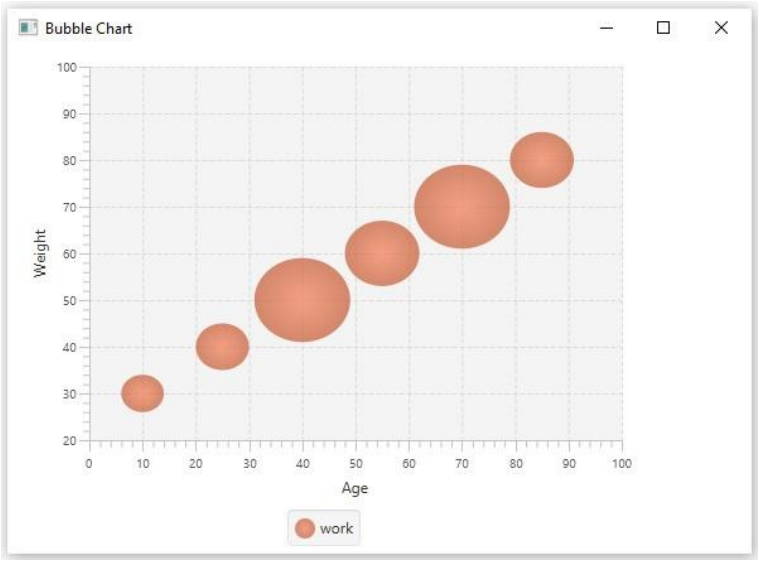


- **PieChart**
  - Representa los datos como **porciones de un total**.
  - Útil para mostrar proporciones o porcentajes.
  - Recomendación: no usar más de **5–7 segmentos** para mantener la legibilidad.



- **ScatterChart**
  - Representa los datos como una **nube de puntos**.
  - Muy útil para analizar **distribución**, **relaciones** y **correlaciones** entre variables.

- **BubbleChart**
  - Similar al ScatterChart, pero añade un **tercer valor** representado por el tamaño del punto.
  - Permite analizar **tres variables simultáneamente** (x, y, tamaño).



4. Jerarquía de clases de Chart

4.1. Clase Base

`Chart` es la clase base de todos los gráficos. Se encarga de dibujar elementos comunes como **fondo, marco, título y leyenda**, y se puede extender para crear gráficos personalizados.

`XYChart<X,Y>` es la clase base de los gráficos **con dos ejes** (X e Y). Hereda de `Chart` y se encarga de los **dos ejes** y del **área de trazado (plot)**.

La mayoría de gráficos extienden `XYChart`, **excepto `PieChart`**, que extiende directamente de `Chart` porque **no tiene ejes**.

Qué implica esto:

- Si el gráfico tiene ejes (Line/Area/Bar/Scatter/Bubble/Stacked…), se usará el modelo de datos de `XYChart`.
- Si es `PieChart`, se usará su propio tipo de datos.

4.2. Ejes

En los **gráficos de dos ejes**, el **tipo de las coordenadas** que se utilizan para representar los datos está directamente condicionado por el **tipo de eje** que se emplea.

La clase **`Axis<T>`** es la clase abstracta base de todos los ejes. A partir de ella se derivan los distintos tipos de ejes que permiten representar la información de formas diferentes:

- **`CategoryAxis`** se utiliza para representar **categorías**, es decir, valores de tipo texto. Cada valor del eje corresponde a una categoría única, como por ejemplo nombres de meses (*"Ene"*, *"Feb"*) o zonas geográficas (*"Norte"*, *"Sur"*).

  Cuando se emplea este eje, lo habitual es que el tipo genérico del eje X sea **`String`**.

- **`NumberAxis`** se utiliza para representar **rangos numéricos**. Este eje muestra marcas principales (*tick marks*) separadas por un intervalo definido mediante la propiedad `tickUnit`.

Como **regla práctica**, en la mayoría de aplicaciones:

- Si el eje X contiene texto → se usa **`CategoryAxis`** → `XYChart<String, Number>`.
- Si el eje X contiene valores numéricos → se usa **`NumberAxis`** → `XYChart<Number, Number>`.
- El eje Y casi siempre es **`NumberAxis`**, por lo que el tipo Y suele ser `Number`.

Ejemplo de creación de Axis

```
// ----------------------------
// Creación de los ejes
// ----------------------------
CategoryAxis xAxis = new CategoryAxis();  // Eje X categórico (meses)
NumberAxis yAxis = new NumberAxis();      // Eje Y numérico (MWh)

// Etiquetas descriptivas para los ejes
xAxis.setLabel("Mes");
yAxis.setLabel("MWh");

// ----------------------------
// Aplicación de los ejes al gráfico
// ----------------------------
LineChart<String, Number> chart = new LineChart<>(xAxis, yAxis);
```

a. Formato y presentación de los valores en los ejes

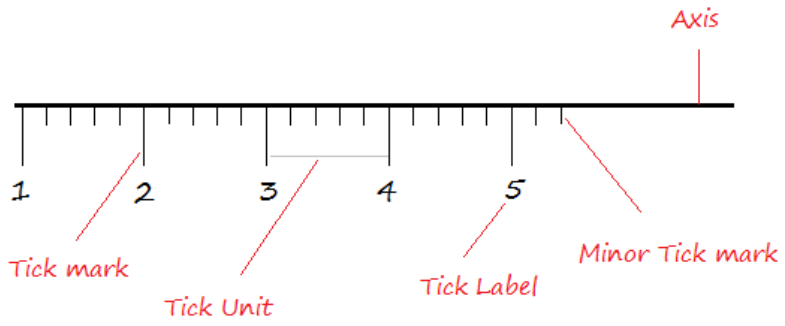
El **JavaFX SDK** proporciona varios métodos para **ajustar la apariencia de los valores mostrados en los ejes**, permitiendo controlar aspectos como:

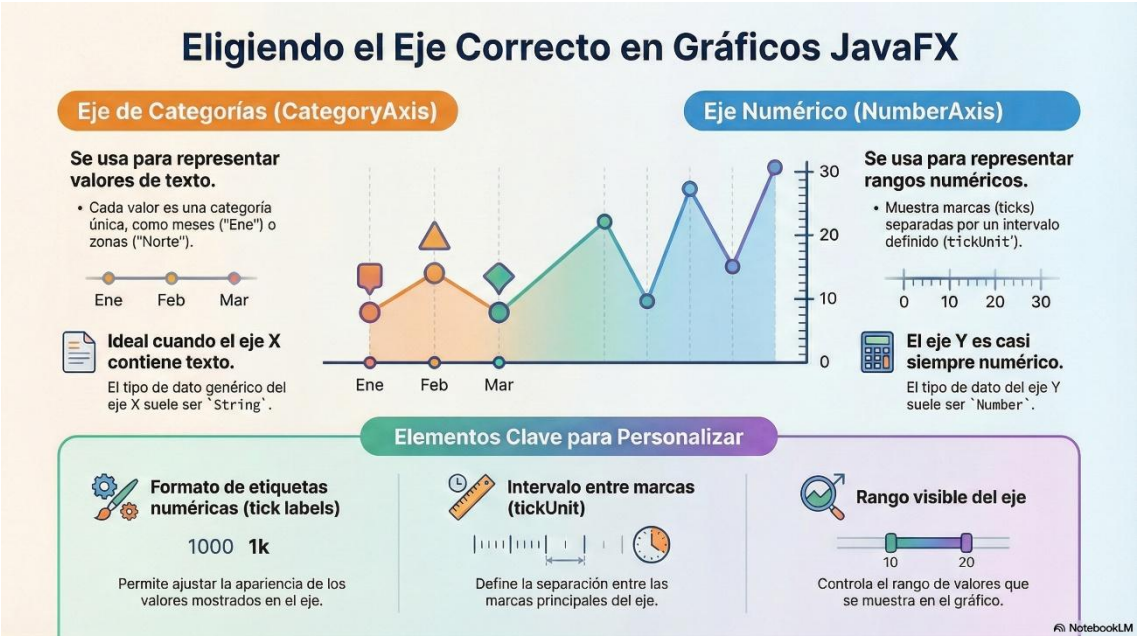
- el formato de las etiquetas numéricas (*tick labels*),
- el intervalo entre marcas (*tickUnit*),
- y el rango visible del eje.

Los ejes de un gráfico están formados por elementos clave como:

- **tick marks** (marcas),
- **tick labels** (etiquetas numéricas),
- y el rango de valores representado.

Mediante la configuración adecuada de estos elementos, es posible adaptar la presentación de los datos para que resulte **más clara, legible y acorde al dominio del problema**, mejorando notablemente la experiencia de usuario.





4.3. Modelo de datos en gráficos con ejes

La clase **`XYChart`** es la **superclase de todos los gráficos de dos ejes** proporciona las capacidades básicas necesarias para construir gráficos como **LineChart, AreaChart, BarChart, ScatterChart, BubbleChart** y sus variantes apiladas (*Stacked*). Estos gráficos comparten un mismo **modelo de datos**, independientemente de cómo se representen visualmente.

Para definir los datos que se van a representar en este tipo de gráficos se utiliza la clase **`XYChart.Data`**, que actúa como unidad básica del modelo de datos. Cada objeto `XYChart.Data` contiene:

- un **valor X** (`xValue`), que determina la posición del elemento en el eje horizontal,
- un **valor Y** (`yValue`), que determina la posición en el eje vertical.

Además, `XYChart.Data` permite definir un **valor adicional** (*extraValue*), que puede emplearse de distintas formas según el tipo de gráfico o simplemente para almacenar información extra asociada al punto. Un ejemplo claro de este uso es el **`BubbleChart`**, donde el `extraValue` se utiliza para determinar el **tamaño (radio) de la burbuja**.

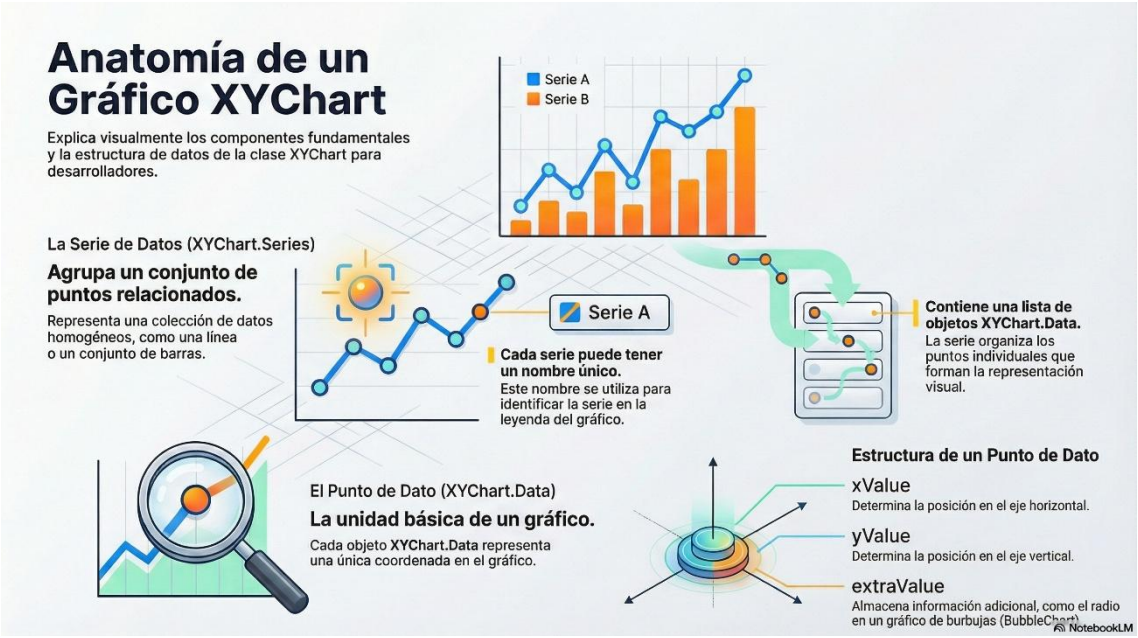
En los gráficos de dos ejes es habitual representar **varias series de datos** simultáneamente. Para ello se emplea la clase **`XYChart.Series<X, Y>`**, que agrupa un conjunto de puntos relacionados entre sí. Cada serie:

- representa un **conjunto de datos homogéneo** dentro del gráfico,
- puede tener un **nombre** asignado mediante **`setName`**, que se mostrará en la **leyenda**,
- contiene una lista de objetos **`XYChart.Data<X, Y>`.**

Es importante tener en cuenta que, si se desea **utilizar el mismo conjunto de datos en dos gráficos diferentes**, **no es posible añadir los mismos objetos XYChart.Data (ni la misma Series) a más de un gráfico a la vez**. JavaFX asocia internamente cada dato a un nodo gráfico, por lo que compartirlos entre gráficos provoca errores. En estos casos, es **necesario clonar los datos o las series**, creando nuevas instancias con los mismos valores, para que cada gráfico disponga de su propio conjunto de objetos.

Ejemplo

```
XYChart.Series<String, Number> prodA = new XYChart.Series<>();
        prodA.setName("Producto A");
        prodA.getData().add(new XYChart.Data<>("Norte", 120));
        prodA.getData().add(new XYChart.Data<>("Sur", 90));
        prodA.getData().add(new XYChart.Data<>("Este", 140));
        prodA.getData().add(new XYChart.Data<>("Oeste", 110));
```



4.4. Modelo de datos en gráficos sin ejes

El **`PieChart`** utiliza un **modelo de datos diferente** al de los gráficos de dos ejes. A diferencia de gráficos como `LineChart`, `BarChart` o `AreaChart`, **no emplea `XYChart.Series` ni `XYChart.Data`**, ya que no representa información en un plano cartesiano con ejes X e Y.

Para definir los datos de un `PieChart` se crean tantos objetos **`PieChart.Data`** como sectores (*slices*) se deseen mostrar. Cada objeto `PieChart.Data` contiene **dos campos fundamentales**:

- el **nombre** del sector, que identifica la categoría,
- el **valor numérico**, que determina el tamaño del sector en relación con el total.

Por defecto, la visualización de un gráfico circular incluye:

- el círculo dividido en sectores,
- las etiquetas asociadas a cada sector,
- y la leyenda del gráfico.

Las etiquetas que aparecen en el gráfico se obtienen directamente del **campo `name`** de cada objeto `PieChart.Data`.

Aunque un sector del `PieChart` no es en sí mismo un objeto `Node`, **cada instancia de `PieChart.Data` tiene un nodo gráfico asociado**. Este nodo puede utilizarse para:

- detectar eventos (por ejemplo, clics del usuario),

- aplicar estilos CSS,
- o ejecutar lógica específica en respuesta a la interacción con el sector.

Cuando es necesario **añadir un conjunto completo de valores** a un PieChart, estos deben agruparse en una **ObservableList<PieChart.Data>**, ya que el constructor del gráfico es: public PieChart(ObservableList<PieChart.Data> data)

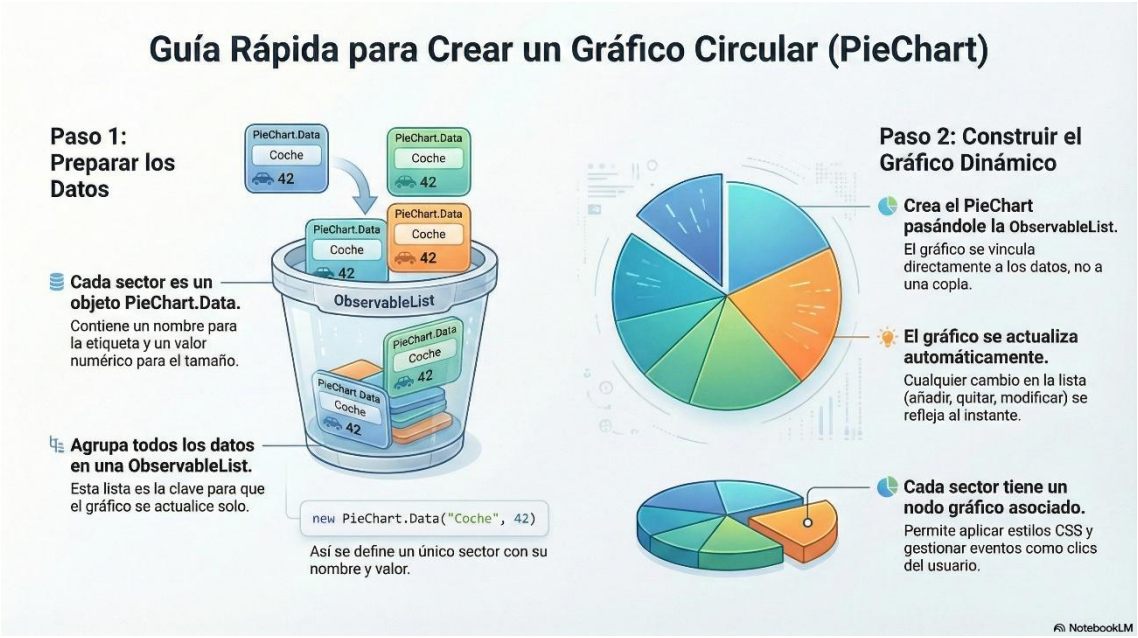
Este constructor crea un nuevo PieChart utilizando directamente la lista proporcionada. Es importante destacar que **la lista pasada es la lista real usada por el gráfico**, no una copia. Esto implica que:

- cualquier **modificación** sobre la ObservableList (añadir, eliminar o modificar datos),
- se reflejará **automáticamente** en el gráfico,
- sin necesidad de volver a crear el PieChart.

Este comportamiento convierte al PieChart en una herramienta especialmente adecuada para **visualizaciones dinámicas**, donde los valores pueden cambiar en tiempo real.

Ejemplo:

```
PieChart chart = new PieChart(
                FXCollections.observableArrayList(
                        new PieChart.Data("Coche", 42),
                              new PieChart.Data("Bicicleta", 15),
                        new PieChart.Data("A pie", 10)
                )
        );
```

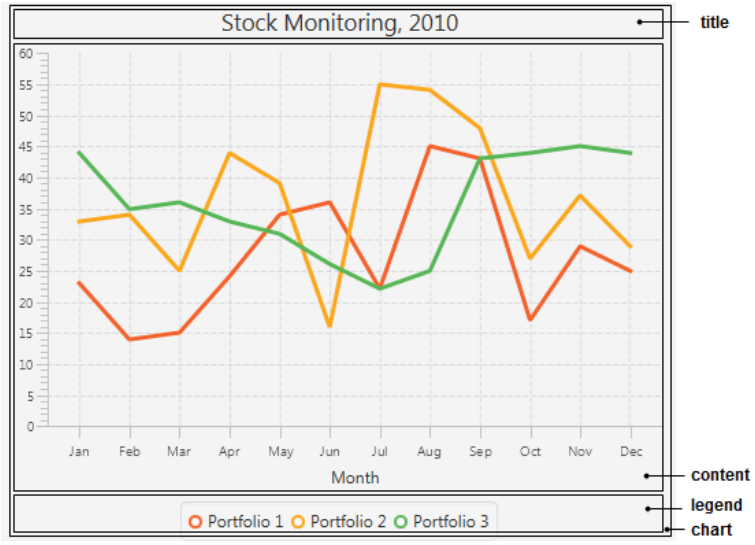


5. Aplicar estilos a gráficos con CSS

JavaFX permite **modificar completamente la apariencia de los gráficos** mediante **CSS**, sin necesidad de cambiar el código Java. Aunque la API de JavaFX ofrece algunos métodos para cambiar colores o textos, **Oracle recomienda usar CSS** para personalizar los gráficos, ya que:

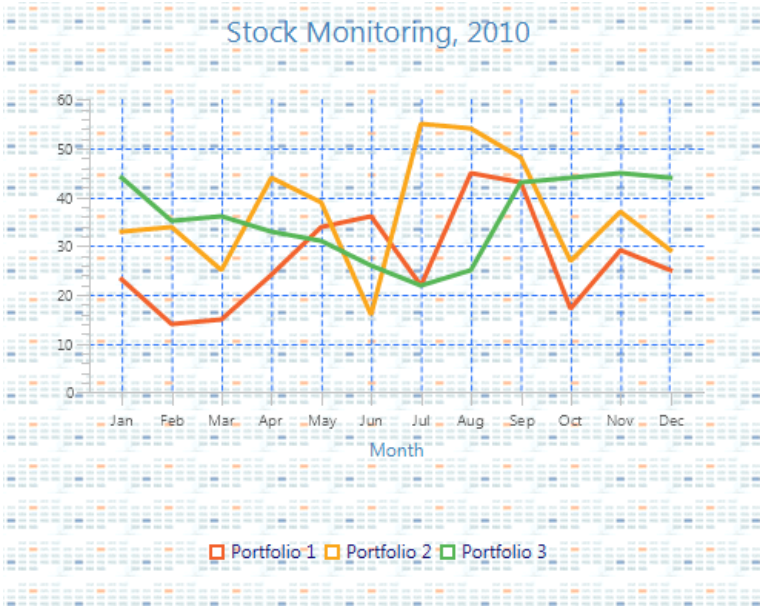
- ofrece mayor control visual,
- mantiene separada la lógica del diseño,
- facilita la reutilización de estilos,
- permite crear una identidad visual propia para la aplicación.

5.1. Elementos genéricos editables de un Chart



Elemento del gráfico	Clase(s) CSS	Qué se puede modificar	Ejemplo CSS
Contenedor del gráfico	`.chart`	Padding, fondo, imagen de fondo	`-fx-padding: 10px;` `-fx-background-image: url("icon.png");`
Contenido del gráfico	`.chart-content`	Padding interno del área del gráfico	`-fx-padding: 30px;`

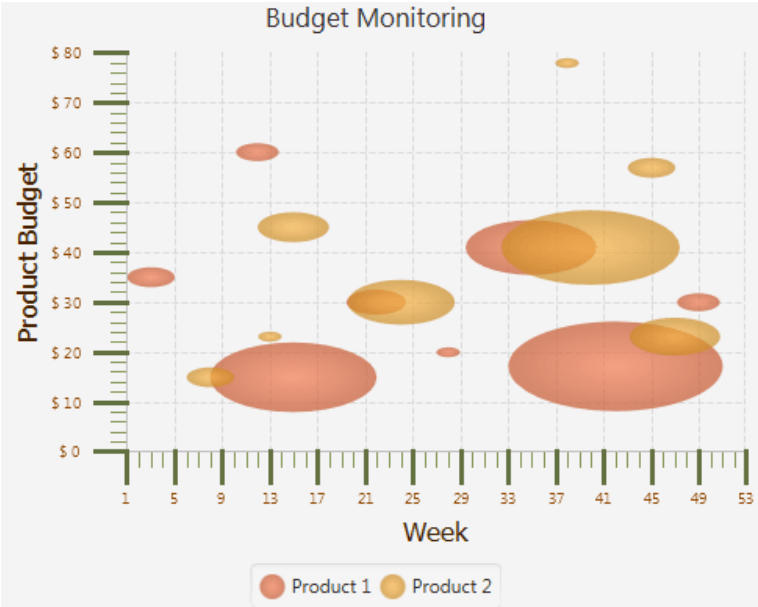
Título del gráfico	`.chart-title`	Color, tamaño y fuente del texto	`-fx-text-fill: #4682b4;` `-fx-font-size: 16px;`
Leyenda	`.chart-legend`	Fondo, padding, borde	`-fx-background-color: transparent;` `-fx-padding: 20px;`
Texto de la leyenda	`.chart-legend-item`	Color del texto	`-fx-text-fill: #191970;`
Símbolos de la leyenda	`.chart-legend-item-symbol`	Forma (círculo/cuadrado), tamaño	`-fx-background-radius: 0;`
Área de trazado (plot)	`.chart-plot-background`	Color o transparencia del fondo	`-fx-background-color: #e2ecfe;`
Filas alternativas	`.chart-alternative-row-fill`	Color, opacidad	`-fx-fill: #99bcfd;`
Cuadrícula vertical	`.chart-vertical-grid-lines`	Color y grosor	`-fx-stroke: #3278fa;`
Cuadrícula horizontal	`.chart-horizontal-grid-lines`	Color y grosor	`-fx-stroke: #3278fa;`
Ejes (general)	`.axis`	Fuente, tamaño, color de etiquetas	`-fx-font-size: 1.4em;`
Etiqueta del eje	`.axis-label`	Color del texto	`-fx-text-fill: #462300;`
Marcas principales	`.axis-tick-mark`	Color y grosor	`-fx-stroke-width: 3;`
Marcas secundarias	`.axis-minor-tick-mark`	Color	`-fx-stroke: #859656;`



Ejemplo CSS elementos genéricos de un Chart

```
.chart {
    -fx-padding: 10px;
    -fx-background-image: url("icon.png");
}
.chart-content {
    -fx-padding: 30px;
}

.chart-title {
  -fx-text-fill: #4682b4;
  -fx-font-size: 1.6em;
}

.axis-label {
  -fx-text-fill: #4682b4;
}

.chart-legend {
   -fx-background-color:  transparent;
   -fx-padding: 20px;
}

.chart-legend-item-symbol{
   -fx-background-radius: 0;
}

.chart-legend-item{
    -fx-text-fill: #191970;
}
```

```
.chart-plot-background {
    -fx-background-color: transparent;
}
.chart-vertical-grid-lines {
    -fx-stroke: #3278fa;
}
.chart-horizontal-grid-lines {
    -fx-stroke: #3278fa;
}
.chart-alternative-row-fill {
    -fx-fill: transparent;
    -fx-stroke: transparent;
    -fx-stroke-width: 0;
}
```

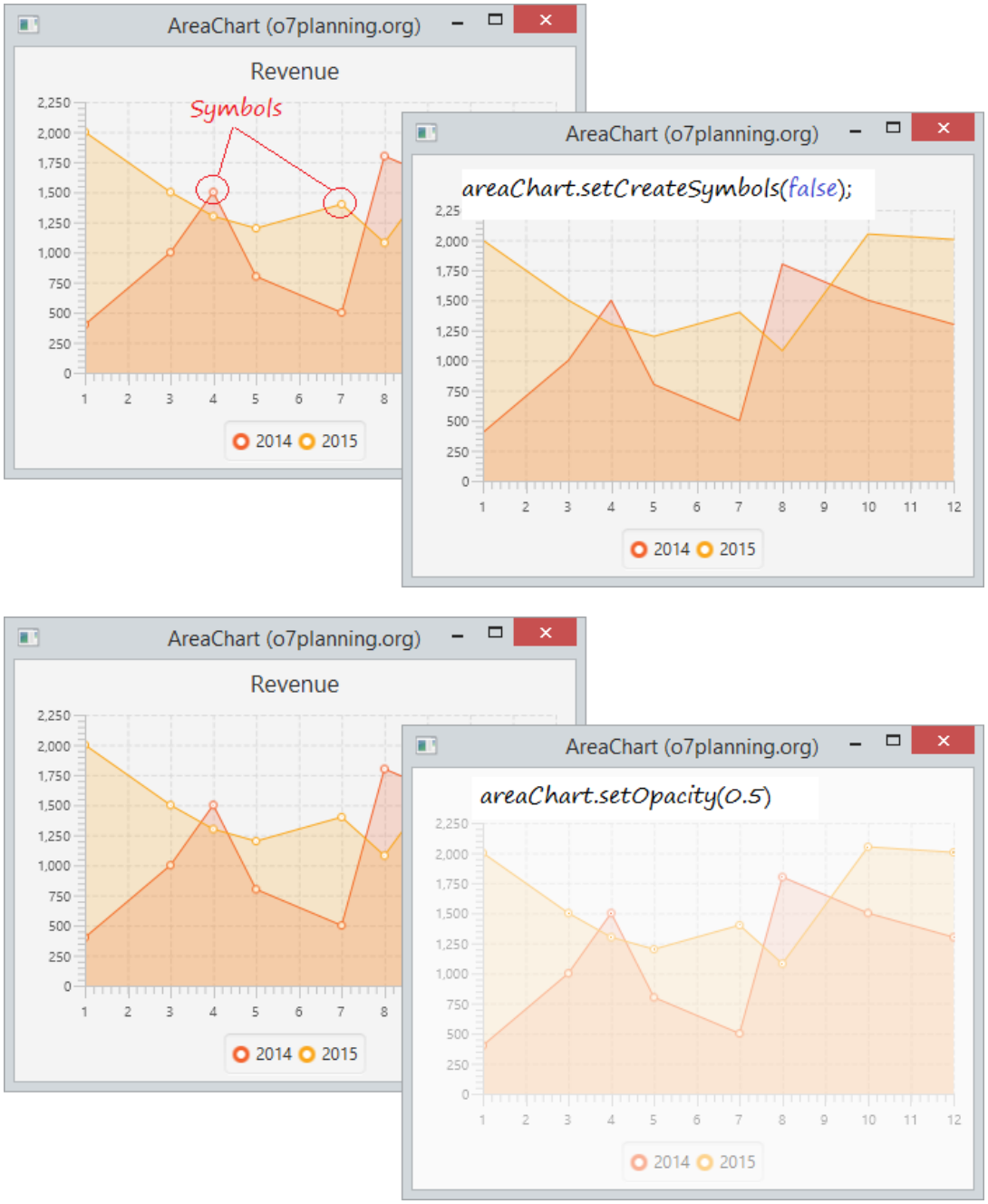


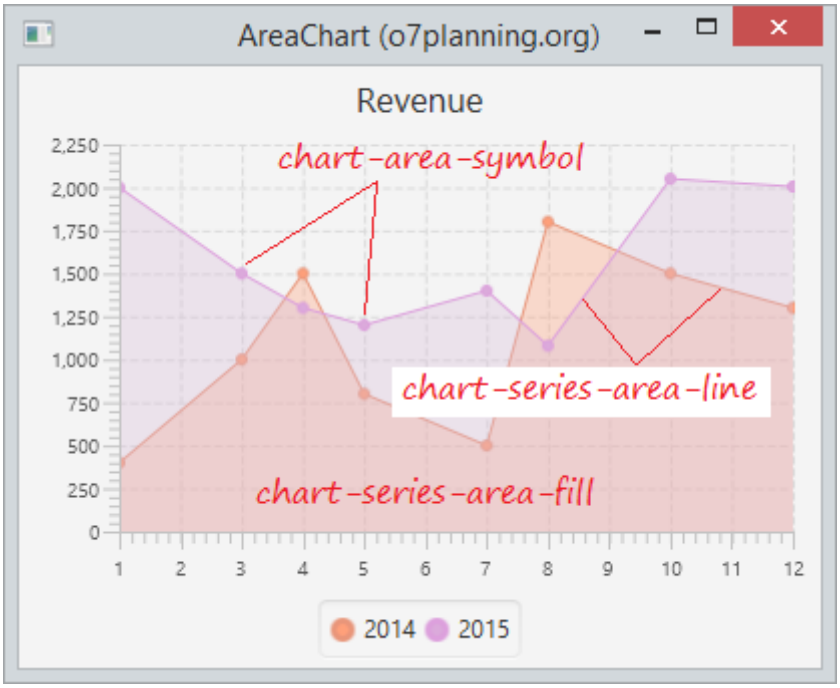
Ejemplo CSS características de los ejes de un Chart

```
.axis {
    -fx-font-size: 1.4em;
    -fx-tick-label-fill: #914800;
    -fx-font-family: Tahoma;
    -fx-tick-length: 20;
    -fx-minor-tick-length: 10;
}

.axis-label {
   -fx-text-fill: #462300;
}

.axis-tick-mark {
```

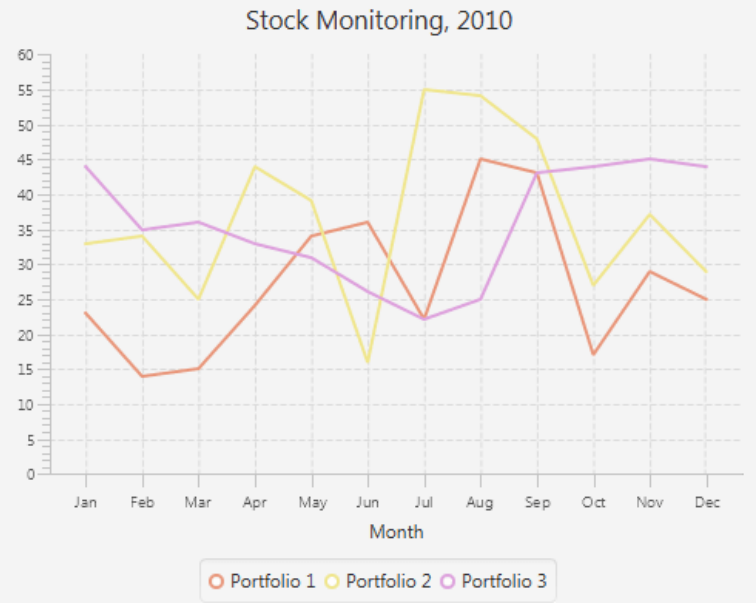
```
    -fx-stroke: #637040;
    -fx-stroke-width: 3;
}
.axis-minor-tick-mark {
    -fx-stroke: #859656;
}
```

5.2. Edición de un LineChart y AreaChart



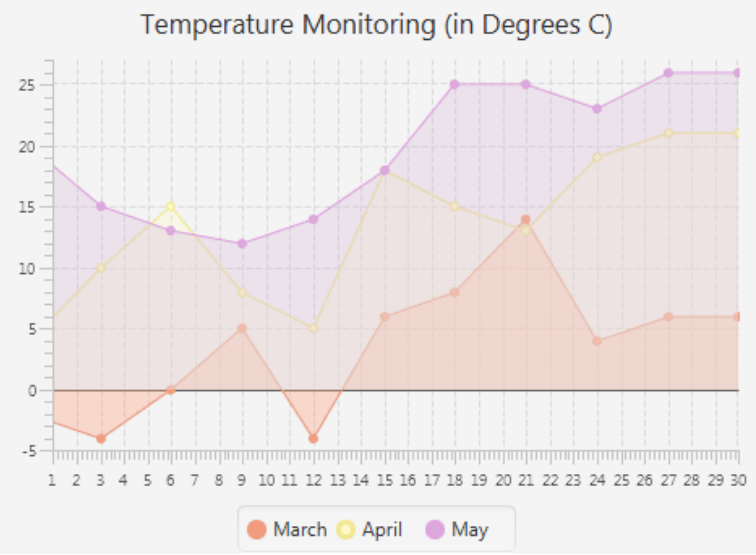


Elemento del gráfico	Clase(s) CSS	Qué se puede modificar	Ejemplo CSS
Líneas (LineChart)	`.chart-series-line`	Color, grosor, efectos	`-fx-stroke-width: 2px;`
Línea por serie	`.default-colorX.chart-series-line`	Color de cada serie	`-fx-stroke: #e9967a;`
Símbolos de línea	`.default-colorX.chart-line-symbol`	Color, forma, borde	`-fx-background-color: red, white;`
Área (AreaChart)	`.chart-series-area-fill`	Color y opacidad	`-fx-fill: rgba(255,160,122,0.3);`
Línea del área	`.chart-series-area-line`	Color de contorno	`-fx-stroke: #e9967a;`
Símbolo del área	`.chart-area-symbol`	Forma (círculo/cuadrado)	`-fx-background-radius: 0;`



Ejemplo CSS características de los LineChart

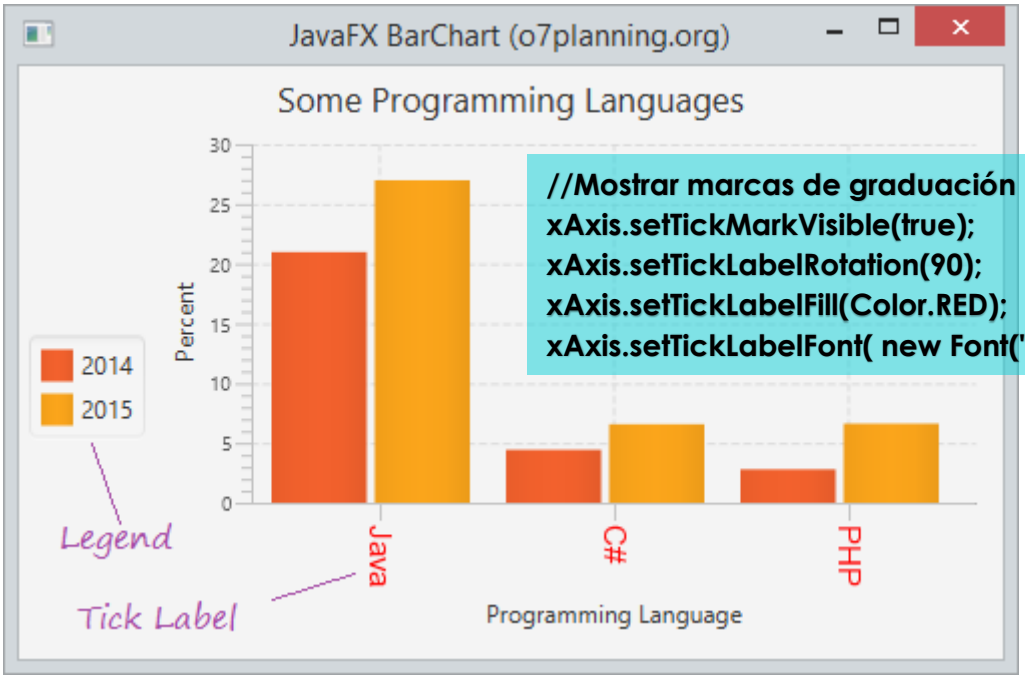
```
.chart-series-line {
    -fx-stroke-width: 2px;
    -fx-effect: null;
}

.default-color0.chart-series-line { -fx-stroke: #e9967a; }
.default-color1.chart-series-line { -fx-stroke: #f0e68c; }
.default-color2.chart-series-line { -fx-stroke: #dda0dd; }

.default-color0.chart-line-symbol { -fx-background-color: #e9967a, white; }
.default-color1.chart-line-symbol { -fx-background-color: #f0e68c, white; }
.default-color2.chart-line-symbol { -fx-background-color: #dda0dd, white; }
```



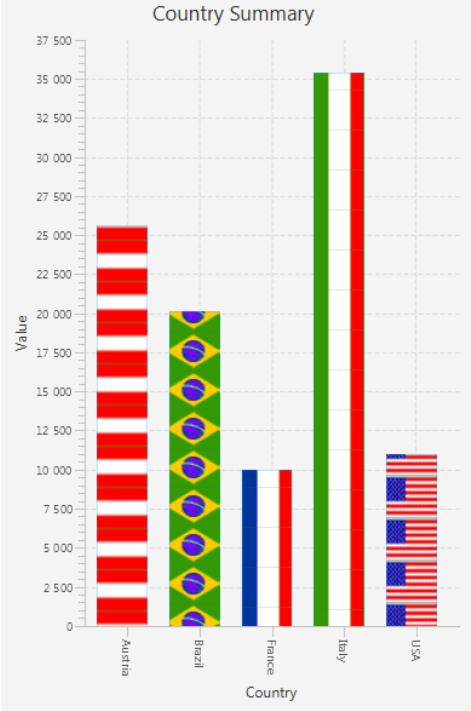
Ejemplo CSS características de los AreaChart

```
.default-color0.chart-area-symbol { -fx-background-color: #e9967a, #ffa07a; }
.default-color1.chart-area-symbol { -fx-background-color: #f0e68c, #fffacd; }
.default-color2.chart-area-symbol { -fx-background-color: #dda0dd, #d8bfd8; }

.default-color0.chart-series-area-line { -fx-stroke: #e9967a; }
.default-color1.chart-series-area-line { -fx-stroke: #f0e68c; }
.default-color2.chart-series-area-line { -fx-stroke: #dda0dd; }

.default-color0.chart-series-area-fill { -fx-fill: #ffa07aaa}
.default-color1.chart-series-area-fill { -fx-fill: #fffacd77; }
.default-color2.chart-series-area-fill { -fx-fill: #d8bfd833; }
```

5.3. Edición de un BarChart



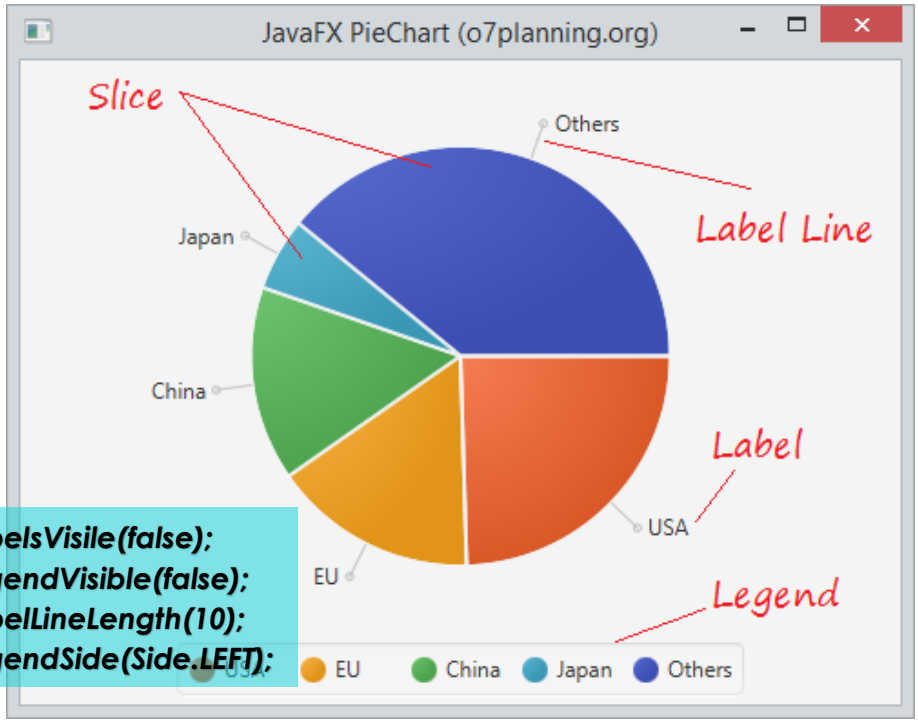
Elemento del gráfico	Clase(s) CSS	Qué se puede modificar	Ejemplo CSS
Barras (BarChart)	`.chart-bar`	Color, gradiente, bordes	`-fx-background-radius: 0;`
Barras por serie	`.dataX.chart-bar`	Imagen o color por serie	`-fx-background-image: url("img.png");`



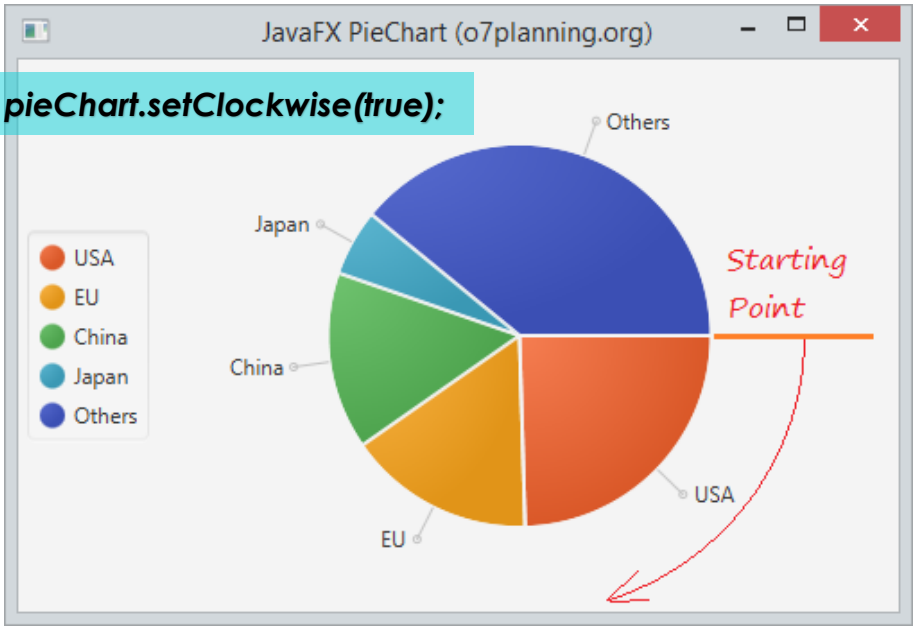
Ejemplo CSS características de los BarChart

```
.chart-bar {
    -fx-background-color: rgba(0,168,355,0.05);
    -fx-border-color: rgba(0,168,355,0.3) rgba(0,168,355,0.3)
        transparent rgba(0,168,355,0.3);
    -fx-background-radius: 0;
    -fx-background-position: left center;
}

.data0.chart-bar {
    -fx-background-image: url("austria.png");
}
.data1.chart-bar {
    -fx-background-image: url("brazil.png");
}
.data2.chart-bar {
    -fx-background-image: url("france.png");
}
.data3.chart-bar {
    -fx-background-image: url("italy.png");
}
.data4.chart-bar {
    -fx-background-image: url("usa.png");
}
```

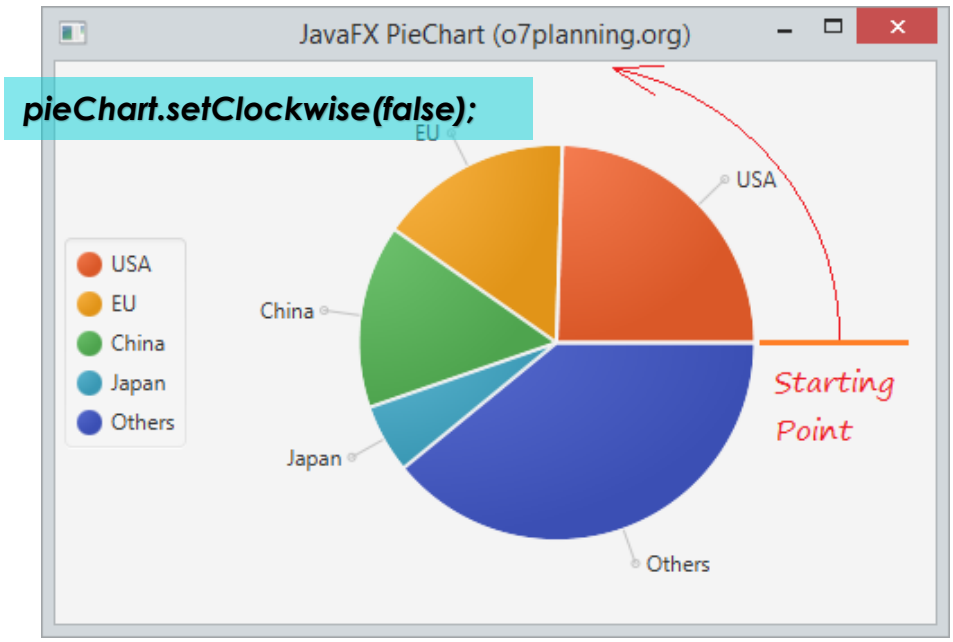
5.4. Edición de un PieChart

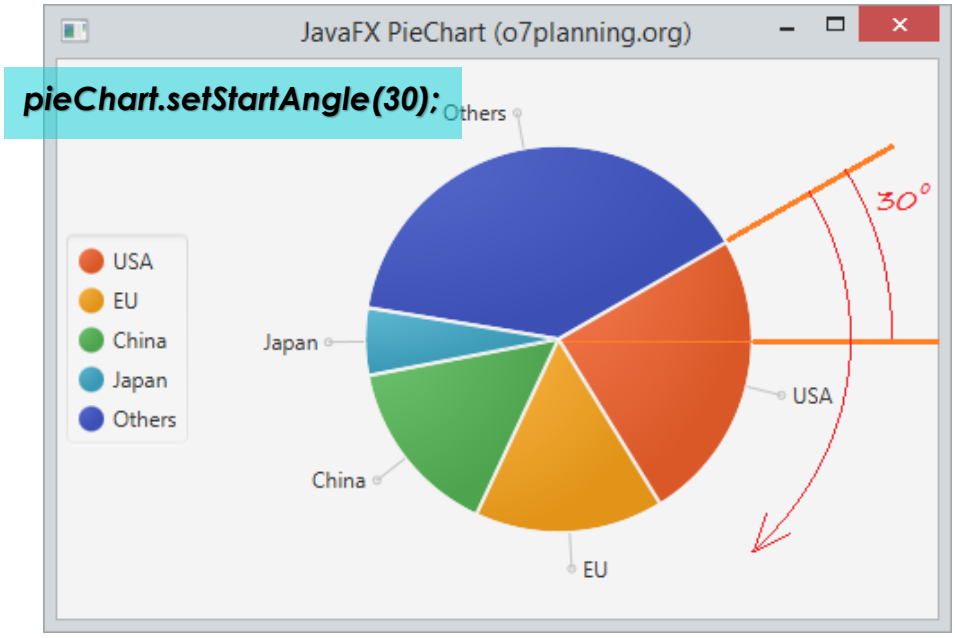



```
pieChart.setLabelsVisile(false);
pieChart.setLegendVisible(false);
pieChart.setLabelLineLength(10);
pieChart.setLegendSide(Side.LEFT);
```



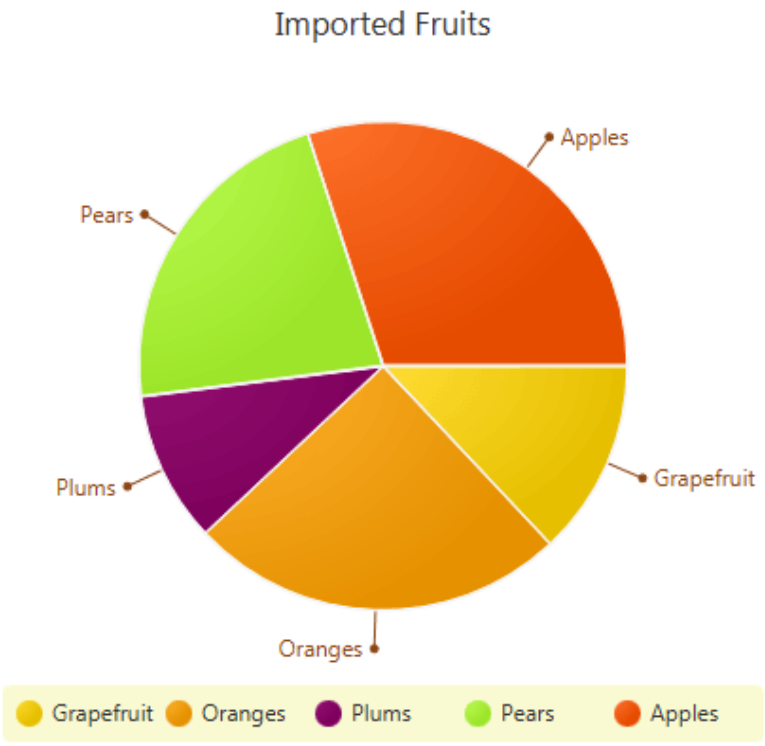

```
pieChart.setClockwise(true);
```





Elemento del gráfico	Clase(s) CSS	Qué se puede modificar	Ejemplo CSS
Sectores (PieChart)	`.chart-pie`	Estilo general del sector	*(base)*
Color por sector	`.default-colorX.chart-pie`	Color de cada porción	`-fx-pie-color: #ffd700;`

Etiquetas del Pie	`.chart-pie-label`	Color y tamaño del texto	`-fx-font-size: 1em;`
Líneas de etiqueta	`.chart-pie-label-line`	Color de la línea	`-fx-stroke: #8b4513;`

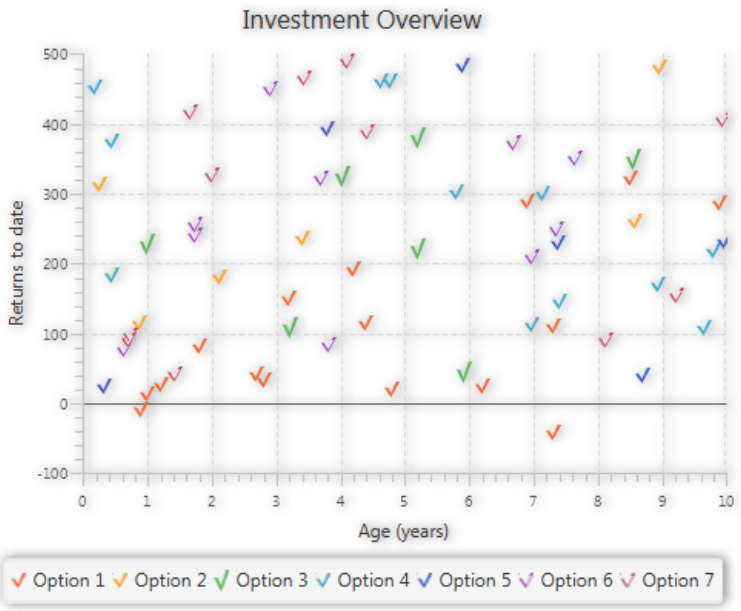


Ejemplo CSS características de los PieChart

```
.default-color0.chart-pie { -fx-pie-color: #ffd700; }
.default-color1.chart-pie { -fx-pie-color: #ffa500; }
.default-color2.chart-pie { -fx-pie-color: #860061; }
.default-color3.chart-pie { -fx-pie-color: #adff2f; }
.default-color4.chart-pie { -fx-pie-color: #ff5700; }

.chart-pie-label-line {
    -fx-stroke: #8b4513;
    -fx-fill: #8b4513;
}

.chart-pie-label {
    -fx-fill: #8b4513;
    -fx-font-size: 1em;
}
.chart-legend {
   -fx-background-color:  #fafad2;
   -fx-stroke: #daa520;
}
```

5.5. Edición de un ScatterChart

Elemento del gráfico	Clase(s) CSS	Qué se puede modificar	Ejemplo CSS
Símbolos (Scatter)	`.chart-symbol`	Forma, tamaño, color	`-fx-shape: "...";`
Símbolo por serie	`.default-colorX.chart-symbol`	Forma y color por serie	`-fx-background-radius: 0;`



Ejemplo CSS características de los ScatterChart

```
.chart-symbol{
    -fx-shape: "M0,4 L2,4 L4,8 L7,0 L9,0 L4,11 Z";
}

.default-color1.chart-symbol { /* solid diamond */
    -fx-background-color: CHART_COLOR_3;
    -fx-background-radius: 0;
    -fx-padding: 7px 5px 7px 5px;
    -fx-shape: "M5,0 L10,9 L5,18 L0,9 Z";
}
```

La propiedad **`-fx-shape`** define la forma del nodo usando un **path SVG**, exactamente igual que en gráficos vectoriales.

JavaFX interpreta esta ruta para dibujar la forma del símbolo.

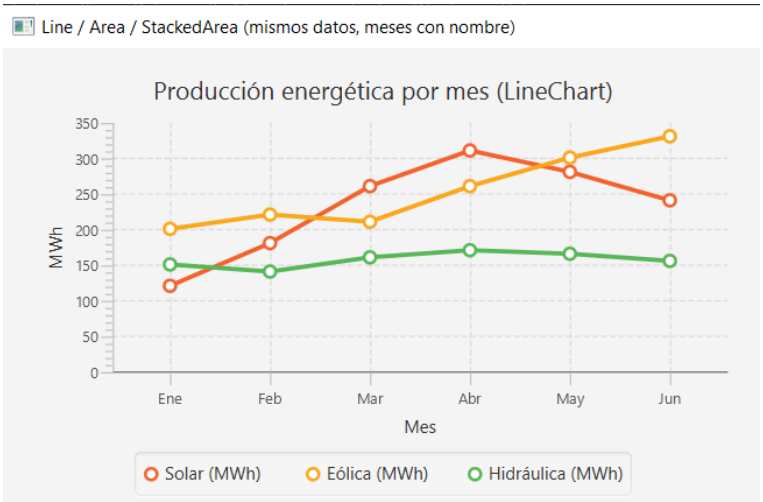
El valor: `"M0,4 L2,4 L4,8 L7,0 L9,0 L4,11 Z"` es una secuencia de comandos SVG:

Comando	Nombre	Coordenadas	Qué hace
`M0,4`	Move to	(0, 4)	Coloca el "lápiz" en el punto indicado. No dibuja nada.
`L2,4`	Line to	(2, 4)	Dibuja una línea desde el punto anterior hasta (2, 4).
`L4,8`	Line to	(4, 8)	Dibuja una línea hasta el punto (4, 8).
`L7,0`	Line to	(7, 0)	Dibuja una línea hasta el punto (7, 0).
`L9,0`	Line to	(9, 0)	Dibuja una línea hasta el punto (9, 0).
`L4,11`	Line to	(4, 11)	Dibuja una línea hasta el punto (4, 11).
`Z`	Close path	—	Cierra la figura uniendo el último punto con el primero.

Visualmente, esta ruta crea una figura similar a: ✔ una marca de verificación (check) o un tick.

6. Ejemplos de gráficos

6.1. LineChart



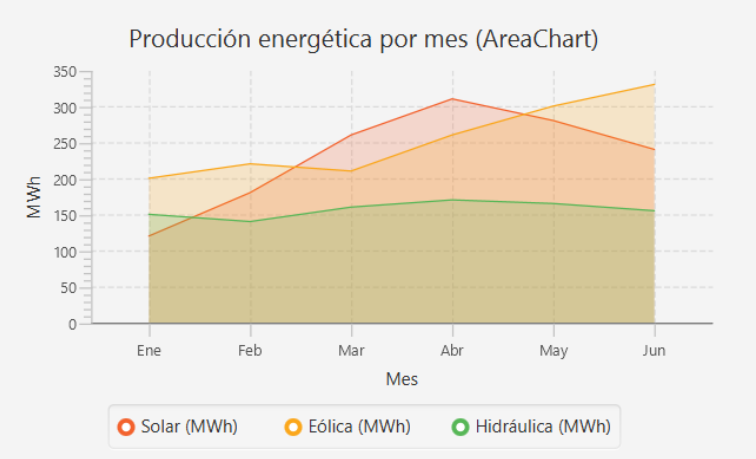
Ejemplo LineChart

```
public class LineChartExample extends Application {

    @Override
    public void start(Stage stage) {

        // ========================
        // 1) Definición del conjunto de datos
        // ========================
        // Serie: Producción solar
        XYChart.Series<String, Number> solar = new XYChart.Series<>();
        solar.setName("Solar (MWh)");
        solar.getData().add(new XYChart.Data<>("Ene", 120));
        solar.getData().add(new XYChart.Data<>("Feb", 180));
        solar.getData().add(new XYChart.Data<>("Mar", 260));
        solar.getData().add(new XYChart.Data<>("Abr", 310));
        solar.getData().add(new XYChart.Data<>("May", 280));
        solar.getData().add(new XYChart.Data<>("Jun", 240));

        // Serie: Producción eólica
        XYChart.Series<String, Number> eolica = new XYChart.Series<>();
        eolica.setName("Eólica (MWh)");
        eolica.getData().add(new XYChart.Data<>("Ene", 200));
        eolica.getData().add(new XYChart.Data<>("Feb", 220));
        eolica.getData().add(new XYChart.Data<>("Mar", 210));
        eolica.getData().add(new XYChart.Data<>("Abr", 260));
        eolica.getData().add(new XYChart.Data<>("May", 300));
```

```
        eolica.getData().add(new XYChart.Data<>("Jun", 330));

        // Serie: Producción hidráulica
        XYChart.Series<String, Number> hidro = new XYChart.Series<>();
        hidro.setName("Hidráulica (MWh)");
        hidro.getData().add(new XYChart.Data<>("Ene", 150));
        hidro.getData().add(new XYChart.Data<>("Feb", 140));
        hidro.getData().add(new XYChart.Data<>("Mar", 160));
        hidro.getData().add(new XYChart.Data<>("Abr", 170));
        hidro.getData().add(new XYChart.Data<>("May", 165));
        hidro.getData().add(new XYChart.Data<>("Jun", 155));

        // =========================
        // 2) Creación del gráfico
        // =========================
        LineChart<String, Number> lineChart = createLineChart(solar, eolica,
hidro);

        // =========================
        // 3) Layout: TilePane
        // =========================
        TilePane root = new TilePane();
        root.setPadding(new Insets(12));

        // Tamaño preferido para que se vean bien en mosaico
        lineChart.setPrefSize(450, 260);
        root.getChildren().addAll(lineChart);

        // =========================
        // 4) Escena y Stage
        // =========================
        stage.setTitle("Line Chart");
        stage.setScene(new Scene(root, 980, 620));
        stage.show();
    }

    // ============================================================
    // MÉTODO: LineChart
    // Crea y configura un gráfico de líneas a partir de tres series
    // ============================================================
    private LineChart<String, Number> createLineChart(
            XYChart.Series<String, Number> s1,
            XYChart.Series<String, Number> s2,
            XYChart.Series<String, Number> s3) {



        // ----------------------------
```

```
        // Creación de los ejes
        // ----------------------------
        // Eje X: categórico, ya que los valores son nombres de meses
        CategoryAxis x = new CategoryAxis();

        // Eje Y: numérico, representa valores de producción en MWh
        NumberAxis y = new NumberAxis();

        // Etiquetas descriptivas para los ejes
        x.setLabel("Mes");
        y.setLabel("MWh");

        // ----------------------------
        // Creación del LineChart
        // ----------------------------
        // El gráfico se construye indicando los ejes X e Y
        LineChart<String, Number> chart = new LineChart<>(x, y);

        // Título que describe el contenido del gráfico
        chart.setTitle("Producción energética por mes (LineChart)");

        // ----------------------------
        // Configuración visual
        // ----------------------------
        // Activa animaciones cuando el gráfico se dibuja o se actualiza
        chart.setAnimated(true);

        // Muestra un símbolo en cada punto de datos
        // Útil para resaltar valores discretos (mes a mes)
        chart.setCreateSymbols(true);

        chart.getData().addAll(s1, s2, s3);

        return chart;
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

6.2. AreaChart



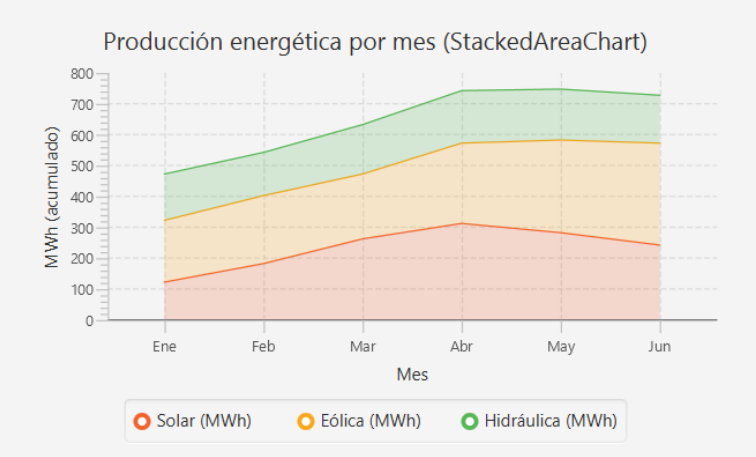
Ejemplo AreaChart

```
public class AreaChartExample extends Application {

    @Override
    public void start(Stage stage) {

        // ========================
        // 1) Definición del conjunto de datos
        // ========================
        // Serie: Producción solar
        XYChart.Series<String, Number> solar = new XYChart.Series<>();
        solar.setName("Solar (MWh)");
        solar.getData().add(new XYChart.Data<>("Ene", 120));
        solar.getData().add(new XYChart.Data<>("Feb", 180));
        solar.getData().add(new XYChart.Data<>("Mar", 260));
        solar.getData().add(new XYChart.Data<>("Abr", 310));
        solar.getData().add(new XYChart.Data<>("May", 280));
        solar.getData().add(new XYChart.Data<>("Jun", 240));

        // Serie: Producción eólica
        XYChart.Series<String, Number> eolica = new XYChart.Series<>();
        eolica.setName("Eólica (MWh)");
        eolica.getData().add(new XYChart.Data<>("Ene", 200));
        eolica.getData().add(new XYChart.Data<>("Feb", 220));
        eolica.getData().add(new XYChart.Data<>("Mar", 210));
        eolica.getData().add(new XYChart.Data<>("Abr", 260));
        eolica.getData().add(new XYChart.Data<>("May", 300));
        eolica.getData().add(new XYChart.Data<>("Jun", 330));

        // Serie: Producción hidráulica
        XYChart.Series<String, Number> hidro = new XYChart.Series<>();
        hidro.setName("Hidráulica (MWh)");
```

```
        hidro.getData().add(new XYChart.Data<>("Ene", 150));
        hidro.getData().add(new XYChart.Data<>("Feb", 140));
        hidro.getData().add(new XYChart.Data<>("Mar", 160));
        hidro.getData().add(new XYChart.Data<>("Abr", 170));
        hidro.getData().add(new XYChart.Data<>("May", 165));
        hidro.getData().add(new XYChart.Data<>("Jun", 155));

        // =========================
        // 2) Creación del gráfico
        // =========================
        AreaChart<String, Number> areaChart = createAreaChart(solar, eolica,
hidro);

        // =========================
        // 3) Layout: TilePane
        // =========================
        TilePane root = new TilePane();
        root.setPadding(new Insets(12));

        // Tamaño preferido para que se vean bien en mosaico
        areaChart.setPrefSize(450, 260);

        root.getChildren().addAll(areaChart);

        // =========================
        // 4) Escena y Stage
        // =========================
        stage.setTitle("Area Chart");
        stage.setScene(new Scene(root, 980, 620));
        stage.show();
    }

    // ============================================================
    // MÉTODO: AreaChart
    // Crea un gráfico de áreas con las mismas series
    // ============================================================
    private AreaChart<String, Number> createAreaChart(
            XYChart.Series<String, Number> s1,
            XYChart.Series<String, Number> s2,
            XYChart.Series<String, Number> s3) {

        // Ejes iguales al LineChart:
        // X categórico (meses) y Y numérico (MWh)
        CategoryAxis x = new CategoryAxis();
        NumberAxis y = new NumberAxis();
        x.setLabel("Mes");
        y.setLabel("MWh");
```

```
        // Creación del gráfico de áreas
        AreaChart<String, Number> chart = new AreaChart<>(x, y);
        chart.setTitle("Producción energética por mes (AreaChart)");

        // Animaciones activadas
        chart.setAnimated(true);

        // En gráficos de área suele ser preferible ocultar los símbolos
        // para evitar saturar visualmente el gráfico
        chart.setCreateSymbols(false);

        // Asociación directa de las series
        chart.getData().addAll(s1, s2, s3);

        return chart;
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

6.3. StackedAreaChart



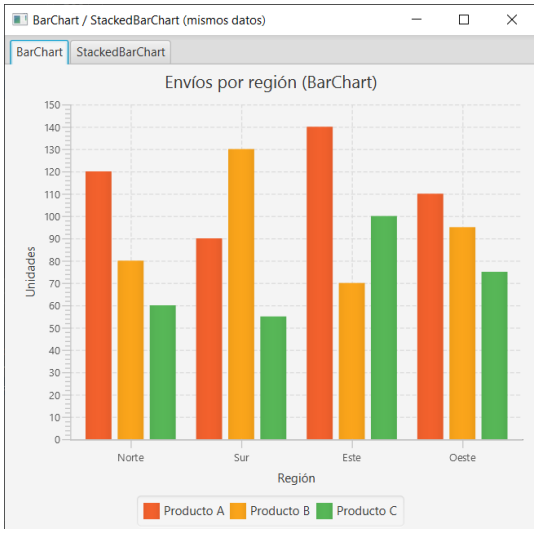
Ejemplo StackedAreaChart

```
public class StackedAreaChartExample extends Application {

    @Override
    public void start(Stage stage) {
```

```
        // ========================
        // 1) Definición del conjunto de datos
        // ========================
        // Serie: Producción solar
        XYChart.Series<String, Number> solar = new XYChart.Series<>();
        solar.setName("Solar (MWh)");
        solar.getData().add(new XYChart.Data<>("Ene", 120));
        solar.getData().add(new XYChart.Data<>("Feb", 180));
        solar.getData().add(new XYChart.Data<>("Mar", 260));
        solar.getData().add(new XYChart.Data<>("Abr", 310));
        solar.getData().add(new XYChart.Data<>("May", 280));
        solar.getData().add(new XYChart.Data<>("Jun", 240));

        // Serie: Producción eólica
        XYChart.Series<String, Number> eolica = new XYChart.Series<>();
        eolica.setName("Eólica (MWh)");
        eolica.getData().add(new XYChart.Data<>("Ene", 200));
        eolica.getData().add(new XYChart.Data<>("Feb", 220));
        eolica.getData().add(new XYChart.Data<>("Mar", 210));
        eolica.getData().add(new XYChart.Data<>("Abr", 260));
        eolica.getData().add(new XYChart.Data<>("May", 300));
        eolica.getData().add(new XYChart.Data<>("Jun", 330));

        // Serie: Producción hidráulica
        XYChart.Series<String, Number> hidro = new XYChart.Series<>();
        hidro.setName("Hidráulica (MWh)");
        hidro.getData().add(new XYChart.Data<>("Ene", 150));
        hidro.getData().add(new XYChart.Data<>("Feb", 140));
        hidro.getData().add(new XYChart.Data<>("Mar", 160));
        hidro.getData().add(new XYChart.Data<>("Abr", 170));
        hidro.getData().add(new XYChart.Data<>("May", 165));
        hidro.getData().add(new XYChart.Data<>("Jun", 155));

        // ========================
        // 2) Creación del gráfico
        // ========================
        StackedAreaChart<String, Number> stackedAreaChart =
createStackedAreaChart(solar, eolica, hidro);


        // ========================
        // 3) Layout: TilePane
        // ========================
        TilePane root = new TilePane();
        root.setPadding(new Insets(12));

        // Tamaño preferido para que se vean bien en mosaico
        stackedAreaChart.setPrefSize(450, 260);
```

```
        root.getChildren().addAll(stackedAreaChart);

        // ========================
        // 4) Escena y Stage
        // ========================
        stage.setTitle("StackedArea Chart");
        stage.setScene(new Scene(root, 980, 620));
        stage.show();
    }

    // ============================================================
    // MÉTODO: StackedAreaChart
    // Crea un gráfico de áreas apiladas (valores acumulados)
    // ============================================================
    private StackedAreaChart<String, Number> createStackedAreaChart(
            XYChart.Series<String, Number> s1,
            XYChart.Series<String, Number> s2,
            XYChart.Series<String, Number> s3) {

        // Eje X: meses
        CategoryAxis x = new CategoryAxis();

        // Eje Y: valores acumulados (suma de las series)
        NumberAxis y = new NumberAxis();
        x.setLabel("Mes");
        y.setLabel("MWh (acumulado)");

        // Creación del gráfico de áreas apiladas
        StackedAreaChart<String, Number> chart = new StackedAreaChart<>(x,y);
        chart.setTitle("Producción energética por mes (StackedAreaChart)");

        // Animaciones activadas para mostrar la construcción del acumulado
        chart.setAnimated(true);

        // Sin símbolos para una visualización más clara
        chart.setCreateSymbols(false);

        // Se añaden las series que se apilarán automáticamente
        chart.getData().addAll(s1, s2, s3);

        return chart;
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

6.4. BarChart



Ejemplo BarChart

```
public class BarChartExample extends Application {

    @Override
    public void start(Stage stage) {

        // ======================================================
        // 1) DEFINICIÓN DEL CONJUNTO DE DATOS
        // ======================================================
        // Cada serie representa un producto
        // Cada dato representa las unidades enviadas por región

        // Serie: Producto A
        XYChart.Series<String, Number> prodA = new XYChart.Series<>();
        prodA.setName("Producto A");
        prodA.getData().add(new XYChart.Data<>("Norte", 120));
        prodA.getData().add(new XYChart.Data<>("Sur", 90));
        prodA.getData().add(new XYChart.Data<>("Este", 140));
        prodA.getData().add(new XYChart.Data<>("Oeste", 110));

        // Serie: Producto B
        XYChart.Series<String, Number> prodB = new XYChart.Series<>();
        prodB.setName("Producto B");
        prodB.getData().add(new XYChart.Data<>("Norte", 80));
        prodB.getData().add(new XYChart.Data<>("Sur", 130));
        prodB.getData().add(new XYChart.Data<>("Este", 70));
        prodB.getData().add(new XYChart.Data<>("Oeste", 95));

        // Serie: Producto C
        XYChart.Series<String, Number> prodC = new XYChart.Series<>();
```

```
        prodC.setName("Producto C");
        prodC.getData().add(new XYChart.Data<>("Norte", 60));
        prodC.getData().add(new XYChart.Data<>("Sur", 55));
        prodC.getData().add(new XYChart.Data<>("Este", 100));
        prodC.getData().add(new XYChart.Data<>("Oeste", 75));

        // ======================================================
        // 2) CREACIÓN DEL GRÁFICO
        // ======================================================
        BarChart<String, Number> barChart =
                createBarChart(prodA, prodB, prodC);

        // ======================================================
        // 3) CONTENEDOR: TabPane
        // ======================================================
        // Cada gráfico se muestra en una pestaña distinta
        TabPane tabs = new TabPane(
                new Tab("BarChart", barChart)
        );

        // Se impide cerrar las pestañas
        tabs.getTabs().forEach(tab -> tab.setClosable(false));

        // ======================================================
        // 4) ESCENA Y STAGE
        // ======================================================
        stage.setTitle("BarChart");
        stage.setScene(new Scene(tabs, 950, 600));
        stage.show();
    }

    // ======================================================
    // MÉTODO: BarChart
    // ======================================================
    private BarChart<String, Number> createBarChart(
            XYChart.Series<String, Number> s1,
            XYChart.Series<String, Number> s2,
            XYChart.Series<String, Number> s3) {

        // Eje X categórico: regiones
        CategoryAxis xAxis = new CategoryAxis();

        // Eje Y numérico: unidades
        NumberAxis yAxis = new NumberAxis();

        xAxis.setLabel("Región");
        yAxis.setLabel("Unidades");
```

```
        // Creación del gráfico de barras
        BarChart<String, Number> chart = new BarChart<>(xAxis, yAxis);
        chart.setTitle("Envíos por región (BarChart)");

        // Animaciones activadas
        chart.setAnimated(true);

        // Espaciado entre categorías y entre barras
        chart.setCategoryGap(20);
        chart.setBarGap(6);

        chart.getData().addAll(s1, s2 ,s3);

        return chart;
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

6.5. StackedBarChart

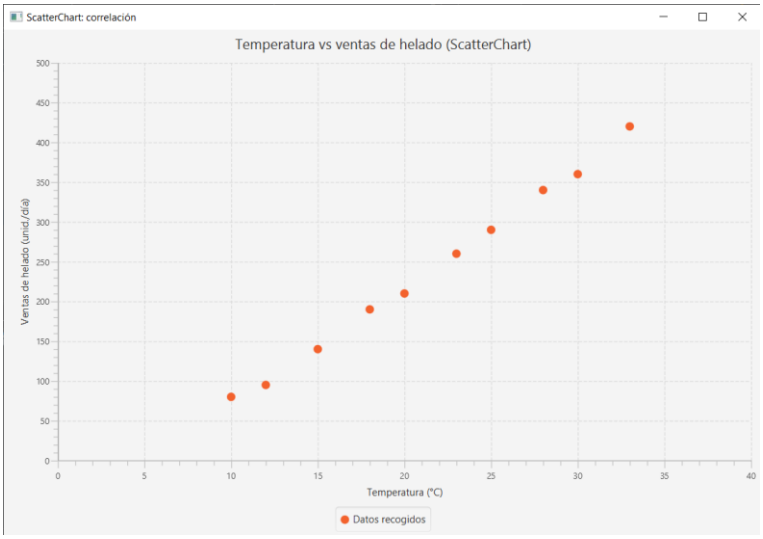
Ejemplo StackedBarChart

```
public class StackedBarChartExample extends Application {

    @Override
```

```
    public void start(Stage stage) {

        // ====================================================
        // 1) DEFINICIÓN DEL CONJUNTO DE DATOS
        // ====================================================
        // Cada serie representa un producto
        // Cada dato representa las unidades enviadas por región

        // Serie: Producto A
        XYChart.Series<String, Number> prodA = new XYChart.Series<>();
        prodA.setName("Producto A");
        prodA.getData().add(new XYChart.Data<>("Norte", 120));
        prodA.getData().add(new XYChart.Data<>("Sur", 90));
        prodA.getData().add(new XYChart.Data<>("Este", 140));
        prodA.getData().add(new XYChart.Data<>("Oeste", 110));

        // Serie: Producto B
        XYChart.Series<String, Number> prodB = new XYChart.Series<>();
        prodB.setName("Producto B");
        prodB.getData().add(new XYChart.Data<>("Norte", 80));
        prodB.getData().add(new XYChart.Data<>("Sur", 130));
        prodB.getData().add(new XYChart.Data<>("Este", 70));
        prodB.getData().add(new XYChart.Data<>("Oeste", 95));

        // Serie: Producto C
        XYChart.Series<String, Number> prodC = new XYChart.Series<>();
        prodC.setName("Producto C");
        prodC.getData().add(new XYChart.Data<>("Norte", 60));
        prodC.getData().add(new XYChart.Data<>("Sur", 55));
        prodC.getData().add(new XYChart.Data<>("Este", 100));
        prodC.getData().add(new XYChart.Data<>("Oeste", 75));

        // ====================================================
        // 2) CREACIÓN DEL GRÁFICO
        // ====================================================
        StackedBarChart<String, Number> stackedBarChart =
                createStackedBarChart(prodA, prodB, prodC);

        // ====================================================
        // 3) CONTENEDOR: TabPane
        // ====================================================
        // Cada gráfico se muestra en una pestaña distinta
        TabPane tabs = new TabPane(
                new Tab("StackedBarChart", stackedBarChart)
        );

        // Se impide cerrar las pestañas
        tabs.getTabs().forEach(tab -> tab.setClosable(false));
```

```
        // ======================================================
        // 4) ESCENA Y STAGE
        // ======================================================
        stage.setTitle("StackedBarChart");
        stage.setScene(new Scene(tabs, 950, 600));
        stage.show();
    }

    // ======================================================
    // MÉTODO: StackedBarChart
    // ======================================================
    private StackedBarChart<String, Number> createStackedBarChart(
            XYChart.Series<String, Number> s1,
            XYChart.Series<String, Number> s2,
            XYChart.Series<String, Number> s3) {

        CategoryAxis xAxis = new CategoryAxis();
        NumberAxis yAxis = new NumberAxis();

        xAxis.setLabel("Región");
        yAxis.setLabel("Unidades (total apilado)");

        // Creación del gráfico de barras apiladas
        StackedBarChart<String, Number> chart =
                new StackedBarChart<>(xAxis, yAxis);

        chart.setTitle("Envíos por región (StackedBarChart)");

        chart.setAnimated(true);
        chart.setCategoryGap(20);

        // Las series se apilan automáticamente
        chart.getData().addAll(
                cloneSeries(s1),
                cloneSeries(s2),
                cloneSeries(s3)
        );

        return chart;
    }


    public static void main(String[] args) {
        launch(args);
    }
}
```

6.6. ScatterChart



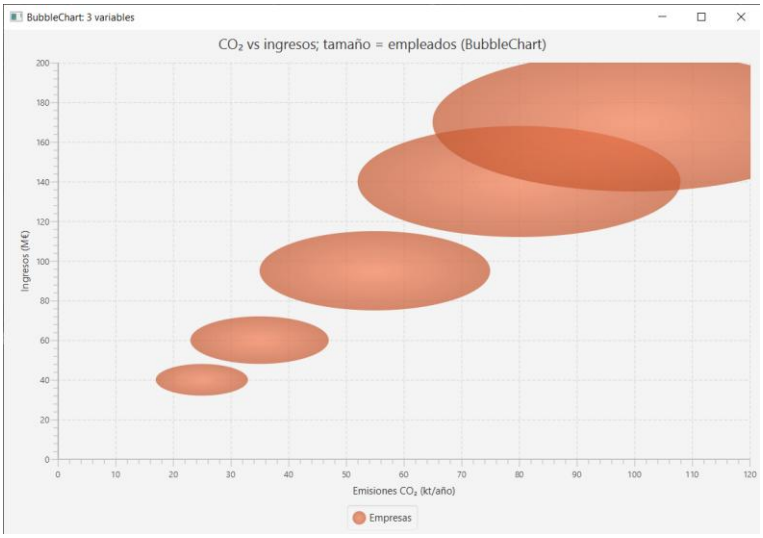
Ejemplo ScatterChart

```
public class ScatterChartExample extends Application {

    @Override
    public void start(Stage stage) {

        // ======================================================
        // 1) DEFINICIÓN DE LOS EJES
        // ======================================================
        // Eje X numérico:
        // - Rango: de 0 a 40
        // - Incremento de las marcas principales: 5
        NumberAxis x = new NumberAxis(0, 40, 5);

        // Eje Y numérico:
        // - Rango: de 0 a 500
        // - Incremento de las marcas principales: 50
        NumberAxis y = new NumberAxis(0, 500, 50);

        // Etiquetas descriptivas de los ejes
        x.setLabel("Temperatura (°C)");
        y.setLabel("Ventas de helado (unid./día)");

        // ======================================================
        // 2) CREACIÓN DEL SCATTERCHART
        // ======================================================
        // El ScatterChart representa los datos como puntos independientes,
        // sin líneas que los unan.
        ScatterChart<Number, Number> chart =
                new ScatterChart<>(x, y);
```

```
        // Título del gráfico
        chart.setTitle("Temperatura vs ventas de helado (ScatterChart)");

        // Activa animaciones al mostrar o actualizar los datos
        chart.setAnimated(true);

        // =====================================================
        // 3) DEFINICIÓN DE LA SERIE DE DATOS
        // =====================================================
        // Una serie representa un conjunto de observaciones.
        // Cada punto es un par (X, Y):
        //  - X: temperatura
        //  - Y: ventas de helado
        XYChart.Series<Number, Number> datos =
                new XYChart.Series<>();

        // Nombre de la serie (aparece en la leyenda)
        datos.setName("Datos recogidos");

        // Añadimos los puntos de datos al gráfico
        datos.getData().addAll(
                new XYChart.Data<>(10, 80),
                new XYChart.Data<>(12, 95),
                new XYChart.Data<>(15, 140),
                new XYChart.Data<>(18, 190),
                new XYChart.Data<>(20, 210),
                new XYChart.Data<>(23, 260),
                new XYChart.Data<>(25, 290),
                new XYChart.Data<>(28, 340),
                new XYChart.Data<>(30, 360),
                new XYChart.Data<>(33, 420)
        );

        // =====================================================
        // 4) ASOCIACIÓN DE LOS DATOS AL GRÁFICO
        // =====================================================
        // Se añade la serie completa al ScatterChart
        chart.getData().add(datos);

        // =====================================================
        // 5) ESCENA Y STAGE
        // =====================================================
        stage.setTitle("ScatterChart: correlación");
        stage.setScene(new Scene(chart, 900, 600));
        stage.show();
    }
```

```
    public static void main(String[] args) {
        launch(args);
    }
}
```

6.7. BubbleChart



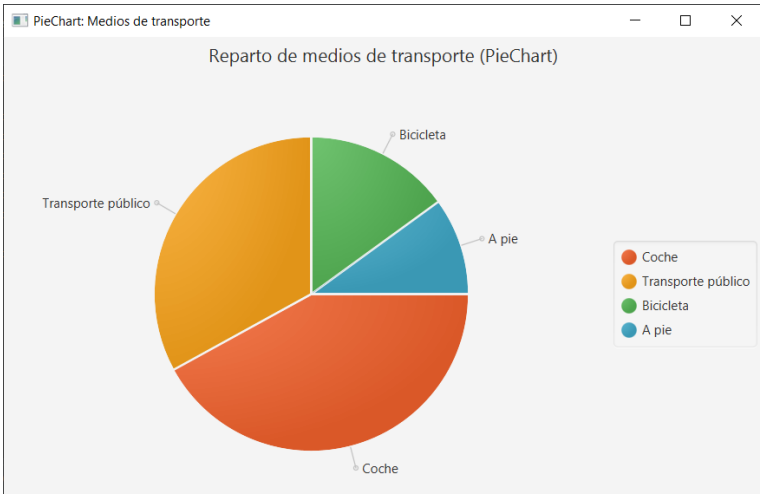
Ejemplo BubbleChart

```
public class BubbleChartExample extends Application {

    @Override
    public void start(Stage stage) {

        // ======================================================
        // 1) DEFINICIÓN DE LOS EJES
        // ======================================================
        // Eje X numérico:
        // - Rango: 0 a 120
        // - Incremento principal: 10
        NumberAxis x = new NumberAxis(0, 120, 10);

        // Eje Y numérico:
        // - Rango: 0 a 200
        // - Incremento principal: 20
        NumberAxis y = new NumberAxis(0, 200, 20);

        // Etiquetas descriptivas de los ejes
        x.setLabel("Emisiones CO₂ (kt/año)");
        y.setLabel("Ingresos (M€)");
```

```
        // ====================================================
        // 2) CREACIÓN DEL BUBBLECHART
        // ====================================================
        // BubbleChart extiende de XYChart y representa los datos
        // como burbujas cuyo tamaño es una tercera variable
        BubbleChart<Number, Number> chart =
                new BubbleChart<>(x, y);

        // Título del gráfico
        chart.setTitle("CO₂ vs ingresos; tamaño = empleados (BubbleChart)");

        // Activa animaciones al mostrar el gráfico
        chart.setAnimated(true);

        // ====================================================
        // 3) DEFINICIÓN DE LA SERIE DE DATOS
        // ====================================================
        // Cada punto tiene tres valores:
        //  - X: emisiones de CO₂
        //  - Y: ingresos
        //  - extraValue: tamaño relativo de la burbuja
        XYChart.Series<Number, Number> empresas =
                new XYChart.Series<>();

        // Nombre de la serie (aparece en la leyenda)
        empresas.setName("Empresas");

        // Añadimos las burbujas al gráfico
        empresas.getData().add(new XYChart.Data<>(35, 60, 12));  // Empresa A
        empresas.getData().add(new XYChart.Data<>(55, 95, 20));  // Empresa B
        empresas.getData().add(new XYChart.Data<>(80, 140, 28)); // Empresa C
        empresas.getData().add(new XYChart.Data<>(25, 40, 8));   // Empresa D
        empresas.getData().add(new XYChart.Data<>(100, 170, 35));// Empresa E

        // ====================================================
        // 4) ASOCIACIÓN DE LOS DATOS AL GRÁFICO
        // ====================================================
        // Se añade la serie completa al BubbleChart
        chart.getData().add(empresas);

        // ====================================================
        // 5) ESCENA Y STAGE
        // ====================================================
        stage.setTitle("BubbleChart: 3 variables");
        stage.setScene(new Scene(chart, 900, 600));
        stage.show();
    }
```

```
    public static void main(String[] args) {
        launch(args);
    }
}
```

6.8. PieChart



Ejemplo PieChart

```
public class PieChartExample extends Application {

    @Override
    public void start(Stage stage) {

        // ====================================================
        // 1) CREACIÓN DEL PIECHART Y SUS DATOS
        // ====================================================
        // El PieChart recibe una lista observable de PieChart.Data.
        // Cada PieChart.Data representa:
        //  - una etiqueta (String)
        //  - un valor numérico (Number)
        //
        // FXCollections.observableArrayList(...) crea una lista
        // que JavaFX puede observar para actualizar el gráfico
        // automáticamente si los datos cambian.
        PieChart chart = new PieChart(
                FXCollections.observableArrayList(
                        new PieChart.Data("Coche", 42),
                        new PieChart.Data("Transporte público", 33),
                        new PieChart.Data("Bicicleta", 15),
```

```
                        new PieChart.Data("A pie", 10)
                )
        );

        // =====================================================
        // 2) CONFIGURACIÓN GENERAL DEL GRÁFICO
        // =====================================================
        // Título principal del gráfico
        chart.setTitle("Reparto de medios de transporte (PieChart)");

        // Posición del título (TOP, RIGHT, BOTTOM, LEFT)
        chart.setTitleSide(Side.TOP);

        // Posición de la leyenda (lista de categorías)
        chart.setLegendSide(Side.RIGHT);

        // Muestra u oculta la leyenda
        chart.setLegendVisible(true);

        // Muestra las etiquetas directamente sobre cada sector
        chart.setLabelsVisible(true);

        // Activa animaciones al dibujar el gráfico
        chart.setAnimated(true);

        // =====================================================
        // 3) ESCENA Y STAGE
        // =====================================================
        stage.setTitle("PieChart: Medios de transporte");
        stage.setScene(new Scene(chart, 700, 450));
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```