

UNIDAD DIDÁCTICA 5

# LECTURA Y ESCRITURA DE INFORMACIÓN

MÓDULO PROFESIONAL:  
PROGRAMACIÓN



**CESUR**  
Tu Centro Oficial de FP

## Índice

RESUMEN INTRODUCTORIO .....	2
INTRODUCCIÓN .....	2
CASO INTRODUCTORIO .....	3
1. INTRODUCCIÓN A FLUJOS. EXCEPCIONES .....	4
2. CONCEPTO DE FLUJOS DE E/S .....	12
2.1 Flujos de bytes (byte streams) .....	12
2.2 Flujos de caracteres .....	16
2.3 Flujos de líneas.....	19
3. ENTRADA/SALIDA DESDE LA LÍNEA DE COMANDOS .....	22
4. FLUJOS DE DATOS .....	24
5. FLUJOS DE OBJETOS.....	30
6. FICHEROS. CLASE FILE .....	33
7. FICHEROS DE ACCESO ALEATORIO. CLASE RANDOMACCESSFILE .....	36
8. INTERFACES GRÁFICAS .....	42
8.1 Preparación del entorno .....	43
8.2 Configuración de Eclipse .....	46
8.3 Creando una librería de usuario .....	48
8.4 Creando un nuevo proyecto JavaFX project .....	53
8.5 Añadir la librería de usuario.....	55
8.6 Configurando el Build Path .....	57
8.7 Configurando Máquina virtual para usar JavaFX .....	60
8.8 Componentes de una aplicación JavaFX.....	63
RESUMEN FINAL .....	70

## RESUMEN INTRODUCTORIO

En esta unidad se explicarán las operaciones de E/S que se realizan normalmente en un programa, centrándose en un lenguaje de programación orientado a objetos que será Java. Veremos el concepto de flujo de entrada y de salida, los tipos de flujos según se clasifiquen. Posteriormente veremos cómo se realiza la entrada y la salida desde o hacia la línea de comandos.

Nos centraremos en los flujos de datos y en los flujos de objetos, en el concepto de serialización de objetos en el objeto File de Java y en los archivos de acceso aleatorio. Finalizaremos viendo una breve introducción a las interfaces gráficas, eventos y creación de controladoras de eventos. Para cada uno de los apartados veremos ejemplos en el lenguaje de programación Java.

## INTRODUCCIÓN

Frecuentemente un programa necesitará obtener información desde un origen o enviar información a un destino. Por ejemplo, obtener información desde el teclado, o bien enviar información a la pantalla. La comunicación entre el origen de cierta información y el destino se realiza mediante un flujo (stream) de información.

Los flujos de información actúan como canales de comunicación que permiten el intercambio de datos entre una aplicación y su entorno, o incluso entre diferentes partes de la aplicación en sí. Este intercambio puede ser sumamente versátil, abarcando desde la comunicación entre la aplicación y el mundo exterior, hasta la transferencia de datos entre archivos y la aplicación, o incluso entre distintos procesos en ejecución. Es esencial comprender el concepto de flujos de Entrada/Salida (E/S) en la programación, ya que constituyen la base para interactuar con datos en Java y en muchos otros lenguajes.

Por otro lado, el desarrollo de aplicaciones con interfaces gráficas de usuario (GUI) es una de las áreas más fascinantes y visualmente impactantes. JavaFX es una tecnología sólida y ampliamente utilizada para crear aplicaciones gráficas de escritorio atractivas y funcionales en Java.

## **CASO INTRODUCTORIO**

Una vez que has realizado en tu empresa el desarrollo de la última aplicación, utilizando interfaces gráficas en Java, te encuentras con el problema de que tienes que enviar datos a la aplicación a través de la línea de comandos y almacenar cierta información en archivos de acceso aleatorio. Para esta tarea es de vital importancia que conozcas cómo se trabaja con los flujos en el lenguaje de programación Java.

Al finalizar el estudio de la unidad, serás capaz de trabajar con datos provenientes de distintas entradas: teclado, archivos, etc., conocerás las clases que implementan estos flujos de entrada y salida, trabajarás con ficheros de acceso aleatorio y sabrás elaborar una interfaz gráfica, crear eventos y controlarlos.

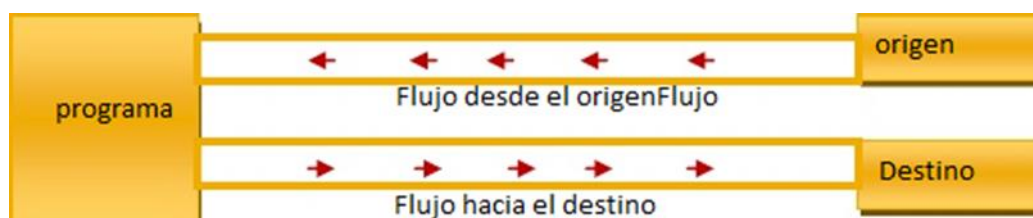
## 1. INTRODUCCIÓN A FLUJOS. EXCEPCIONES

*Un nuevo cliente encarga una aplicación que tiene como objetivo principal leer el contenido de ficheros que se encuentran almacenados en diferentes formatos y cuya información será utilizada en diferentes procesos que son vitales para la empresa. Debes controlar y manejar los escenarios de entrada/salida, así como los tipos de excepciones que se pueden producir para que el programa no termine de forma brusca o muestre mensajes no apropiados por el usuario.*

Un **flujo** es un objeto que hace de intermediario entre el programa, y el origen o el destino de la información. Esto es, el programa leerá o escribirá en el flujo sin importarle desde dónde viene la información o a dónde va y tampoco importa el tipo de los datos que se leen o escriben. Este nivel de abstracción hace que el programa no tenga que saber nada ni del dispositivo ni del tipo de información, lo que hace que la programación sea más fácil.

Los algoritmos para leer y escribir datos son siempre, más o menos, los mismos:

Leer	Escribir
Abrir un flujo desde un origen. Mientras haya información leer información. Cerrar el flujo.	Abrir un flujo hacia un destino. Mientras haya información escribir información. Cerrar el flujo.



Antes de comenzar a profundizar con el tratamiento de las operaciones de entrada y salida en Java es fundamental detenernos antes en el concepto de excepción y su tratamiento.

Las excepciones son eventos inusuales o condiciones de error que pueden ocurrir durante la ejecución de un programa. Estas situaciones pueden ser imprevistas y si no se manejan adecuadamente, pueden causar que el programa termine de manera abrupta o muestre resultados incorrectos. El uso adecuado de las excepciones en Java es fundamental para el manejo adecuado de errores y fallos en los programas. Permite que el código sea más robusto y evita que las fallas interrumpan la ejecución del programa de manera abrupta.

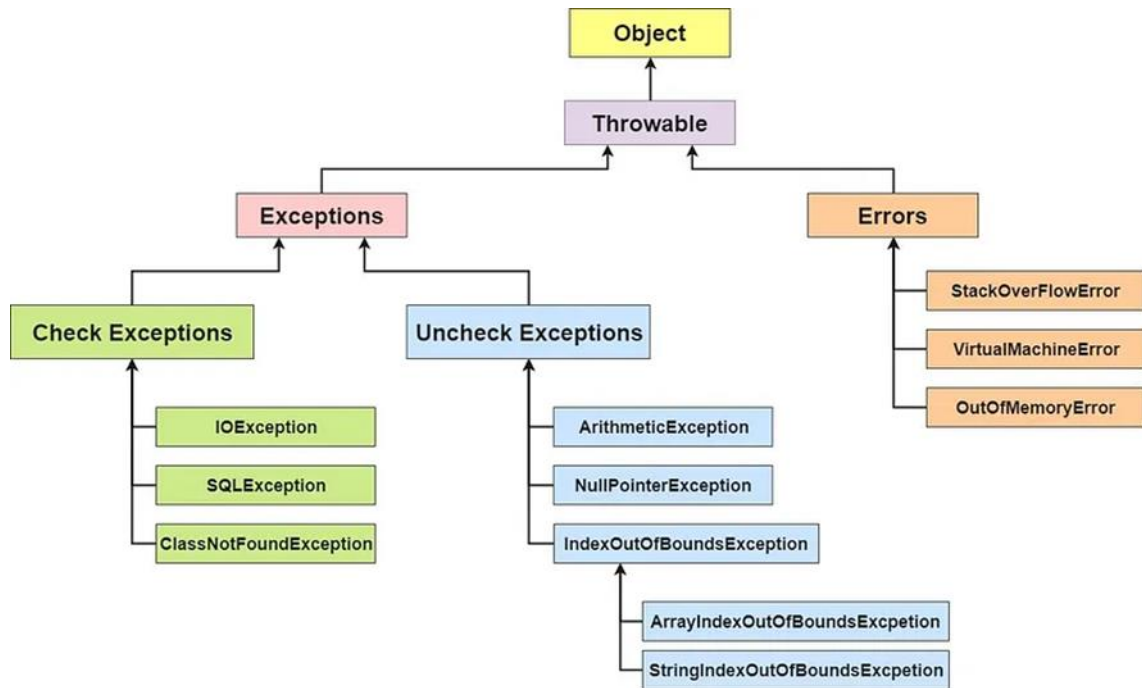
Las excepciones en Java se representan mediante objetos de clases que heredan de la clase `Throwable`. Existen dos tipos principales de excepciones en Java:

**Excepciones verificadas (checked exceptions):** Son aquellas que el compilador obliga a capturar o declarar en la firma del método que las lanza. Estas excepciones suelen estar relacionadas con problemas externos al programa, como operaciones de entrada/salida (I/O), conexiones a bases de datos, entre otras.

**Excepciones no verificadas (unchecked exceptions):** Son aquellas que el compilador no obliga a capturar o declarar en la firma del método que las lanza. Estas excepciones generalmente son causadas por errores en la lógica del programa, como división por cero, acceso a un índice fuera de rango en un array, entre otras.

En Java, la clase `Throwable` es la clase base de todas las clases que representan excepciones y errores. Tanto las excepciones como los errores heredan de `Throwable`, lo que significa que comparten ciertos métodos y comportamientos comunes.

- La clase `Exception`, es una subclase de `Throwable` y representa excepciones que pueden ser controladas y recuperadas durante la ejecución del programa. Por ejemplo, una excepción puede ser lanzada cuando se intenta leer un archivo que no existe o cuando ocurre un error de conexión a una base de datos.
- La clase `Errors` es también subclase de `Throwable` y representa problemas más graves que generalmente están fuera del control del programador. Estos errores pueden ocurrir debido a condiciones inusuales y en general, no es recomendable intentar recuperarse de ellos, ya que pueden indicar problemas de hardware, agotamiento de recursos, etc.



Jerarquía e clases que representan excepciones y errores.

Fuente: Recuperado de <https://medium.com/@manishaprabhodashani/head-first-java-chapter-11-summary-risky-behavior-bf0db9662d64>

La clase `Throwable` proporciona algunos métodos importantes que pueden ser utilizados por todas las clases que representan excepciones y errores, como `getMessage()` para obtener el mensaje asociado a la excepción, `printStackTrace()` para imprimir la traza de la pila que indica dónde se originó la excepción, entre otros.

Para controlar situaciones que produzcan excepciones en Java es común utilizar excepciones derivadas de la clase `Exception` y así capturarla y manejarla de manera adecuada garantizando una respuesta controlada ante situaciones imprevistas y facilitando la depuración y el mantenimiento de código.

### **Bloque try-catch.**

Es una construcción en Java que se utiliza para manejar excepciones. Permite proteger una sección de código donde pueden ocurrir excepciones, y en caso de que una excepción se produzca, proporciona un mecanismo para capturar y manejar dicha excepción, esta forma, el programa no se detendrá inesperadamente o se producirán mensajes del sistema no esperados.

La sintaxis básica del bloque "try-catch" es la siguiente:

```
try {  
    /* Aquí se incluyen las operaciones que podrían  
    generar excepciones, como operaciones de I/O,  
    división por cero, acceso a elementos fuera de los  
    límites de un array, entre otros*/  
} catch (TipoDeExcepcion e) {  
    /* Manejo de la excepción.  
    Si ocurre una excepción dentro del bloque "try", el  
    flujo del programa se desviará automáticamente  
    aquí*/  
}
```

El nombre "e" (o cualquier otro identificador) es el nombre de la variable que representa la excepción capturada, y se utiliza para acceder a la información relacionada con la excepción, como el mensaje de error o el seguimiento de la pila.

### **Bloque try-finally.**

Es una estructura de control en Java que se utiliza para garantizar que cierto código se ejecute independientemente de si ocurre una excepción o no dentro del bloque try. Es especialmente útil cuando se necesita asegurar que ciertos recursos sean liberados o acciones sean tomadas al final de una sección de código, sin importar si ha ocurrido un error o no.

```
try {  
    // Código que puede lanzar una excepción  
} finally {  
    // Código que se ejecutará siempre, ya sea que haya  
    // ocurrido una excepción o no.  
}
```

Si al ejecutar el código dentro del bloque try, ocurre una excepción, el flujo del programa se desvía a buscar un bloque catch que pueda manejar esa excepción (aunque puede no haberlo) y a continuación se ejecutará el bloque finally sin importar lo que haya sucedido en el bloque try.

El bloque finally es útil para realizar acciones de limpieza o liberar recursos que deben ocurrir siempre, independientemente de si ha ocurrido una excepción o no. Por ejemplo, cerrar un archivo, liberar una conexión de base de datos, cerrar un socket, etc.

A continuación, vamos a escribir código que produzca una excepción.



Utilizaremos la clase `Integer` (wrapper class) denominada de envoltura que proporciona Java para trabajar con tipos primitivos como `int`. Esta clase envuelve o empaqueta un valor entero dentro de un objeto. Los tipos primitivos como `int`, `double`, `boolean`, etc. no son considerados objetos y no tienen métodos asociados a ellos. Pero hay ocasiones en las que es útil trabajar con estos tipos primitivos como si fueran objetos. Es aquí donde entran en juego las clases de envoltura, que proporcionan funcionalidades adicionales para los tipos primitivos.

La siguiente línea, `numero = Integer.parseInt(str);` realiza la conversión de un valor tipo cadena y devuelve el valor equivalente en formato entero.

`parseInt(str)` es un método proporcionado por la clase `Integer`, que toma como argumento una cadena de caracteres en este caso "12" y devuelve el valor equivalente de la conversión en formato entero y lo almacena en la variable `número`.

¿Qué va a ocurrir?, pues bien, se producirá una excepción del tipo `NumberFormatException` porque no se puede convertir "12" a número entero porque tiene espacios incluidos (si solo son números, no habría problema). Esta excepción será capturada y se imprime el mensaje "No es un número".

Se pueden manejar más de una excepción en el bloque try-catch. Imaginemos el caso en el que queremos dividir dos números. Se pueden producir dos excepciones en la introducción de números, una de ellas puede ser que se introduzcan caracteres no numéricos (`NumberFormatException`) o que se divida un número entre cero (`ArithmeticException`).

```
public class Ud5EjemploExcepcion
{
    public static void main(String[] args) {
        String str=" 12 ";
        int numero;
        try{
            numero=Integer.parseInt(str);
        }catch(NumberFormatException ex){
            System.out.println("No es un número");
        }
    }
}
```

La salida por pantalla es la siguiente:

No es un número

En este otro ejemplo, se maneja además de la excepción `NumberFormatException` la excepción `ArithmeticException` que se producirá en la expresión que realiza el cociente cuando el denominador es cero. El flujo del programa sale del bloque `try` y entra en el bloque `catch` que recoge inmediatamente la excepción y la maneja, ejecutándose las sentencias que hay en dicho bloque `catch`. En este caso se guarda en el `String` `respuesta` el texto "División entre cero". El flujo del programa continuará después del bloque `catch`, ejecutando una a una las instrucciones que sigan a continuación. Por tanto, se mostrará en pantalla el valor de la variable `respuesta`.

En caso de que todo vaya bien y no se produzca la excepción en la operación de división entonces el flujo de programa continuará por la sentencia que sigue a continuación y que es la siguiente línea de código

```
respuesta=String.valueOf(cociente);
```

cuya misión es convertir el valor de la variable numérica `cociente` en su representación como cadena de caracteres (`String`) para asignarlo a la variable `respuesta` que será mostrada con la instrucción.

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

```
public class Ud5EjemploExcepciones {  
    public static void main(String[] args) {  
  
        String str1="12";  
        String str2="0";  
        String respuesta;  
        int numerador, denominador, cociente;  
  
        try{  
            numerador=Integer.parseInt(str1);  
            denominador=Integer.parseInt(str2);  
            cociente=numerador/denominador;  
            respuesta=String.valueOf(cociente);  
        }catch(NumberFormatException ex){  
            respuesta="Se han introducido caracteres no  
numéricos";  
        }catch(ArithmeticException ex){  
            respuesta="División entre cero";  
        }  
    }  
}
```

```
        System.out.println(respuesta);  
    }  
}
```

Las excepciones además de producir las el sistema se pueden lanzar por el desarrollador cuando el propósito es comunicar que ha ocurrido una situación inusual o un error que debe ser manejado en un lugar diferente del código, posiblemente en un nivel superior de la jerarquía de llamadas de métodos. Al lanzar excepciones, se permite que el control del programa pase al bloque try-catch correspondiente que podrá manejar la excepción de la manera indicada. Se creará una instancia de una clase excepción y se enviará a lo largo del flujo de ejecución del programa para indicar una situación excepcional o un error que el programa no puede manejar en ese momento.

Para lanzar una excepción, se utiliza la palabra clave throw seguida de una instancia de la clase de excepción que deseamos enviar.

En este ejemplo, cuando el código se ejecuta, se intenta realizar la división dividendo / divisor, pero dado que divisor tiene el valor de cero, se lanzará una excepción ArithmeticException. Luego, el bloque catch captura la excepción y muestra el mensaje "Error aritmético: No se puede dividir por cero." en la salida.

```
public class Ud5EjemploExcepcionThrow {  
    public static void main(String[] args) {  
  
        try {  
            int dividendo = 10;  
            int divisor = 0;  
  
            if (divisor == 0) {  
                throw new ArithmeticException("No se puede  
                                                dividir por cero.");  
            }  
  
            int resultado = dividendo / divisor;  
            System.out.println("Resultado de la división: " +  
                               resultado);  
        } catch (ArithmeticException e) {  
            System.out.println("Error aritmético: " +  
e.getMessage());  
        }  
    }  
}
```

La salida por pantalla es la siguiente:

Error aritmético: No se puede dividir por cero.

En el manejo de archivos, pueden ocurrir varias excepciones que debemos tener en cuenta para garantizar que nuestro código funcione de manera robusta y adecuada. Algunas de las excepciones más comunes relacionadas con el manejo de archivos en Java que veremos en la unidad son:

**FileNotFoundException:** Se lanza cuando intentamos acceder a un archivo que no existe en el sistema.

**IOException:** Es una excepción más general que puede ocurrir durante operaciones de entrada/salida, como leer o escribir en un archivo. Esta excepción engloba varios subtipos específicos de excepciones relacionadas con I/O.

**EOFException:** En Java es una subclase de la clase IOException, que a su vez es una excepción de entrada/salida (I/O). EOF significa "End of File" o "Fin de Archivo" y esta excepción se lanza cuando se llega al final de un flujo de entrada y se intenta leer más datos (cuando no hay más), es muy utilizada cuando se trabaja con archivos binarios.

Es importante manejar adecuadamente estas excepciones para evitar que nuestro programa se bloquee o se detenga abruptamente en caso de ocurrir algún problema con los archivos. El manejo de excepciones nos permite tomar acciones adecuadas, como mostrar mensajes de error al usuario, cerrar recursos abiertos o realizar alguna recuperación de errores, si es posible.

Cuando trabajamos con archivos, también es recomendable cerrar correctamente los recursos después de utilizarlos (método `close()`), esto libera los recursos del sistema y evita problemas de manejo de archivos en el futuro.



#### ENLACE DE INTERÉS

Consulta este resumen sobre las excepciones en Java y el uso del bloque `try..catch..finally`



## 2. CONCEPTO DE FLUJOS DE E/S

*En la aplicación en la que trabajas actualmente debes de utilizar flujos para permitir que la interacción con los datos sea uniforme y no tengas que preocuparte por la fuente o el destino específico de esos datos. Esto te proporcionará ciertas ventajas, como por ejemplo leer o escribir datos en un flujo sin conocer los detalles internos de dónde provienen o hacia dónde van.*

Ahora sabremos que significan los siguientes conceptos de flujos:

- Un flujo de **entrada/salida** (I/O stream, Input/Output stream) representa una fuente desde la cual se reciben datos o un destino hacia el cual se envían datos.
- Un flujo de datos puede **provenir o dirigirse** hacia archivos en disco, dispositivos de comunicaciones, otros programas o arrays en memoria.
- Los datos pueden ser **bytes**, **tipos primitivos**, **caracteres** propios de un idioma local, u **objetos**.



### ARTÍCULO DE INTERÉS

Para ampliar información sobre los flujos y algunas clases utilizadas se recomienda visitar este enlace:



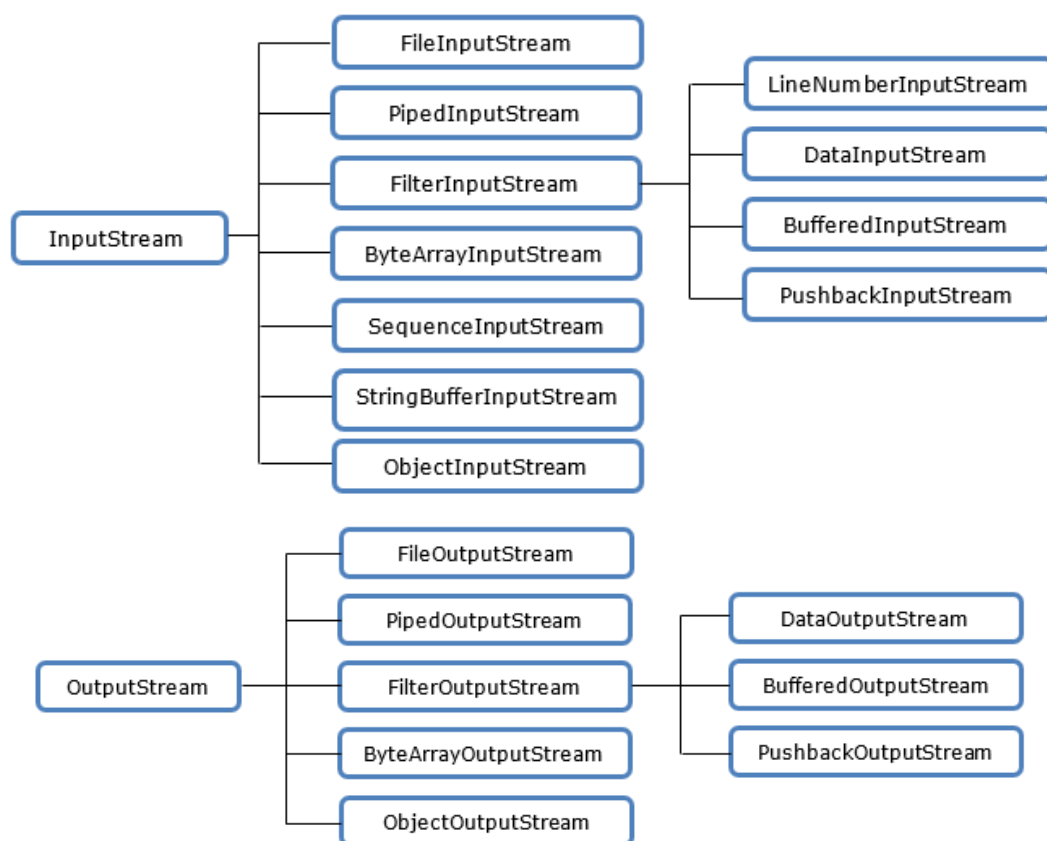
### 2.1 Flujos de bytes (byte streams)

Los flujos de bytes realizan operaciones de entrada y salida basándose en **bytes** de 8 bits. Todas las clases de flujos de bytes descienden (heredan) de las clases `InputStream` y `OutputStream`. Las clases `FileInputStream` y `FileOutputStream` manipulan flujos de bytes provenientes o dirigidos hacia archivos en disco se utilizan para realizar operaciones de entrada y salida (I/O) de bytes desde y hacia archivos, respectivamente. Ambas clases son parte del paquete `java.io` y son utilizadas para trabajar con archivos binarios.

La clase `FileInputStream` se utiliza normalmente para leer bytes desde un archivo binario. Se encarga de leer los bytes sin realizar ninguna interpretación especial, lo que la hace adecuada para leer archivos binarios como imágenes, videos, archivos de audio, etc.

La clase `FileOutputStream` se utiliza normalmente para escribir bytes en un archivo binario. Al igual que `FileInputStream`, es útil para escribir archivos binarios.

Ambas, se pueden utilizar para leer y escribir archivos de texto(.txt) respectivamente, pero hay que tener en cuenta que leerán y escribirán bytes sin realizar ninguna interpretación especial de los caracteres.





### EJEMPLO PRÁCTICO

En el siguiente ejemplo se propone leer un fichero origen y copiar su información a un fichero destino, si el fichero no existe se lanzará una excepción que será controlada.

Debe incluir el método `getProperty(String clave)` que pertenece a la clase `System` utilizado para obtener valores de propiedades del sistema o del entorno de ejecución de Java que será útil si no sabemos dónde busca el programa o donde se crean los archivos con los que se trabaja en ese momento.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Ud5EjemploCopiaFichero {

    public static void main(String[] args) throws IOException {
        //declaración de objetos de tipo FileReader
        FileInputStream in = null;
        FileOutputStream out = null;

        String respuesta;

        String ficheroOrigen = "ficheroOrigen.txt";
        String ficheroDestino = "ficheroDestino.txt";

        //probando usos de getProperty...

        /*Establecemos la ruta usando getProperty, para que nos cree
        el fichero en la ruta en la que estamos ahora*/

        String rutaFicheroALeer = System.getProperty("user.dir") +
                                   "\\ " + ficheroOrigen;
        String rutaFicheroAEscribir = System.getProperty("user.dir") +
                                      "\\ " + ficheroDestino;

        String sistemaOperativo = System.getProperty("os.name");

        System.out.println("Sistema operativo " + sistemaOperativo);
        System.out.println("La ruta de trabajo es " +
                           rutaFicheroALeer);

        try {
            /*se crean los flujos de entrada y salida,
            para ello se instancian los objetos de las clases*/
            in = new FileInputStream(rutaFicheroALeer);
```

```
out = new FileOutputStream(rutaFicheroAEscribir);
int c;
//cada byte se guarda en una variable de tipo int
//Se repite el bucle mientras no sea fin de fichero
while((c=in.read())!=-1) {
    out.write(c);
}
}catch(IOException ex) {
    respuesta="El fichero de lectura origen " +
        ficheroOrigen +
        " no existe, debes crearlo antes ";
    System.out.println(respuesta);
}finally {
    if(in!=null)
        in.close();
    if(out!=null)
        out.close();
}
```

La salida por pantalla sería:

Sistema operativo Windows 10

La ruta de trabajo es E:\ProyectosJava\ficheroOrigen.txt

El fichero de lectura origen ficheroOrigen.txt no existe, debes crearlo antes.

El método `read()` devuelve un valor entero, lo cual permite indicar con el valor -1 el final del flujo. El tipo primitivo `int` puede almacenar un byte. Mantener flujos abiertos implica un gasto de recursos; deben cerrarse estos flujos para evitar malgastar recursos. El programa anterior cierra los flujos en el bloque `finally`. En este bloque se verifica que los flujos fueron efectivamente creados (sus referencias no son `null`) y luego se cierran.



### ENLACE DE INTERÉS

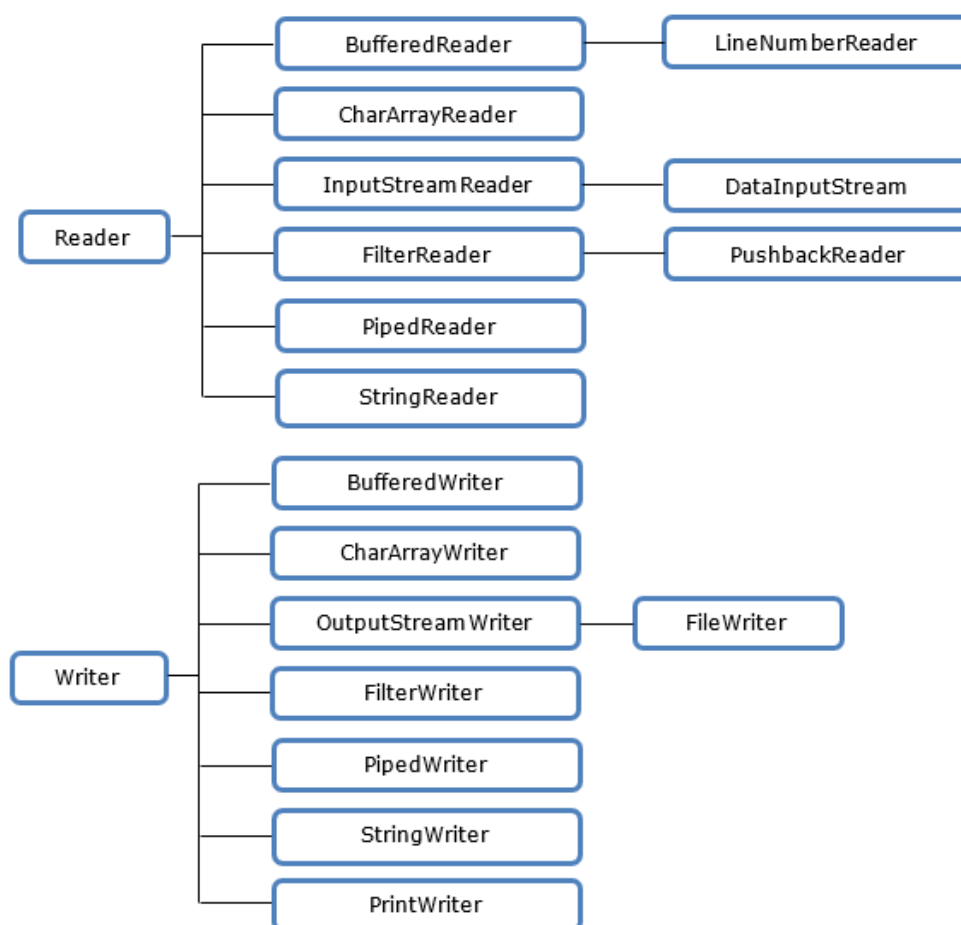
Si quiere saber más sobre la clase `InputStream` visita esta web:





## 2.2 Flujos de caracteres

El uso de flujos de bytes sólo es apto para las operaciones más elementales de entrada salida; es preciso usar los flujos más adecuados según los tipos de datos a manejar. En el ejemplo anterior, como se sabe que es un archivo con caracteres, lo mejor es usar los flujos de caracteres definidos en las clases `FileReader` y `FileWriter`. Estas clases heredan de `Reader` y `Writer`, y están destinadas a la lectura y escritura de caracteres en archivos.



Cuando el objetivo sea trabajar específicamente con archivos de texto y tratar directamente con caracteres en lugar de bytes, es más recomendable utilizar las clases `FileReader` y `FileWriter`, ya que estas realizan la conversión de bytes a caracteres y viceversa automáticamente utilizando la codificación de caracteres predeterminada del sistema.



### EJEMPLO PRÁCTICO

En este ejemplo se leerá de un fichero origen y se copiará la información a un fichero destino, si el fichero no existe se lanzará una excepción que será controlada. La diferencia con el ejemplo anterior son las clases utilizadas.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Ud5EjemploCopiaFichero {

    public static void main(String[] args) throws IOException {
        //declaración de objetos de tipo FileReader
        FileReader in = null;
        FileWriter out = null;

        String respuesta;

        String ficheroOrigen = "ficheroOrigen.txt";
        String ficheroDestino = "ficheroDestino.txt";

        //probando usos de getProperty...

        /*Establecemos la ruta usando getProperty, para que nos cree
        el fichero en la ruta en la que estamos ahora*/

        String rutaFicheroALeer = System.getProperty("user.dir") +
                                   "\\ " + ficheroOrigen;
        String rutaFicheroAEscribir = System.getProperty("user.dir") +
                                       "\\ " + ficheroDestino;

        System.out.println("La ruta de trabajo es " +
                           rutaFicheroALeer);

        try {
            /*se crean los flujos de entrada y salida,
            para ello se instancian los objetos de las clases*/
            in = new FileReader(rutaFicheroALeer);
            out = new FileWriter(rutaFicheroAEscribir);
            int c;
            //cada byte se guarda en una variable de tipo int
            //Se repite el bucle mientras no sea fin de fichero
            while((c=in.read())!=-1) {
                out.write(c);
            }
        }
    }
}
```

```
    }  
    }catch(IOException ex) {  
        respuesta="El fichero de lectura origen " +  
            ficheroOrigen +  
            "no existe, debes crearlo antes ";;  
        System.out.println(respuesta);  
    }finally {  
        if(in!=null)  
            in.close();  
        if(out!=null)  
            out.close();  
    }  
}  
}
```

La salida por pantalla podría ser:

La ruta de trabajo es E:\ProyectosJava\ficheroOrigen.txt

El fichero de lectura origen ficheroOrigen.txt no existe, debes crearlo antes.



### ENLACE DE INTERÉS

Visualiza este **canal completo** en el que se muestran pequeños vídeos sobre la programación en Java. En concreto tiene dos videos muy interesantes sobre Entrada y Salida en Java.



## 2.3 Flujos de líneas

Para la lectura y escritura por líneas se emplean las clases `BufferedReader` y `PrintWriter`. Estas clases son muy eficientes cuando se trabaja con archivos de texto y se necesitan leer o escribir líneas completas en lugar de caracteres individuales.



### ENLACE DE INTERÉS

Analiza un ejemplo completo utilizando la clase `BufferedReader`.



En el siguiente ejemplo que veremos, utilizamos `BufferedReader` para leer el archivo "Archivo.txt". El método `readLine()` se llama dentro del bucle `while` y leerá una línea completa en cada iteración de modo que esta línea leída se escribirá como una línea en el archivo "CopiaLineas.txt" archivo.

También se puede usar para la salida, la clase `BufferedWriter` en vez de `PrintWriter`.

Cuando se necesita escribir datos formateados o se busca simplicidad y comodidad en la escritura de datos, `PrintWriter` puede ser una buena opción.

En aquellas situaciones en las que se escriben datos en un archivo de texto de forma repetida o en grandes cantidades, `BufferedWriter` puede ser más adecuado debido a su capacidad para utilizar un búfer interno, que es un área de memoria utilizada como almacenamiento intermedio que almacenará temporalmente los datos antes de que se escriban físicamente en el archivo. Esto permite agrupar varias operaciones de escritura en bloques más grandes y reducir el número de llamadas al sistema operativo para realizar operaciones de escritura física en el disco.

El uso del búfer puede mejorar significativamente el rendimiento en situaciones en las que se realizan muchas escrituras pequeñas o en operaciones de escritura frecuentes.

En muchos casos, ambas clases pueden utilizarse de manera intercambiable según tus preferencias y requerimientos. Ambas son herramientas útiles para escribir en archivos de texto en Java.



## EJEMPLO PRÁCTICO

Analiza el ejemplo propuesto que usa un flujo de entrada con buffer.

```
package flujosbytes;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class CopiarLineas {
    public static void main(String[] args) throws IOException {
        BufferedReader in = null;
        PrintWriter out = null;
        try {
            // Se crean los flujos de entrada y salida
            in = new BufferedReader(new FileReader("archivo.txt"));
            out = new PrintWriter(new FileWriter("CopiaLineas.txt"));
            // Cada byte se guarda en una variable de tipo String
            String linea;
            while ((linea = in.readLine()) != null) {
                out.println(linea);
            }
        } finally {
            if (in != null)
                in.close();
            if (out != null)
                out.close();
        }
    }
}
```

`BufferedReader` es una clase que se encuentra en el paquete `java.io`. Se utiliza para leer texto de una fuente de entrada, como un archivo o un flujo de caracteres. Una de las principales ventajas de `BufferedReader` es que almacena en memoria un búfer (buffer) que permite leer datos en bloques, lo que mejora el rendimiento de la lectura de datos grandes.

Para leer líneas completas, la clase `BufferedReader` proporciona el método `readLine()`. Este método lee una línea de texto desde la fuente de entrada hasta que encuentra un

salto de línea ('\n') o llega al final del archivo, y devuelve esa línea como una cadena (String).

Cuando se usan buffers sólo se lee o escribe en el dispositivo final cuando el buffer está lleno, reduciendo la cantidad de operaciones de lectura y escritura sobre los dispositivos lentos (más lentos que la memoria).



#### **NOTA DE INTERÉS**

Las clases disponibles para entrada y salida con buffer son `BufferedInputStream` y `BufferedOutputStream` para flujos de bytes, `BufferedReader` y `BufferedWriter` para flujos de caracteres. `Reader` y `Writer` son las clases bases de la jerarquía para los flujos de caracteres.

### 3. ENTRADA/SALIDA DESDE LA LÍNEA DE COMANDOS

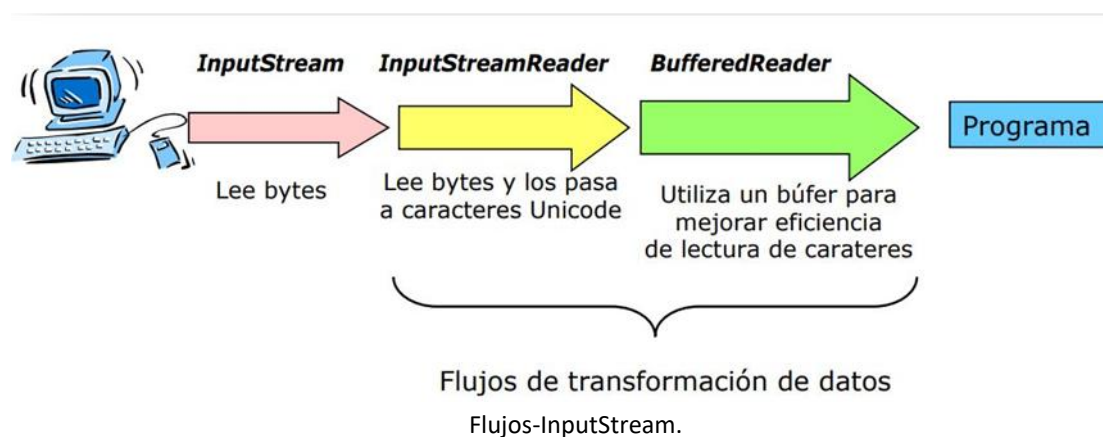
*En la aplicación software que estás desarrollando los flujos de entrada/salida estándar se han convertido en una herramienta fundamental, ya que se encargarán de la comunicación entre el programa y el usuario, proporcionando una mejor experiencia de usuario. Debes solicitar los datos por pantalla y una vez recibidos por la aplicación esta responderá con resultados significativos y mensajes apropiados.*

En Java existen varios flujos para la interacción con el usuario en la línea de comandos. Estos flujos se denominan flujos estándar (standard streams), y son comunes en varios sistemas operativos. Por defecto estos flujos leen del teclado y escriben en pantalla. Como hemos visto estos flujos pueden ser manipulados mediante clases e interfaces que están ubicadas dentro del paquete java.io y pueden redirigirse hacia archivos u otros programas. En Java hay tres flujos estándar:

- La entrada estándar (**Standard Input**), accesible a través del objeto `System.in`; La entrada estándar está asignada al teclado.
- La salida estándar (**Standard Output**), accesible a través del objeto `System.out`; La salida estándar está asignada a la pantalla.
- La salida de mensajes de error, accesible a través del objeto `System.err`;

Estos objetos se definen automáticamente y no requieren ser abiertos. Para usar la entrada estándar como flujo de caracteres se "envuelve" el objeto `System.in` en un objeto `InputStreamReader`.

```
InputStreamReader cin = new InputStreamReader(System.in);
```



Fuente: <https://www.fdi.ucm.es/profesor/jpavon/poo/2.13.EntradaYSalida.pdf>

Este ejemplo solicita al usuario ingresar una línea de caracteres, finalizando con la tecla Enter, y la muestra después en pantalla. Si se produce una excepción del tipo `IOException`, será capturada y se imprimirá la traza del error con la instrucción `e.printStackTrace()`; a continuación continuará el flujo de ejecución a la siguiente instrucción que se encuentre justo después del `catch`. Si la variable `línea` tiene un valor distinto de `null` significará que no se ha producido la excepción por tanto la línea se ha escrito y se informa de ello, por otro lado, si la variable `línea` tiene el valor `null`, no se ha escrito nada, igualmente se indicará con un mensaje. Por último, se cierra el flujo de entrada si está abierto.

```
public class Ud5EjemploEntradaStandar {

    public static void main(String[] args) throws IOException {
        BufferedReader stdin = null;
        stdin = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Introduzca caracteres y pulse
Return: ");
        String linea = null;
        try {
            linea = stdin.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
        if(linea!=null) {
            System.out.println("Ha escrito:" + linea);
        }else {
            System.out.println("No se ha escrito línea
porque ha
habido una excepción");
        }
        if(stdin!=null) stdin.close();
    }
}
```





### ARTÍCULO DE INTERÉS

Una alternativa a los flujos estándar es Console. En este enlace podrás aprender más sobre esta clase:



## 4. FLUJOS DE DATOS

*Te dispones a crear una nueva funcionalidad en tu aplicación. De modo que para cada compra que realiza un cliente se genere una factura que contiene los datos de los artículos comprados, concepto, cantidad, precio unitario y total. Como vas a trabajar con datos de tipo primitivo y de cadenas de caracteres decides utilizar los flujos de datos en Java.*

Los flujos de datos soportan operaciones de entrada/salida de datos de tipo primitivo (boolean, char, byte, short, int, long, float, y double) así como cadenas de caracteres (String).

Las clases `DataInputStream` y `DataOutputStream` pertenecen al paquete `java.io`, permiten escribir un objeto en un flujo de salida y leerlo de nuevo desde un flujo de entrada, manteniendo así su estado y estructura original.

La clase `DataOutputStream`, es una clase en Java muy útil para la manipulación y transmisión de datos estructurados en aplicaciones Java. Nos proporciona clases para realizar operaciones de salida en Java. Es una subclase de `FilterOutputStream` y se utiliza para escribir datos primitivos y otros tipos de datos en un flujo de salida con un formato portable.

`DataOutputStream` permite escribir datos de diferentes tipos, como enteros, números de punto flotante, caracteres, booleanos, cadenas y más, en un flujo de salida. Los datos se escriben en un formato binario que es independiente de la plataforma, lo que garantiza que los datos puedan ser leídos correctamente en diferentes sistemas operativos.

Para utilizar `DataOutputStream`, es común crear una instancia de esta clase pasando un flujo de salida (`OutputStream`) como argumento al constructor. Luego, puedes utilizar los diversos métodos proporcionados por `DataOutputStream`, como `writeInt()`, `writeDouble()`, `writeUTF()`, etc., para escribir los datos en el flujo.

Podemos decir, que la característica más destacada de la clase `DataOutputStream` en Java es que se utiliza para escribir datos en un formato binario portable en un flujo de salida.



### EJEMPLO PRÁCTICO

En este ejemplo se crea un archivo llamado `Factura.txt` y escribe en él una serie de datos correspondientes a una factura de venta (Parte 1 de 2).

```
import java.io.BufferedOutputStream;
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class prueba {

    static String archDatos = "Factura.txt";
    static final double[] precios = {18.00, 160.00, 25.00, 14.00, 2.50};
    static final int[] cants = {4,2,1,4,50};
    static final String[] items = {"Marcador Azul", "Papel A4 500
                                    hojas", "Borrador", "DVD", "Sobres
                                    A4"};

    public static void main(String[] args) throws IOException {
        /*se crea un objeto DataOutputStream para escribir datos en el
        archivo de texto*/
        DataOutputStream out = null;
        try {
            out = new DataOutputStream(new BufferedOutputStream(new
                FileOutputStream(archDatos)));
            for(int i =0; i< precios.length;i++) {
                out.writeDouble(precios[i]);
                out.writeInt(cants[i]);
                out.writeUTF(items[i]);
            }
        }catch(IOException ex) {
            System.out.println("Error E/S");
        }finally {
            //cerramos flujo y liberamos recursos.
            out.close();
        }
    }
}
```

```
        }  
    }  
}
```

En el ejercicio se han declarado una serie de variables de tipo `static`. Este modificador todavía no se ha visto en el temario, pero de momento solo es necesario saber que declaradas de esta forma se va a permitir su acceso directo desde el método estático `main` que también es un método estático y se ejecuta sin necesidad de crear una instancia de la clase `FlujoDeDatos`.

- La variable `archDatos` de tipo `String`, contendrá el nombre del archivo donde se escribirán los datos.
- El array `precios` almacena los precios de diferentes artículos. Cada precio representa el precio de un artículo en una posición específica del array. Por ejemplo, el primer elemento (índice 0) contiene el precio del "Marcador Azul" (18.00), el segundo elemento (índice 1) contiene el precio del "Papel A4 500 hojas" (160.00), y así sucesivamente.
- El array `cants`, contiene las cantidades de cada artículo. Cada elemento del array `cants` representa la cantidad de un artículo en una posición específica del array. Por ejemplo, el primer elemento (índice 0) contiene la cantidad del "Marcador Azul" (4), el segundo elemento (índice 1) contiene la cantidad del "Papel A4 500 hojas" (2), y así secuencialmente.

La siguiente instrucción crea una cadena de flujos que permitirá escribir datos en un archivo de texto.

```
out=new DataOutputStream (new BufferedOutputStream(new  
    FileOutputStream(archDatos)));
```

Está compuesta de las siguientes partes:

- **New FileOutputStream(archDatos):** Esta parte crea un objeto de tipo `FileOutputStream`, que representa un flujo de salida hacia un archivo. Toma como argumento el nombre del archivo `archDatos`. Este objeto será la base para el siguiente paso.

- **new BufferedOutputStream(...):** Aquí, se crea un objeto de tipo `BufferedOutputStream`, que envuelve al objeto `FileOutputStream`. Un `BufferedOutputStream` es un tipo de flujo de salida que utiliza un búfer interno para mejorar el rendimiento al escribir datos en el archivo. En lugar de escribir cada byte directamente en el archivo, se acumulan en el búfer y luego se escriben en bloques más grandes de una sola vez. Esto reduce la cantidad de operaciones de escritura al sistema de archivos, lo que puede mejorar la eficiencia.
- **new DataOutputStream(...):** Por último, se crea un objeto de tipo `DataOutputStream`, que envuelve al `BufferedOutputStream`. Un `DataOutputStream` es un flujo de salida que puede escribir tipos de datos primitivos y otros datos en un formato binario. Permite escribir datos de diferentes tipos (como `int`, `double`, UTF-8, entre otros) en el archivo de forma estructurada.

En resumen, esta instrucción crea un flujo de datos secuencial, donde primero se envían los datos al `DataOutputStream`, que a su vez envuelve un `BufferedOutputStream`, que a su vez envuelve un `FileOutputStream`, creando así un flujo de datos que apunta al archivo "Factura.txt" para escribir datos en él.

A continuación, se inicia un bucle `for` que recorrerá los arrays `precios`, `cants`, e `items`. En cada iteración del bucle, se escriben los datos en el archivo utilizando los métodos de escritura correspondientes de `DataOutputStream`: `writeDouble()` para escribir el precio (valor decimal de 64 bits), `writeInt()` para escribir la cantidad (valor entero de 32 bits), y `writeUTF()` para escribir el nombre del artículo (cadena de texto UTF-8).

El bloque `try...catch...finally`, se encarga de capturar la excepción de tipo `IOException` durante la escritura en la sección `catch` correspondiente. En este caso, simplemente imprime un mensaje de error indicando "Error E/S".

Luego, en la sección `finally`, que siempre se ejecuta después del bloque `try` (ya sea que ocurra una excepción o no) se cierra el recurso `out` llamando al método `close()` para liberar los recursos y asegurarse de que los datos se escriban correctamente.



Vamos a desglosar cada parte de la cadena de formato de la siguiente instrucción para entender mejor su funcionamiento:

```
System.out.format(" %4d %25s a %6.2f€ c/u %8.2f€%n", cant, item,  
precio, cant * precio);
```

- **%4d:** Especifica que se reemplazará con un número entero (d) ocupando al menos 4 caracteres. Si el número tiene menos de 4 dígitos, se añadirán espacios en blanco a la izquierda para completar los 4 caracteres.
- **%25s:** Especifica que se reemplazará con una cadena (s) ocupando al menos 25 caracteres. Si la cadena tiene menos de 25 caracteres, se añadirán espacios en blanco a la derecha para completar los 25 caracteres.
- **%6.2f€:** Especifica que se reemplazará con un número de punto flotante (f) ocupando al menos 6 caracteres, con 2 decimales. El símbolo "€" indica que se trata de una cantidad monetaria.
- **%8.2f€:** Especifica que se reemplazará con un número de punto flotante (f) ocupando al menos 8 caracteres, con 2 decimales. El símbolo "€" indica que se trata de una cantidad monetaria.
- **%n:** Especifica un carácter de nueva línea, que se utiliza para cambiar de línea en la salida.

El fin de archivo se detecta a través de la captura de la excepción `EOFException` y será entonces cuando se muestre el total de la factura.

Para valores monetarios existe un tipo especial, `java.math.BigDecimal`. No se ha usado en este ejemplo por ser objetos y no tipos primitivos; los objetos no pueden tratarse como flujos de datos, deben tratarse como flujos de objetos.

## 5. FLUJOS DE OBJETOS

*En la reunión matinal de trabajo, el equipo ha optado por utilizar flujos de objetos para gestionar la entrada y salida de datos de la aplicación porque permiten guardar y recuperar objetos completos. La combinación de flujos de objetos y `BigDecimal` garantizará que la aplicación sea capaz de manejar grandes cantidades de datos financieros con la máxima precisión y eficiencia, proporcionando una experiencia de usuario confiable y profesional.*

Los flujos de objetos permiten realizar operaciones de entrada salida de objetos. Muchas de las clases estándar soportan serialización de sus objetos, implementando la interfaz `Serializable`. La serialización de objetos permite guardar el objeto en un archivo escribiendo sus datos en un flujo de bytes. Es posible luego leer desde el archivo el flujo de bytes y reconstruir el objeto original. Las clases de flujos de objetos son `ObjectInputStream` y `ObjectOutputStream`. Estas clases implementan las interfaces `ObjectInput` y `ObjectOutput`, subinterfaces de `DataInput` y `DataOutput`.

En consecuencia, todos los métodos de entrada/salida que estaban disponibles para flujos de datos primitivos estarán implementados también para flujos de objetos.

El siguiente programa implementa la misma aplicación, pero usando objetos `BigDecimal` para los precios, y un objeto `Calendar` para la fecha. Si el método `readObject()` no devuelve el tipo correcto, el casting puede lanzar la excepción `ClassNotFoundException`, lo cual es notificado en el método `main()` mediante la cláusula `throws`.

`BigDecimal` es una clase en Java que implementa la interfaz **`Serializable`**, que proporciona métodos para realizar operaciones aritméticas, como suma, resta, multiplicación y división, con precisión arbitraria. Se utiliza a menudo en aplicaciones financieras y de contabilidad, donde se requiere una precisión exacta en los cálculos. Los objetos `BigDecimal` se crean mediante un constructor que toma un valor numérico como argumento.





La fecha será la fecha del día en que se ejecute el programa:

```
El jueves, agosto 9, 2018, se compró:
4      Marcador Azul a 18,00€ c/u    72,00€
2      Papel A4 500 hojas a 160,00€ c/u 320,00€
1      Borrador a 25,00€ c/u    25,00€
4      DVD a 14,00€ c/u    56,00€
50     Sobres A4 a 2,50€ c/u    125,00€
                                TOTAL 598,00€
```

Existen operaciones de entrada salida sobre archivos que no pueden tratarse como flujos de datos. La clase `File` permite examinar y manipular archivos y directorios, de forma independiente de la plataforma (MS Windows, Linux, MacOS). Los archivos se pueden acceder también en forma no secuencial o aleatoria (random access files); existen clases específicas para acceder a los archivos sin necesidad de recorrerlos ordenadamente.



### RECUERDA

Todos los métodos de entrada/salida que estaban disponibles para flujos de datos primitivos estarán implementados también para flujos de objetos.



### PARA SABER MÁS

Cuando una clase implementa la interfaz **Serializable**, se le permite ser serializada y deserializada utilizando las clases **ObjectOutputStream** y **ObjectInputStream**, respectivamente. Es el caso de la clase **BigDecimal** utilizada en el ejemplo práctico.

Accede a este enlace para saber más sobre la clase `BigDecimal`:



## 6. FICHEROS. CLASE FILE

*Sigues avanzando en la aplicación. Cada cliente tiene un directorio o carpeta en la que se irán almacenando toda la información que le concierne, como facturas, pedidos, etc. Necesitas que se pueda hacer un listado de los archivos que contiene dicho directorio e incluso sería buena idea poder almacenar este listado en un fichero para poder imprimirlo. Después de pensar mucho optas por utilizar la clase File. Te pones manos a la obra.*

Las instancias de la clase **File** representan nombres de archivo, no a los archivos en sí. El archivo correspondiente a un nombre puede no existir.

Un objeto de clase File permite examinar el nombre del archivo, descomponerlo en su rama de directorios, o crear el archivo si no existe pasando el objeto de tipo File a un constructor adecuado como `FileWriter(File f)`. Que recibe como parámetro un objeto File.

Para archivos existentes, a través del objeto File un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos. Estas operaciones pueden hacerse independientemente de la plataforma sobre la que esté corriendo el programa.

Si el objeto File se refiere a un archivo existente un programa puede usar este objeto para realizar una serie de operaciones sobre el archivo:

- **delete()** borra el archivo inmediatamente;
- **deleteOnExit()** lo borra cuando finaliza la ejecución de la máquina virtual Java.
- **setLastModified()** permite fijar la fecha y hora de modificación del archivo:

```
new File("factura.txt").setLastModified(new  
Date().getTime());
```

- **renameTo()** permite renombrar el archivo.
- **mkdir()** crea un directorio, **mkdirs()** también, pero crea los directorios superiores si no existen.

- **list()** y **listFiles()** listan el contenido de un directorio. **list()** devuelve un array de String con los nombres de los archivos, **listFiles()** devuelve un array de objetos File.
- **createTempFile()** crea un nuevo archivo con un nombre único y devuelve un objeto File que apunta a él. Es útil para crear archivos temporales, que luego se borran, asegurándose de tener un nombre de archivo no repetido.
- **listRoots()** devuelve una lista de nombres de archivo correspondientes a la raíz de los sistemas de archivos. En Microsoft Windows serán de formato a:\ y c:\, en UNIX, MacOS y Linux será el directorio raíz único /.



### ARTÍCULO DE INTERÉS

Visualiza esta lectura para ampliar y clarificar conceptos sobre la clase File disponible en Java:



### EJEMPLO PRÁCTICO

En este ejemplo verás la lista de los archivos y subdirectorios que se encuentran en el directorio raíz c:\ por consola, utilizando un bucle for.

```
package flujosbytes;

import java.io.File;

public class Dir {

    public static void main(String[] args) {
        System.out.println("Archivos en el directorio actual: ");
        File ficheros = new File("c:\\");
        String[] archivos = ficheros.list();
        for (int i = 0; i < archivos.length; i++) {
            System.out.println(archivos[i]);
        }
    }
}
```

Es una forma sencilla de obtener una lista de los elementos contenidos en un directorio utilizando la clase File en Java.

Cuando se invoca el método `list()` de la clase File, este devuelve un array de cadenas (`String[]`) que contiene los nombres de los archivos y subdirectorios presentes en el directorio especificado. En este caso, el array `archivos` almacena esta lista de nombres.

Luego, el código utiliza un bucle `for` para recorrer cada elemento en el array `archivos`, que corresponde a los nombres de los archivos y subdirectorios en el directorio. En cada iteración del bucle, el código imprime el nombre del archivo o subdirectorio utilizando `System.out.println()`.

El uso del array es necesario porque `list()` devuelve múltiples nombres de archivos y subdirectorios en el directorio y se utiliza para almacenar temporalmente los nombres de los archivos y subdirectorios obtenidos del directorio al que se hace referencia.



### EJEMPLO PRÁCTICO

El resultado de ejecutar el programa anterior podría ser, por ejemplo:

```
Archivos en el directorio actual:  
$Recycle.Bin  
$SysReset  
Archivos de programa  
Documents and Settings  
hiberfil.sys  
Intel  
pagefile.sys  
PerfLogs  
Program Files  
Program Files (x86)  
ProgramData  
Recovery  
swapfile.sys  
swsetup  
System Volume Information  
System.sav  
Users  
Windows
```



### NOTA DE INTERÉS

En próximas unidades se verán en profundidad los arrays.

De momento solo tienes que entender para que se han usado en los ejemplos, pero sin tratar de profundizar todavía mucho más para centrarte en las operaciones de E/S con ficheros que se tratan en esta unidad.

## 7. FICHEROS DE ACCESO ALEATORIO. CLASE RANDOMACCESSFILE

*Estás muy contento, la aplicación va tomando forma, ahora vas a aumentar la complejidad y quieres poder acceder aleatoriamente a diferentes partes de un archivo y realizar operaciones de lectura y escritura en esas ubicaciones específicas, a diferencia de las clases de flujo de entrada/salidas convencionales, que leen o escriben datos secuencialmente, la clase RandomAccessFile permite buscar, leer y escribir datos en cualquier posición dentro del archivo.*

Un **archivo de acceso aleatorio** permite leer o escribir datos en forma no secuencial. El contenido de un archivo suele consistir en un conjunto de partes o registros, generalmente de distinto tamaño.

La **búsqueda de información en el archivo** equivale a ubicar un determinado registro. En el **acceso secuencial** es preciso leer el archivo pasando por todos sus registros hasta llegar al registro que se desea ubicar. En promedio debe leerse la mitad del archivo en cada búsqueda. Si el tamaño de los registros es conocido puede crearse un **índice** con punteros hacia cada registro. La búsqueda de un registro comienza entonces por ubicar ese registro en el índice, obtener un puntero hacia el lugar del archivo donde se encuentra el contenido de ese registro, y desplazarse hacia esta posición directamente. El acceso aleatorio descrito es **mucho más eficiente** que el acceso secuencial.

### CLASE RANDOMACCESSFILE

La clase **java.io.RandomAccessFile** implementa las interfaces **DataInput** y **DataOutput**, lo cual permite leer y escribir en el archivo. Para usar **RandomAccessFile** se debe indicar un **nombre de archivo** para abrir o crear si no existe. Se debe indicar también si se abrirá para **lectura** o también para **escritura** (para poder escribir es necesario también poder leer).

La siguiente sentencia abre un archivo de nombre `archiuno.txt` para lectura, y la siguiente abre el archivo `archidos.txt` para lectura y escritura:

```
//Archivo de solo lectura
RandomAccessFile f1 = new RandomAccessFile("archiuno.txt","r");
//Archivo de lectura/escritura
RandomAccessFile f2 = new RandomAccessFile("archidos.txt","rw");
```

Una vez abierto el archivo pueden usarse los métodos **read()** o **write()** definidos en las interfaces **DataInput** y **DataOutput** para realizar operaciones de entrada/salida sobre los archivos.

La clase **RandomAccessFile** maneja un puntero al archivo (**file pointer**). Este puntero indica la **posición actual** en el archivo. Cuando el archivo se crea, el puntero al archivo se coloca en el 0, apuntando al principio del archivo. Las sucesivas llamadas a los métodos **read()** y **write()** ajustan el puntero según la cantidad de bytes leídos o escritos.

Además de los métodos de entrada/salida que ajustan el puntero automáticamente, la clase **RandomAccessFile** tiene métodos específicos para manipular el puntero al archivo:

- **int skipBytes(int):** mueve el puntero hacia adelante la cantidad especificada de bytes.
- **void seek(long):** ubica el puntero justo antes del byte especificado en el entero long.
- **long getFilePointer():** devuelve la posición actual del puntero, el número de byte indicado por el entero long devuelto.



### VÍDEO DE INTERÉS

Visualiza este interesante vídeo sobre cómo escribir y leer un fichero con RandomAccessFile.



El siguiente ejemplo muestra el uso de un archivo de acceso aleatorio y el valor de los punteros. Crea una tabla de raíces cuadradas de los números del 0 al 9 expresada como decimales doble precisión tipo `double`, de tamaño 8 bytes.

Realizar las siguientes tareas:

- Calcular los cuadrados, guardarlos en un archivo de acceso aleatorio y cerrar el archivo.
- Abrir el archivo recién creado, desplazar el puntero 40 bytes (5 `double` de 8 bytes cada uno),
- Leer el registro ubicado a partir del byte 40 (raíz cuadrada del número 5: 2,23...),
- Verificar el avance del puntero a 48 (avanzó un `double` en la lectura), cambiar su valor por el número arbitrario 333.0003 y cerrar el archivo.
- Abrir nuevamente el archivo, ahora en sólo lectura, y mostrar punteros y valores.
- Intentar escribir en el archivo de sólo lectura, lanzando y capturando la excepción.

```

package aleatorio;

import java.io.IOException;
import java.io.RandomAccessFile;

public class Aleatorio {

    public static void main(String[] args) {
        RandomAccessFile rf = null;
        try {
            rf = new RandomAccessFile("dobles.dat", "rw");
            for (int i = 0; i < 10; i++) {
                rf.writeDouble(Math.sqrt(i));
            }
            rf.close();
        } catch (IOException e) {
            System.out.println("Error de E/S parte 1:\n"+e.getMessage());
        }
        try {
            rf = new RandomAccessFile("dobles.dat", "rw");
            rf.seek(5*8); //Avanza el puntero 5 registros * 8 bytes
            System.out.println("\nPuntero antes de lectura: "+rf.getFilePointer());
            System.out.println(" Valor:"+rf.readDouble());
            System.out.println("\nPuntero después de lectura: "+rf.getFilePointer());
            rf.seek(rf.getFilePointer()-8); //Restaura el puntero a registro 6
            System.out.println("\nPuntero restaurado: "+rf.getFilePointer()+"\n");
            rf.writeDouble(333.0003); //Cambia el registro 6 y avanza el puntero
            rf.close();
        } catch (IOException e) {
            System.out.println("Error de E/S parte 2:\n"+e.getMessage());
        }
    }

    try {
        rf = new RandomAccessFile("dobles.dat", "r");
        for (int i = 0; i < 10; i++) {
            System.out.print("\nPuntero en: "+rf.getFilePointer()+" ");
            System.out.println("valor: "+rf.readDouble());
        }
        rf.writeDouble(1.111); //Intento de escribir en archivo de solo lectura
        rf.close();
    } catch (IOException e) {
        System.out.println("Error de E/S parte 3:\n"+e.getMessage());
    }
}
}

```



Resultado de la ejecución del programa:

```
Puntero antes de lectura: 40
Valor:2.23606797749979

Puntero después de lectura: 48

Puntero restaurado: 40


Puntero en: 0 valor: 0.0
Puntero en: 8 valor: 1.0
Puntero en: 16 valor: 1.4142135623730951
Puntero en: 24 valor: 1.7320508075688772
Puntero en: 32 valor: 2.0
Puntero en: 40 valor: 333.0003
Puntero en: 48 valor: 2.449489742783178
Puntero en: 56 valor: 2.6457513110645907
Puntero en: 64 valor: 2.8284271247461903
Puntero en: 72 valor: 3.0
Error de E/S parte 3:
Acceso denegado
```



### ENLACE DE INTERÉS

En este enlace podrás profundizar más sobre la entrada y salida con Java:





### PARA SABER MÁS

Conoce más sobre los ficheros Java a través de estos ejercicios resueltos:



### ENLACE DE INTERÉS

Visualiza esta lista de vídeos con ejemplos y ejercicios resueltos de ficheros en Java:



## 8. INTERFACES GRÁFICAS

*Ya tienes todas las funcionalidades más importantes realizadas y funcionan perfectamente por tanto tu jefe de proyecto te encomienda la tarea de elaborar una interfaz gráfica que sea amigable e intuitiva para el usuario que la maneje. Te recomienda utilizar la tecnología JavaFx que es un conjunto de bibliotecas y API que proporciona Java y permite el desarrollo de interfaces gráficas ricas y modernas tanto para aplicaciones de escritorio como para móviles.*

JavaFX, también conocido como OpenJFX, es un software gratuito; con licencia bajo GPL con la excepción del class path, al igual que OpenJDK. Es una plataforma de desarrollo de aplicaciones cliente y un conjunto de bibliotecas gráficas y multimedia para Java de código abierto que permite a los desarrolladores crear aplicaciones gráficas interactivas y atractivas para escritorio, web y dispositivos móviles. JavaFX proporciona una amplia variedad de controles, gráficos, animaciones, efectos visuales y capacidades multimedia que facilitan la creación de interfaces de usuario modernas y sofisticadas. Es una tecnología moderna para el desarrollo de interfaces gráficas en Java y requiere una versión Java 8 o posterior.

Algunas características y ventajas de JavaFX son:

- **Facilidad de uso:** JavaFX proporciona una API intuitiva y fácil de usar para el desarrollo de interfaces gráficas.
- **Diseño declarativo:** Permite la construcción de interfaces gráficas mediante archivos FXML, que representan la estructura de la interfaz de manera declarativa y separada del código Java.
- **Gráficos avanzados:** Ofrece soporte para gráficos 2D y 3D, lo que permite crear visualizaciones y efectos visuales impresionantes.
- **Multimedia:** Proporciona capacidades multimedia para reproducir audio y video, lo que es útil para aplicaciones de reproducción de medios.
- **Efectos y animaciones:** Permite crear animaciones y efectos visuales de manera sencilla, lo que mejora la experiencia del usuario.
- **Soporte multiplataforma:** Las aplicaciones JavaFX pueden ejecutarse en diferentes sistemas operativos y dispositivos, ya que está basado en Java, lo que proporciona portabilidad y reutilización de código.

- **Interoperabilidad con Swing:** JavaFX se integra bien con las aplicaciones existentes que utilizan Swing, lo que permite una migración gradual a la nueva plataforma.

Durante mucho tiempo, la tecnología más utilizada para el desarrollo de interfaces gráficas de usuario ha sido Swing, que se compone de un conjunto de bibliotecas gráficas y componentes de interfaz de usuario para Java.

Sin embargo, a pesar de todas sus ventajas, Swing ha sido reemplazado en gran medida por JavaFX como la tecnología recomendada para el desarrollo de interfaces gráficas en Java, aunque sigue siendo una opción válida para ciertos proyectos, especialmente aquellos que requieren una mayor compatibilidad con versiones anteriores o que no necesitan características avanzadas de gráficos y multimedia, además sigue contando con una amplia comunidad de desarrolladores. Esto significa que hay abundante documentación, tutoriales y recursos de aprendizaje disponibles en línea.

JavaFX ofrece una experiencia más moderna y enriquecida, con soporte nativo para gráficos 2D/3D, animaciones, multimedia y una mayor flexibilidad de diseño. Aunque Swing sigue siendo compatible y funcional, JavaFX es la tecnología que utilizaremos en la unidad.

## 8.1 Preparación del entorno

Para utilizar JavaFX, necesitas tener instalado Java Development Kit (JDK) en tu sistema (esto ya lo habrás hecho para probar los ejemplos propuestos en unidades anteriores).

Asegúrate de tener instalada una versión de JDK que sea compatible con JavaFX. A partir de JDK 8, JavaFX venía incluido en la distribución estándar, por lo que no necesitabas descargarlo por separado. Sin embargo, a partir de JDK 11, JavaFX se ha separado del JDK estándar y se distribuye como un módulo independiente.

A continuación, se muestran los pasos para usar JavaFX con versiones a partir de JDK 11 en adelante:

### Instalación del JDK

Descarga e instala el JDK desde el sitio oficial de Oracle o desde el proveedor de tu preferencia.



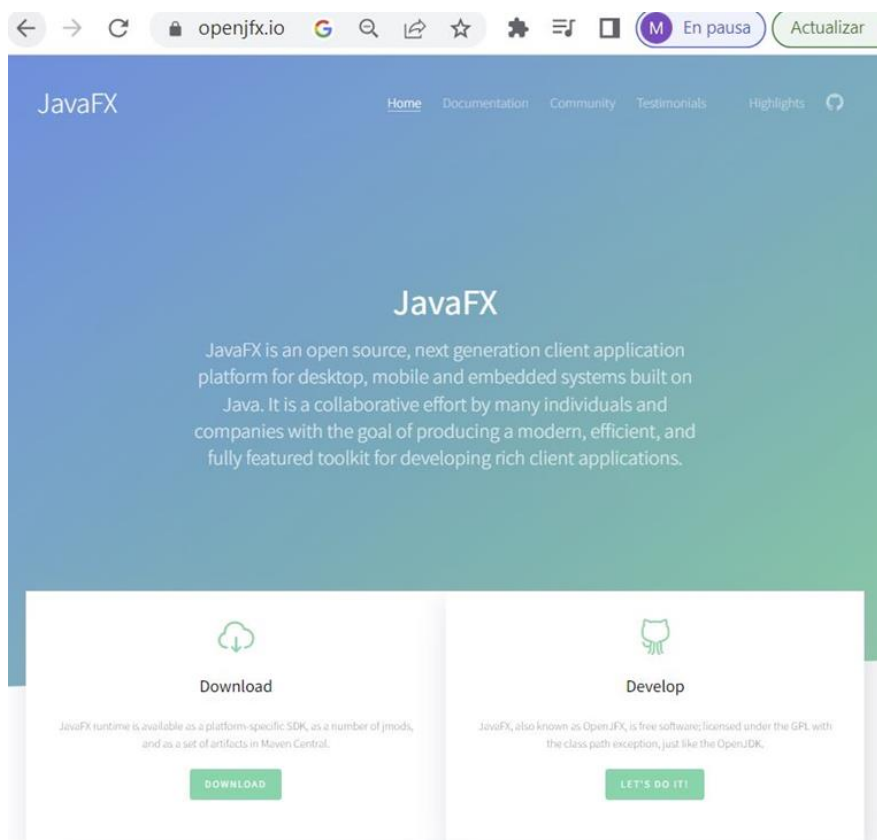
## ENLACE DE INTERÉS

Aquí tienes una página para descarga, por si todavía no lo has hecho.



### Instalación de JavaFX SDK

Descarga el SDK de JavaFX desde el sitio oficial de OpenJFX.



Instalación JavaFX SDK.

Fuente: Elaboración propia.



## ENLACE DE INTERÉS

En este enlace tienes acceso a la descarga de JavaFx:



Hacemos clic en el botón Download y en la nueva página que aparece bajamos un poco hasta encontrar la sección Downloads.

gluonhq.com/products/javafx/

GLUON Products Developers Pricing Services Insights Contact

### Downloads

JavaFX version: 20.0.2 Operating System: [any] Architecture: [any] Type: [any]

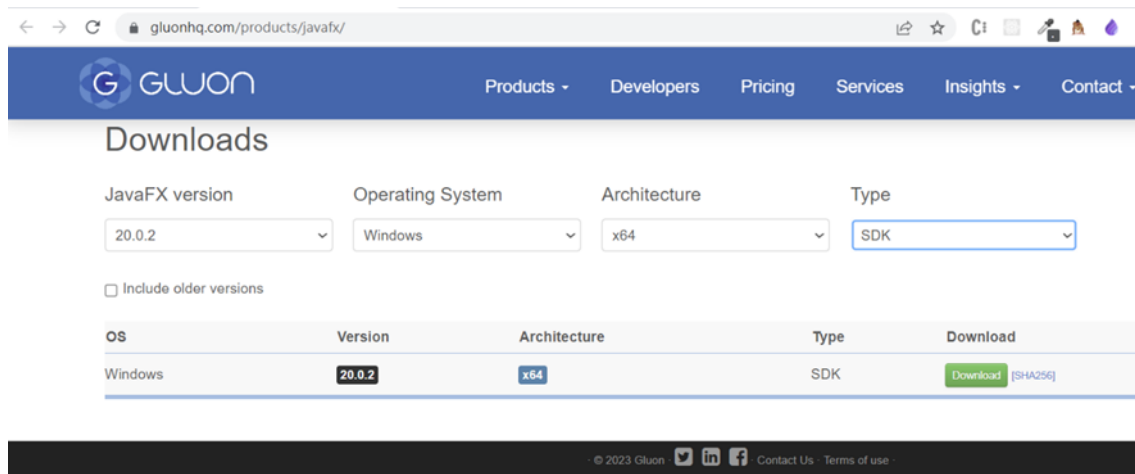
☐ Include older versions

OS	Version	Architecture	Type	Download
Linux	20.0.2	aarch64	SDK	<a href="#">Download</a> [SHA256]
Linux	20.0.2	aarch64	jmods	<a href="#">Download</a> [SHA256]
Linux	20.0.2	aarch64	Monocle SDK	<a href="#">Download</a> [SHA256]
Linux	20.0.2	arm32	SDK	<a href="#">Download</a> [SHA256]
Linux	20.0.2	x64	SDK	<a href="#">Download</a> [SHA256]
Linux	20.0.2	x64	jmods	<a href="#">Download</a> [SHA256]
macOS	20.0.2	aarch64	SDK	<a href="#">Download</a> [SHA256]
macOS	20.0.2	aarch64	jmods	<a href="#">Download</a> [SHA256]
macOS	20.0.2	aarch64	Monocle SDK	<a href="#">Download</a> [SHA256]

Instalación JavaFX SDK.

Fuente: Elaboración propia.

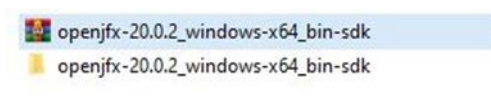
En esta sección seleccionamos los parámetros adecuados según el sistema operativo y su arquitectura y en Type seleccionamos SDK y clic en Download.



Instalación JavaFX SDK.

Fuente: Elaboración propia.

En la carpeta de descargas tendremos el siguiente archivo, que tenemos que descomprimir (más adelante nos hará falta).



Archivo descargado y descomprimido OpenJFX.

Fuente: Elaboración propia.

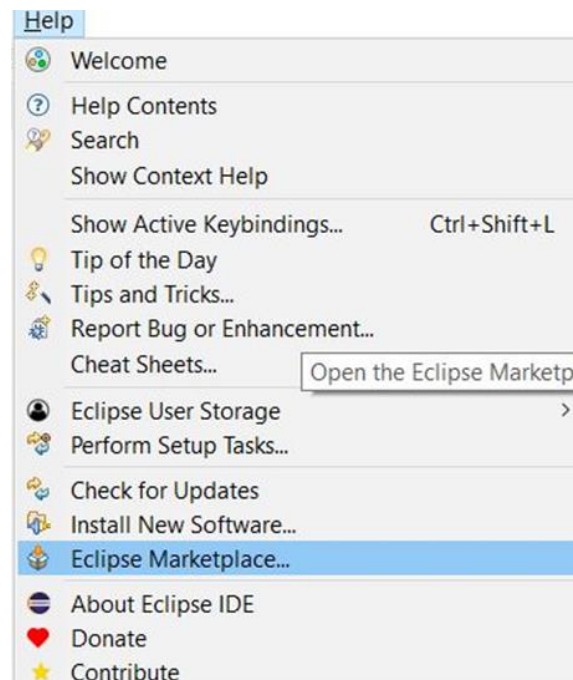
## 8.2 Configuración de Eclipse

Como Oracle decidió dejar de incluir las librerías de JavaFX en el kit de desarrollo Java(JDK), como se comentó anteriormente, para construir aplicaciones usando JavaFX es necesario configurar Eclipse y nuestro proyecto para que se reconozcan estas librerías.

### Instalación de JavaFX plugin

El JavaFX plugin, también conocido como JavaFX Gradle plugin, es una herramienta que permite integrar y facilitar el desarrollo de aplicaciones JavaFX dentro del sistema de construcción y administración de dependencias Gradle, proporciona tareas y configuraciones específicas y gestión de dependencias para proyectos que utilizan JavaFX, simplificando la configuración y compilación.

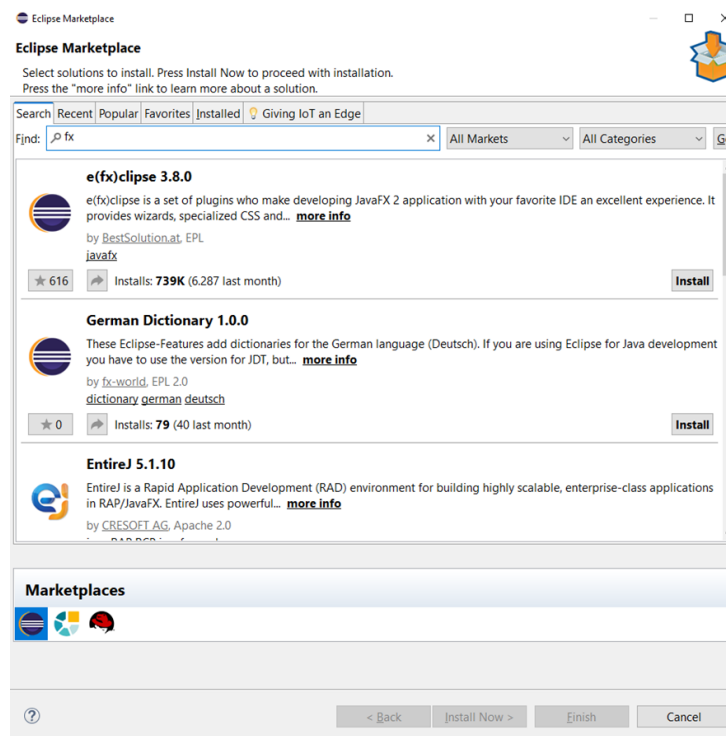
Abrimos Eclipse y en la opción del menú Help escogemos Eclipse Marketplace.



Instalación JavaFX SDK.

Fuente: Elaboración propia.

Escribimos FX en el cuadro de búsqueda Find, aparecerá un listado de software y en la fila donde aparece e(fx)clipse 3.8.0 (versión en el momento de la edición del temario) hacemos clic en botón install.



Instalación JavaFX SDK.

Fuente: Elaboración propia.





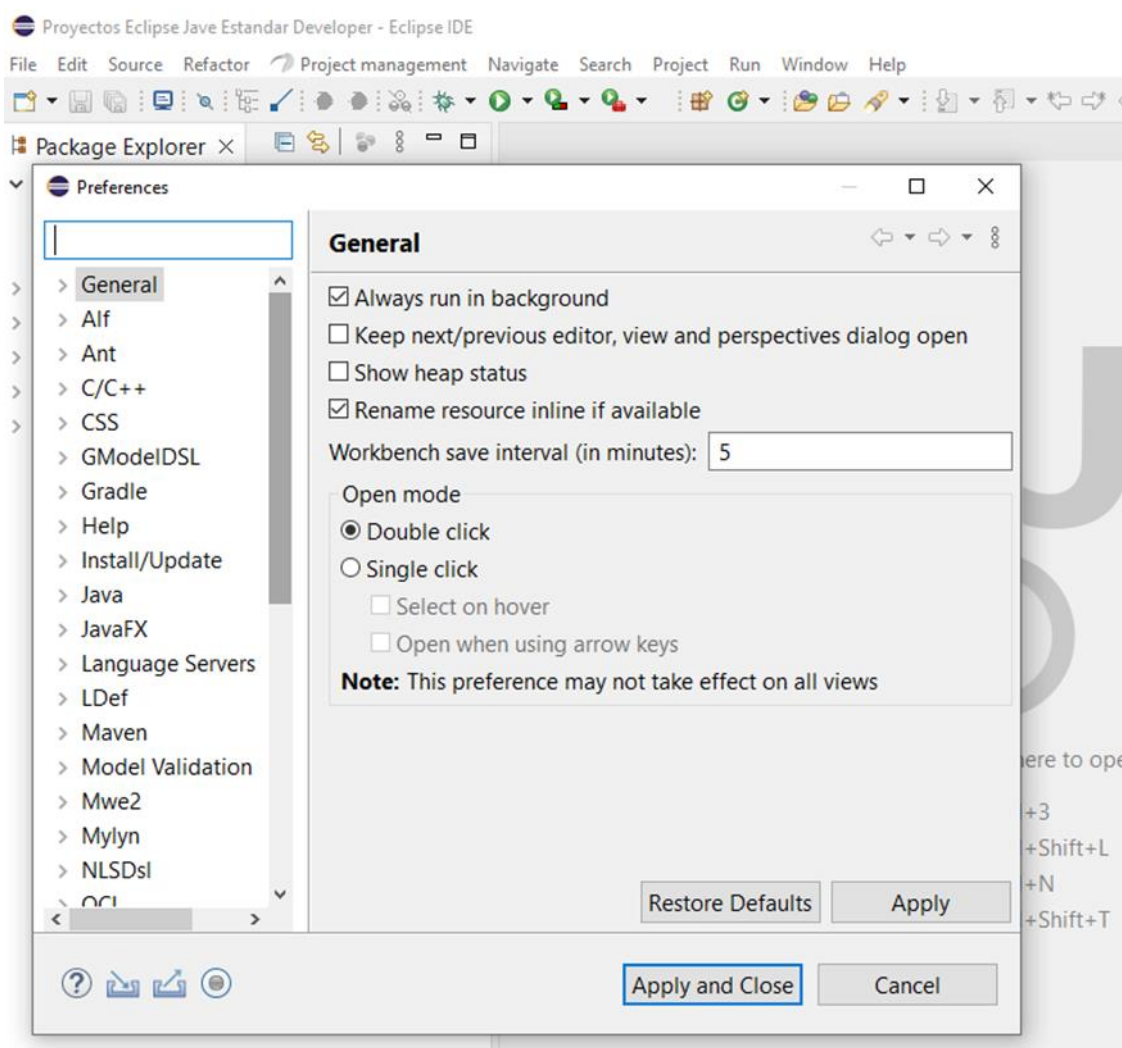
### ¿SABÍAS QUE...?

**Gradle** es una herramienta de construcción y automatización de proyectos que permite gestionar de manera eficiente las dependencias, la compilación, el empaquetado y otras tareas relacionadas con el desarrollo de software.

## 8.3 Creando una librería de usuario

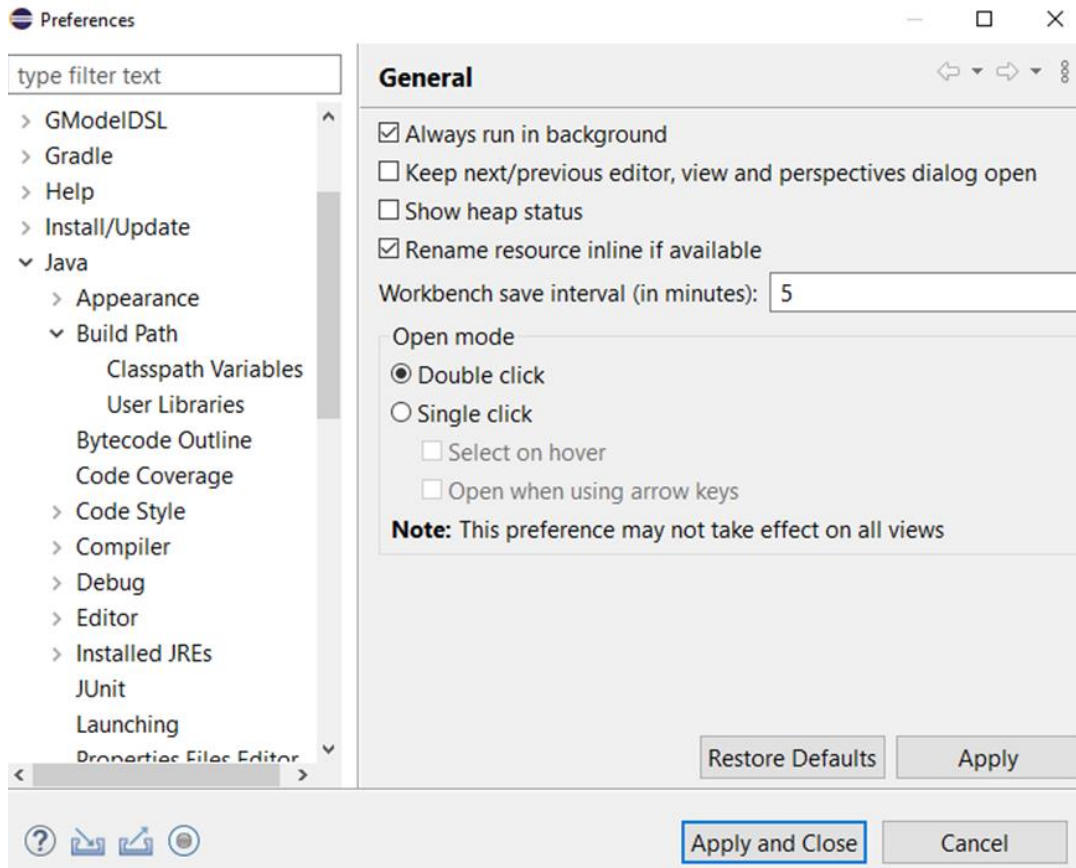
Accede a la barra de menú siguiente:

Windows / Preferences / Java / Build Path / User libraries



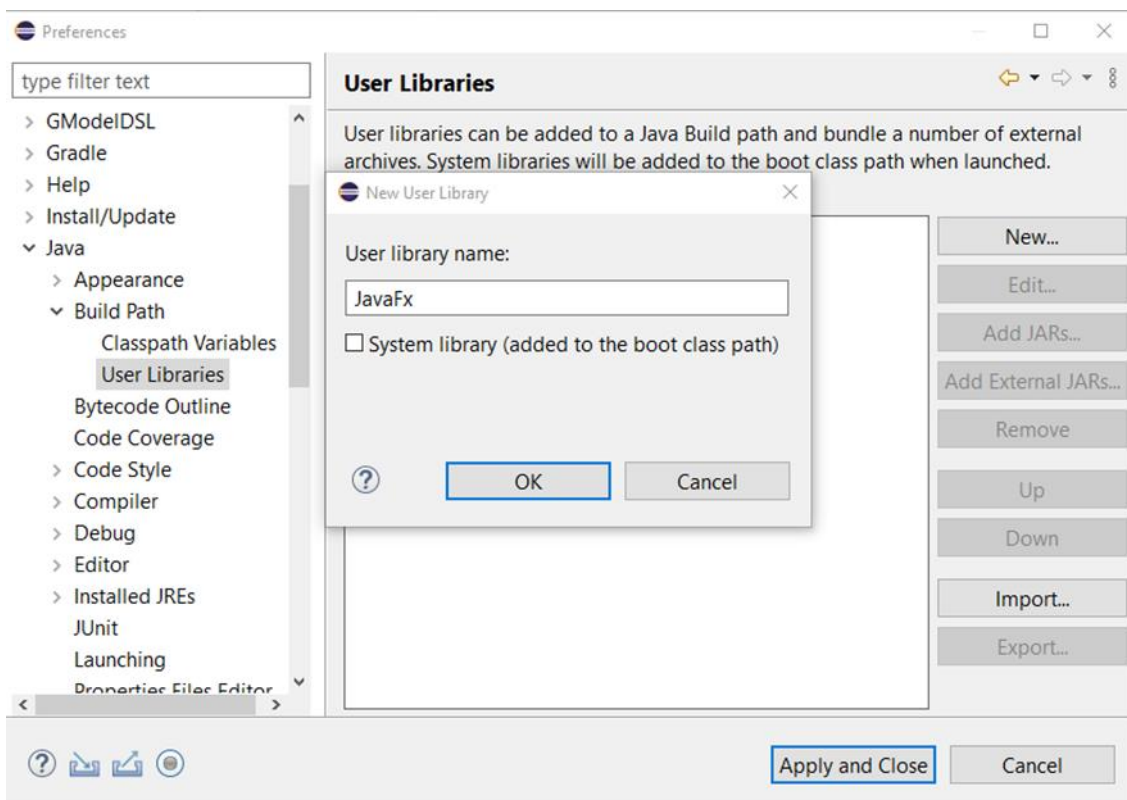
Creando librería de usuario en Eclipse para JavaFX-Menú Preferences.

Fuente: Elaboración propia.



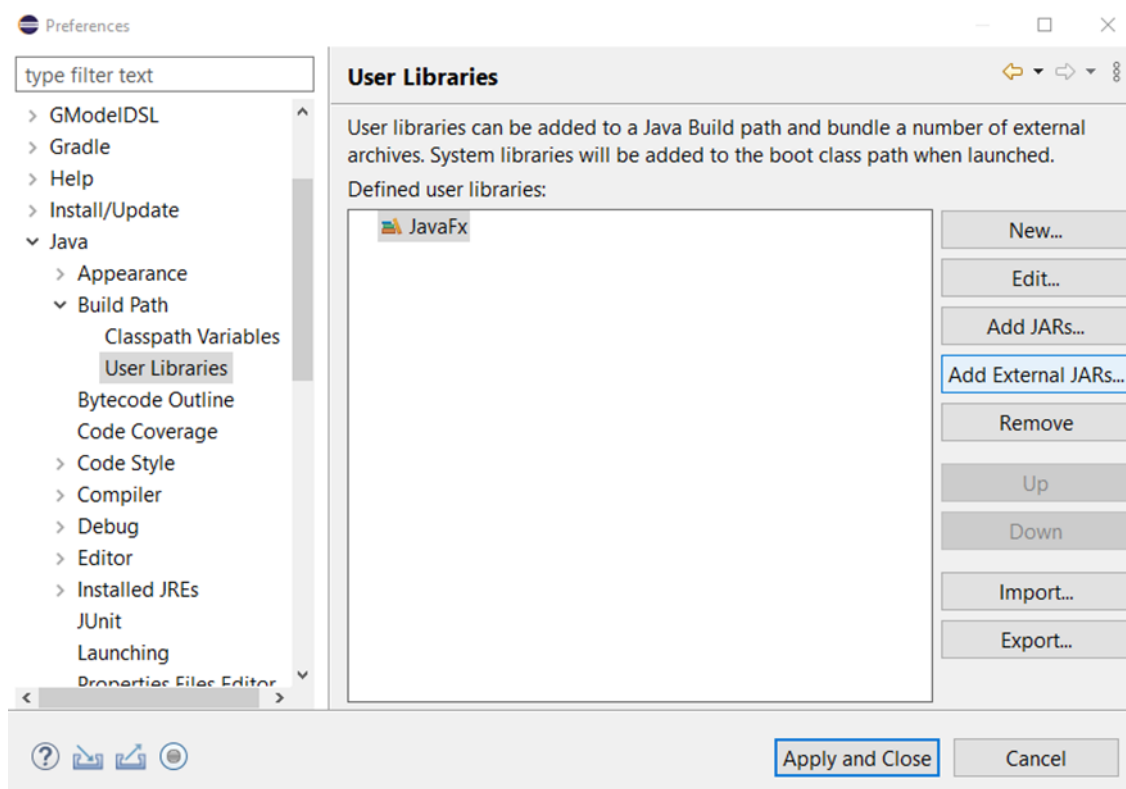
Creando librería de usuario en Eclipse para JavaFX-Menú Java.

Fuente: Elaboración propia.



Creando librería de usuario en Eclipse para JavaFX-User Libraries.

Fuente: Elaboración propia.

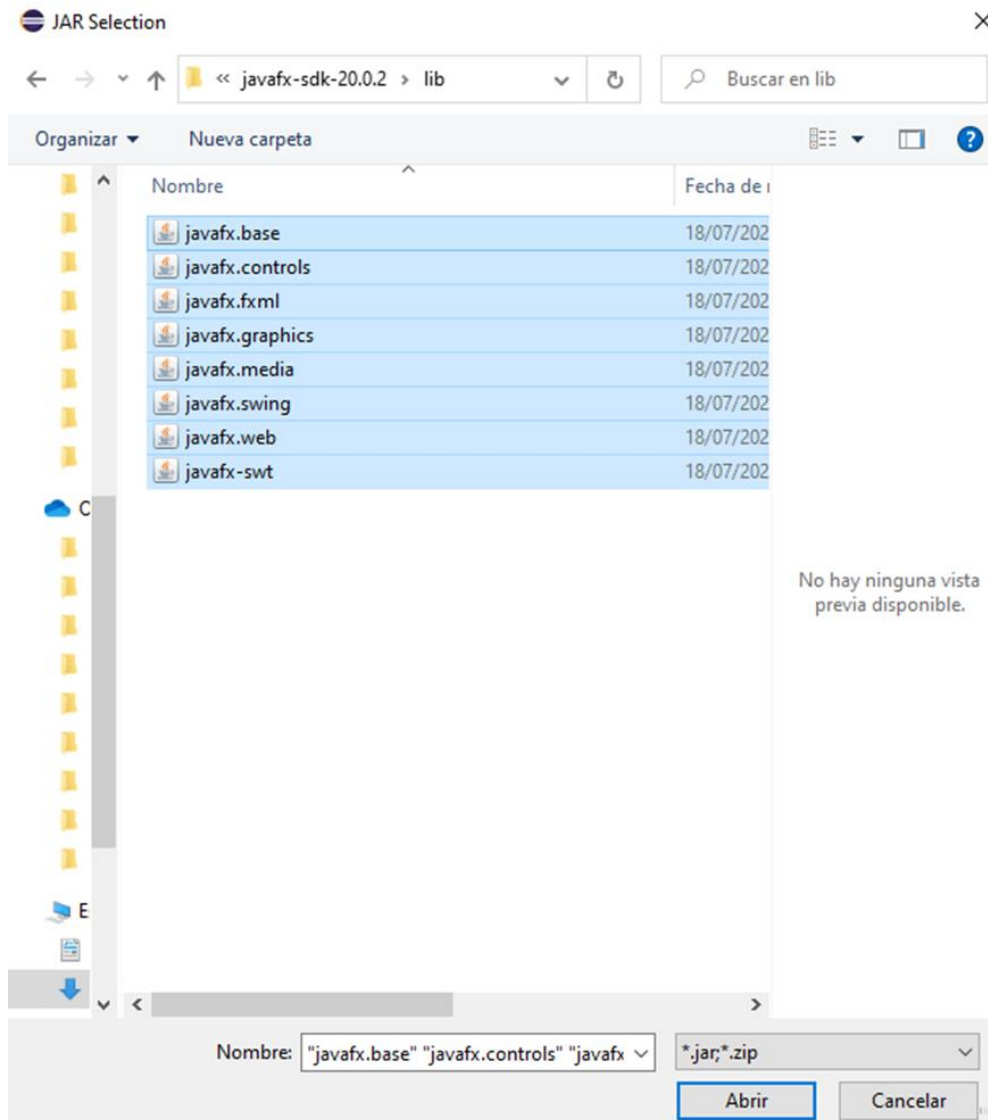


Creando librería de usuario en Eclipse para JavaFX-Add External JAR.

Fuente: Elaboración propia.

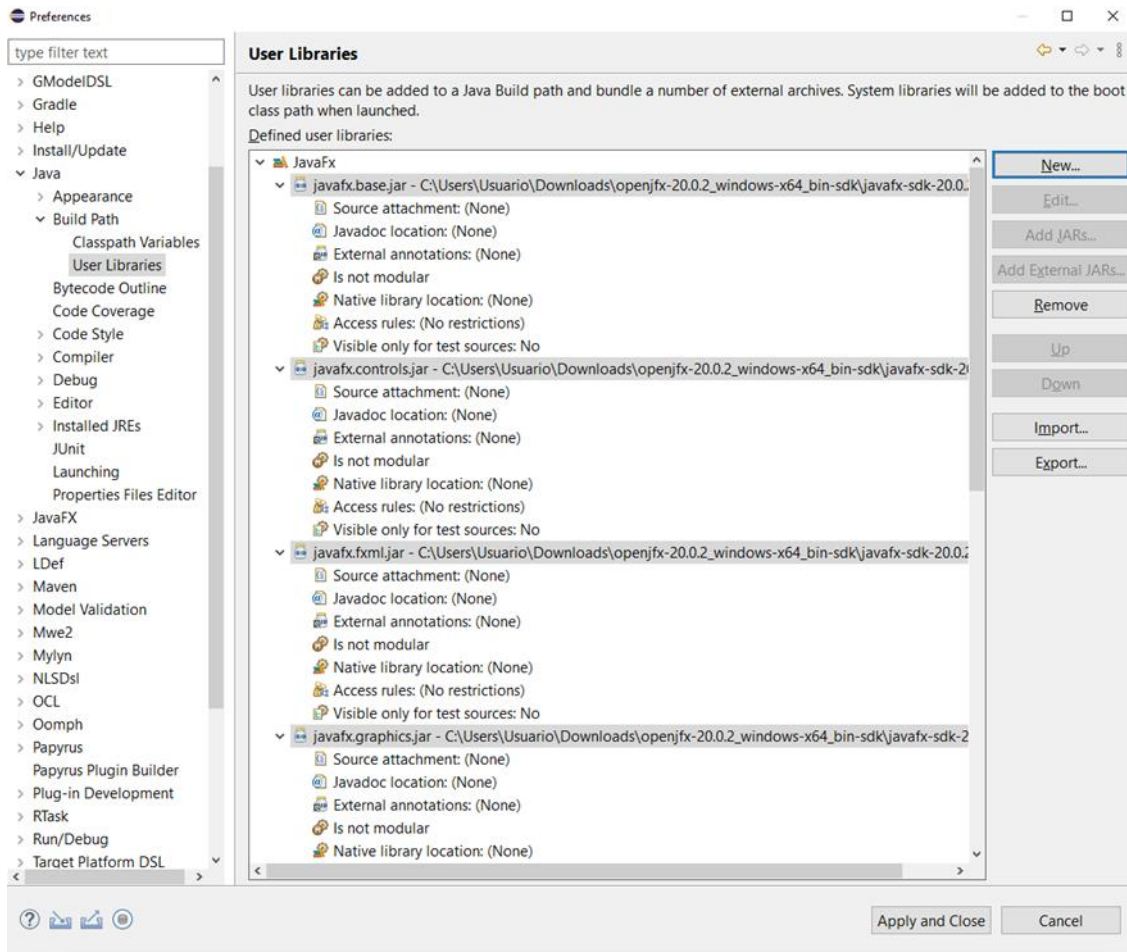
Seleccionamos la librería y hacemos clic en el botón Add External JARs.

Buscamos la carpeta donde se había descargado JavaFX SDK y accedemos a la carpeta lib para seleccionar todas las librerías como se muestra en la imagen.



Creando librería de usuario en Eclipse para JavaFX-librerías a incluir.

Fuente: Elaboración propia.



Y, por último, clic en el botón **Apply and Close**.



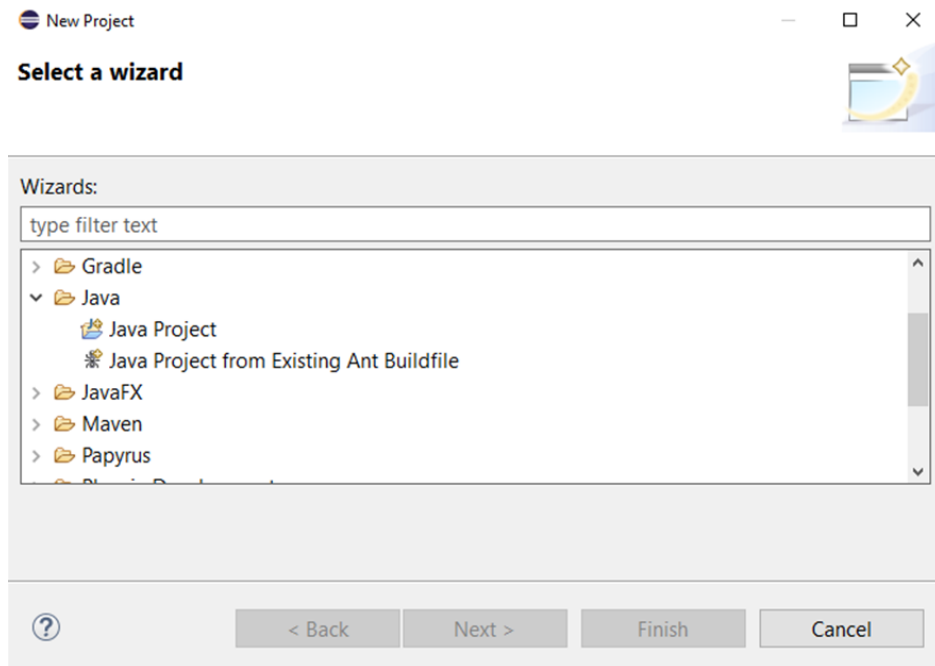
### RECUERDA

Si mueves la carpeta, en la cual tienes las librerías de JavaFx, a otro lugar o la borras, debes volver a incluirlas de nuevo (vuelve a realizar los pasos anteriores), para indicarle a Eclipse dónde debe buscar estas clases, en caso contrario no las reconoce y tendrás errores en el programa al hacer referencia a ellas.

## 8.4 Creando un nuevo proyecto JavaFX project

Accede a la barra de menú siguiente:

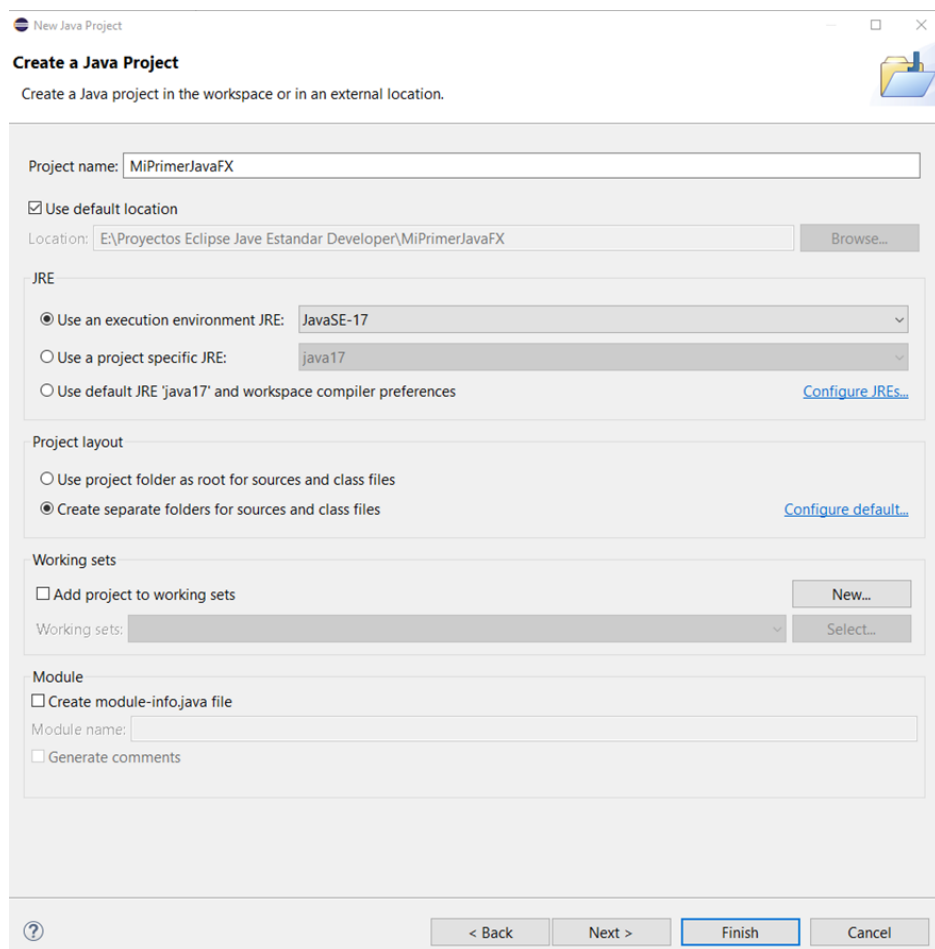
File / New / Project / JavaFX Project



Creando un nuevo proyecto JavaFX.

Fuente: Elaboración propia.

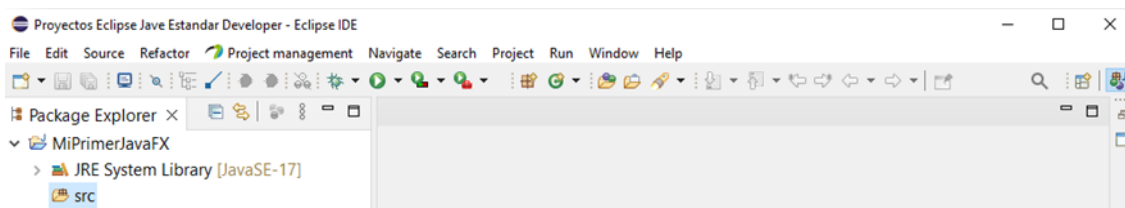
Introduce un nombre al proyecto, por ejemplo, MiPrimerJavaFX y a continuación Finish.



Creando un nuevo proyecto JavaFX.

Fuente: Elaboración propia.

En la parte izquierda de la pantalla aparecerá un nuevo proyecto con este título.



Creando un nuevo proyecto JavaFX.

Fuente: Elaboración propia.

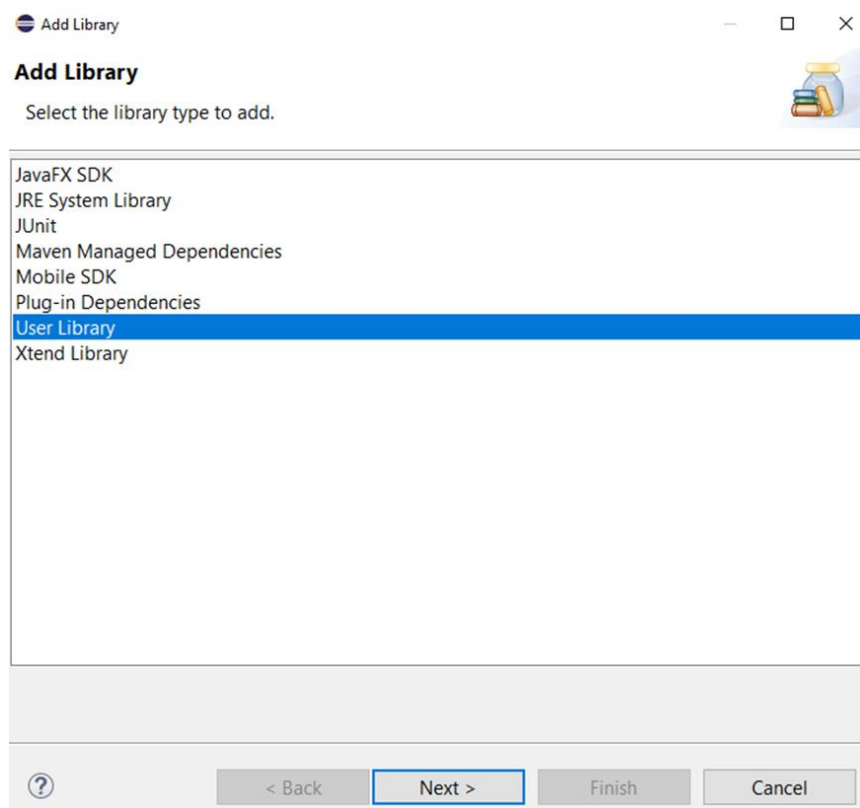
## 8.5 Añadir la librería de usuario

En la parte izquierda, dentro del menú Package explorer, tenemos el proyecto creado y se va a proceder a añadir la librería de usuario creada en apartados anteriores.

Para ello, hacemos clic con el derecho del ratón sobre nuestro proyecto que figura en la parte izquierda dentro del menú Package explorer y vamos siguiendo el siguiente orden:

Build Path / Libraries / User Library

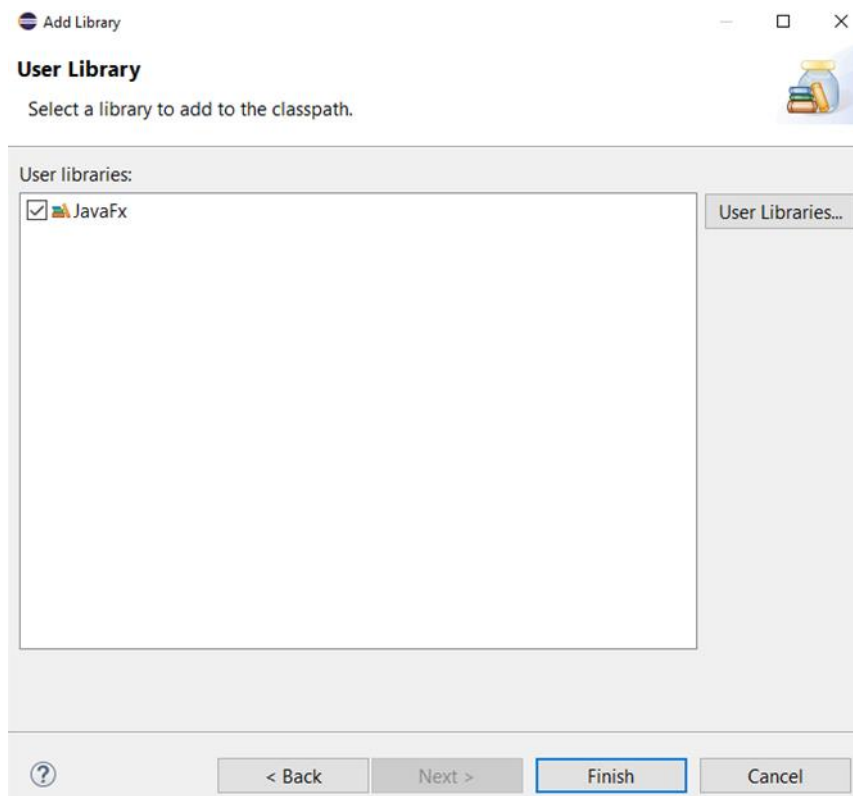
Clic en Next.



Añadiendo librería al proyecto JavaFX .

Fuente: Elaboración propia.

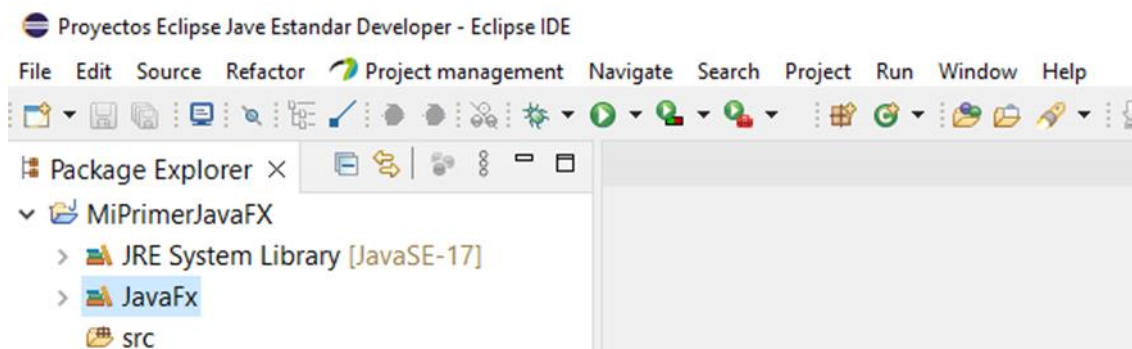




Añadiendo librería al proyecto JavaFX.

Fuente: Elaboración propia.

Seleccionamos la librería JavaFX y para finalizar **Finish** y podemos observar que la librería JavaFx ha sido añadida al proyecto.

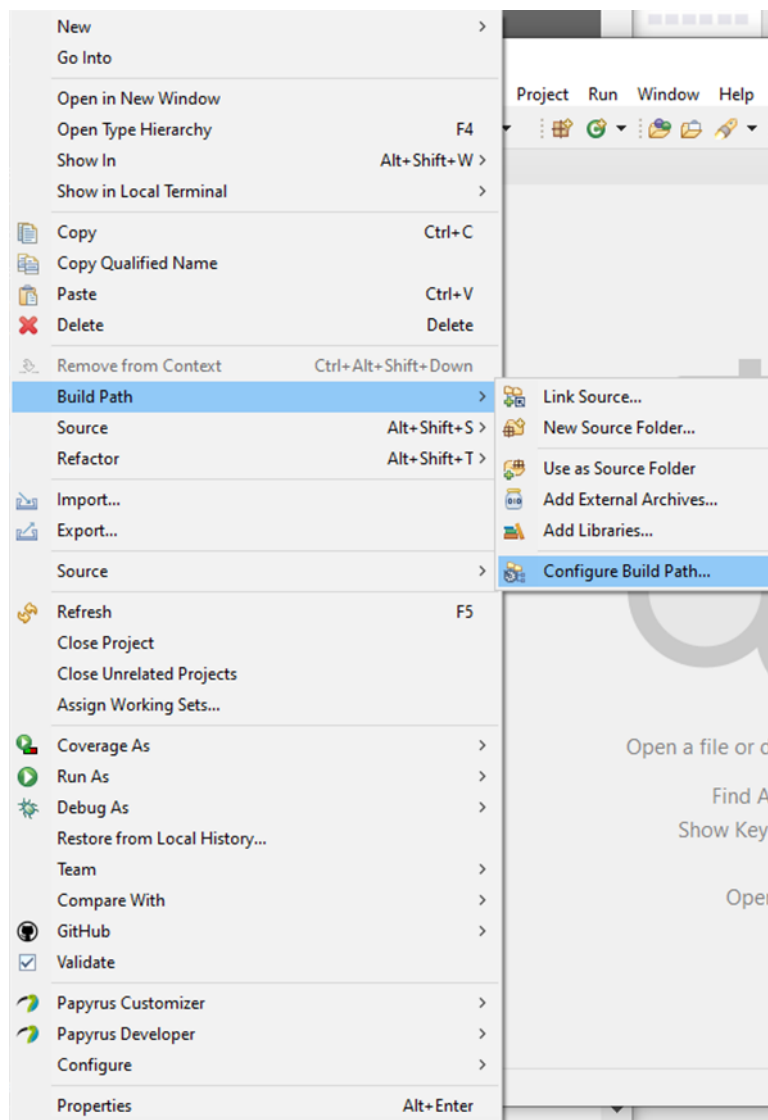


Librería añadida al proyecto JavaFX.

Fuente: Elaboración propia.

## 8.6 Configurando el Build Path

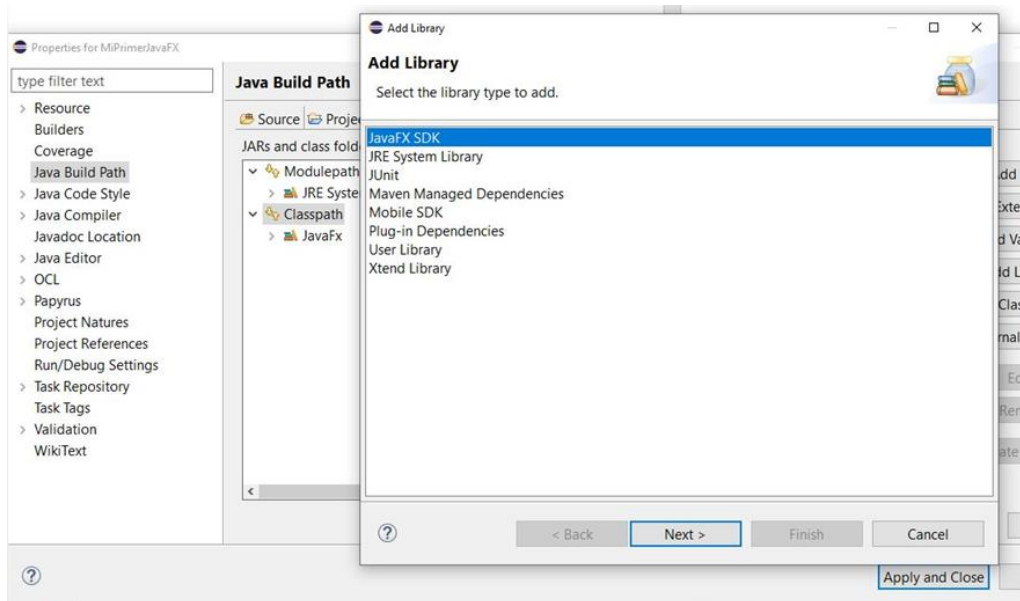
En la parte izquierda, dentro de la ventana Package explorer, hacemos un clic derecho en el nuevo proyecto y escogemos Build Path / Configure Build Path...



Configurando el Build Path para JavaFX

Fuente: Elaboración propia

Se debe comprobar que en ClassPath existen JavaFX y JavaFX SDK, si no está JavaFX SDK entonces debemos añadir la librería como se puede ver en la imagen.

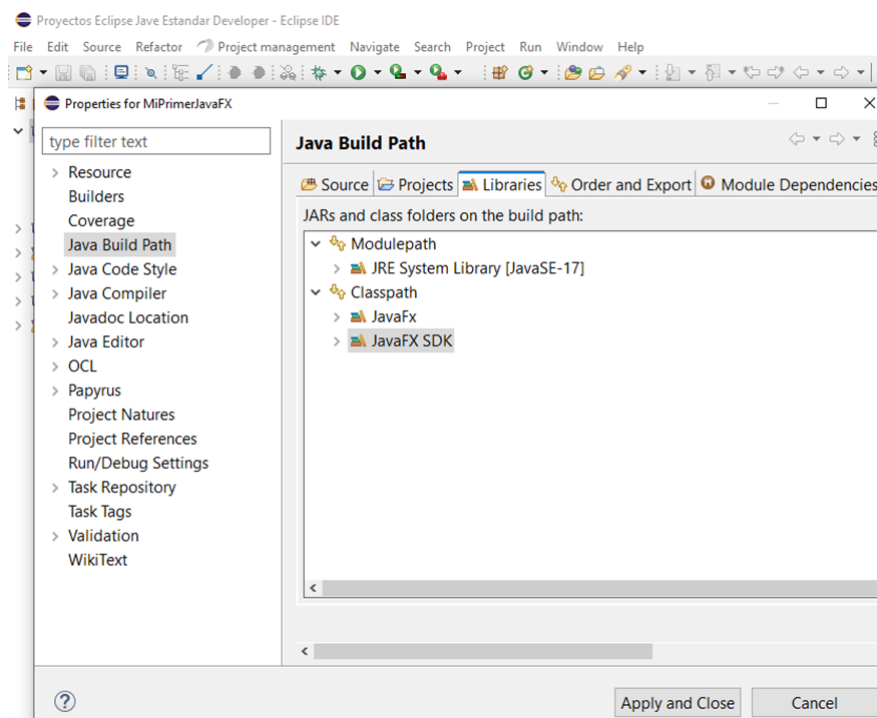


Añadiendo librería JavaFX SDK al Build Path.

Fuente: Elaboración propia.

Clic en Next y después en Finish

Se ha añadido la librería, para terminar clic en Apply and Close.



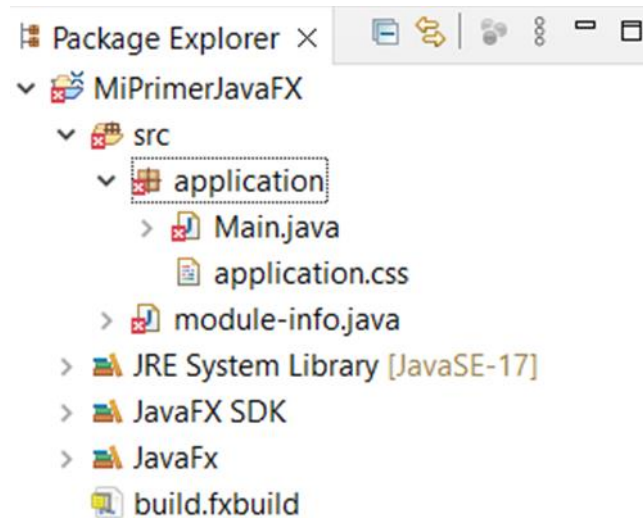
Librerías añadidas al proyecto JavaFX

Fuente: Elaboración propia.

El proceso de crear y añadir las librerías se puede hacer en el mismo momento que creamos el proyecto, pero es interesante que conozcas como hacerlo una vez creado el proyecto.

Si todo ha ido bien, deberías tener los siguientes archivos que se han creado por defecto, si no es así, elimina el proyecto y créalo de nuevo añadiendo las librerías indicadas.

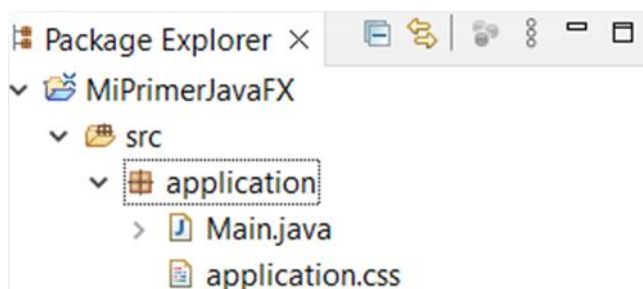
Si tienes algún error prueba a borrar el archivo **module-info.java**.



Ficheros creados por defecto en proyecto JavaFX.

Fuente: Elaboración propia.

Procedemos con el borrado de module-info.java (de momento no lo utilizamos) y desaparece el error.



Borrado de archivo module-info.java en proyecto JavaFX.

Fuente: Elaboración propia.

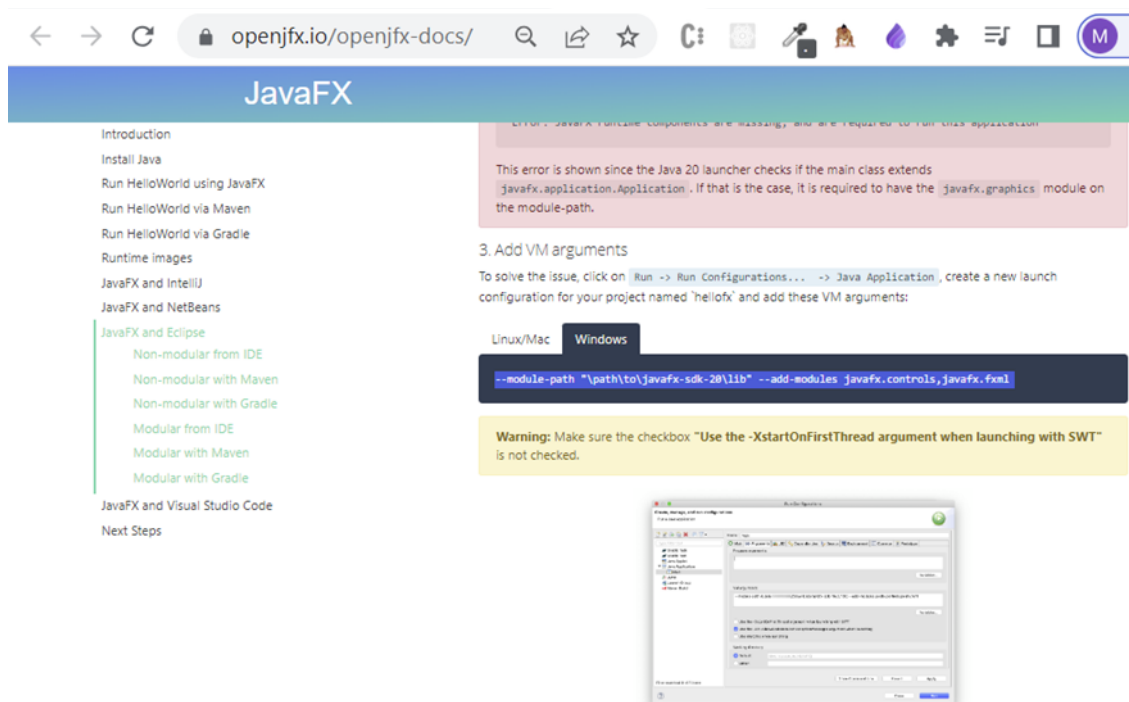
## 8.7 Configurando Máquina virtual para usar JavaFX

Para poder ejecutar nuestro proyecto necesitamos tener configurada la VM (iniciales de la máquina virtual en inglés) para que tenga los módulos de JavaFX que utilizaremos, esto lo haremos de la siguiente manera.

Accedemos a la siguiente dirección web:



Seleccionamos del menú JavaFX and Eclipse del menú que aparece en la parte izquierda, para copiar la instrucción sombreada de azul en caso de que nuestro sistema sea el sistema operativo Windows.

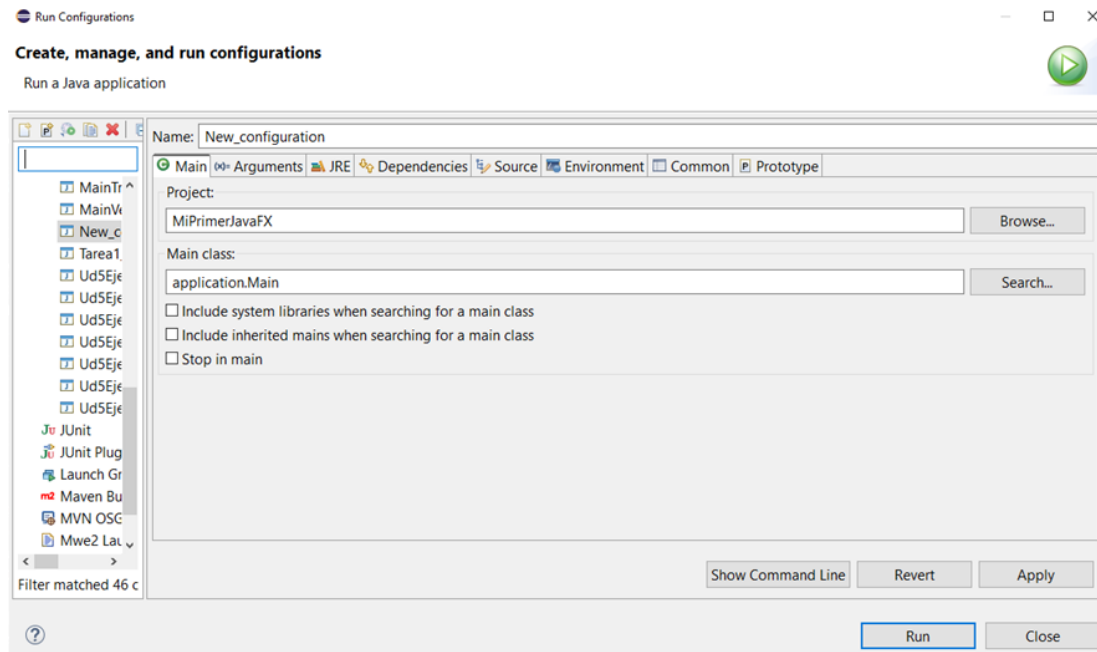


The screenshot shows the openjfx.io website. The sidebar on the left lists various topics, with 'JavaFX and Eclipse' selected. The main content area displays instructions for adding VM arguments to the Eclipse IDE. A code block shows the command: `--module-path "%path%\javalx-sdk-20\lib" --add-modules javafx.controls,javafx.fxml`. A warning box states: 'Warning: Make sure the checkbox "Use the -XstartOnFirstThread argument when launching with SWT" is not checked.' Below the text is a small image of the Eclipse IDE 'Run' configuration dialog.

Configurando Máquina virtual Java para JavaFX.

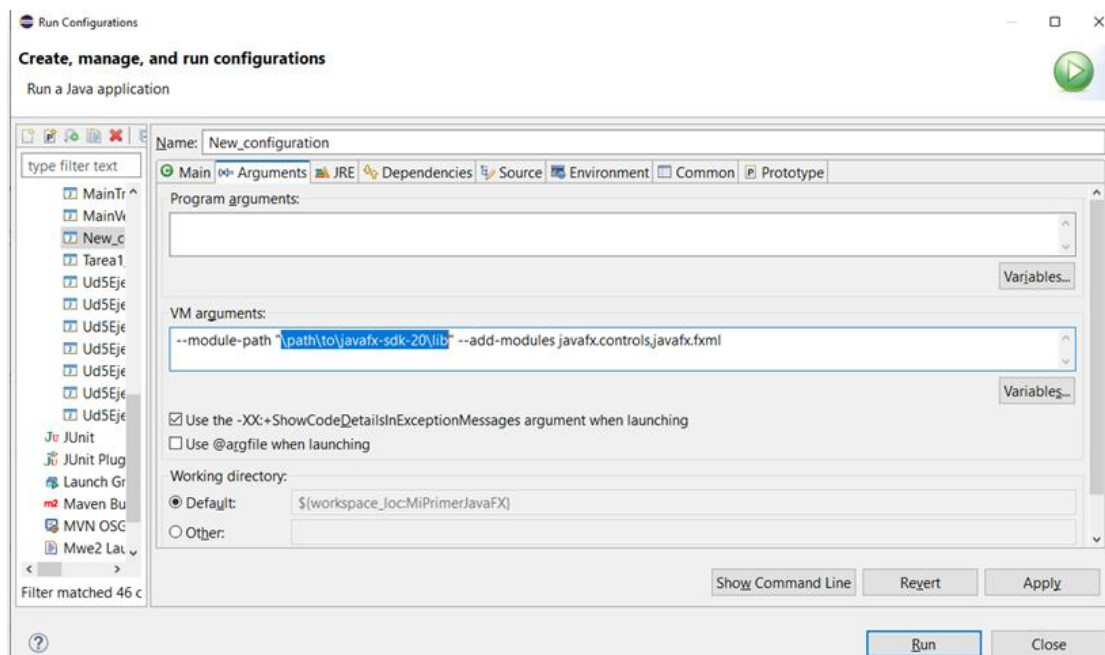
Fuente: Elaboración propia.

Volvemos a Eclipse, hacemos un clic derecho en nuestro proyecto seleccionamos Run As / Run Configurations, comprobamos el nombre de nuestro proyecto y que esté seleccionada la clase que queremos que sea la ejecutable, es decir, la clase que queremos que sea la clase principal (puede haber más de una clase con método main y ejecutable pero solo una será la que se ejecute cuando seleccionemos la opción Run) y en la pestaña Arguments en la sección VM arguments pegaremos la instrucción que copiamos anteriormente.



Run Configuration-Main para JavaFX.

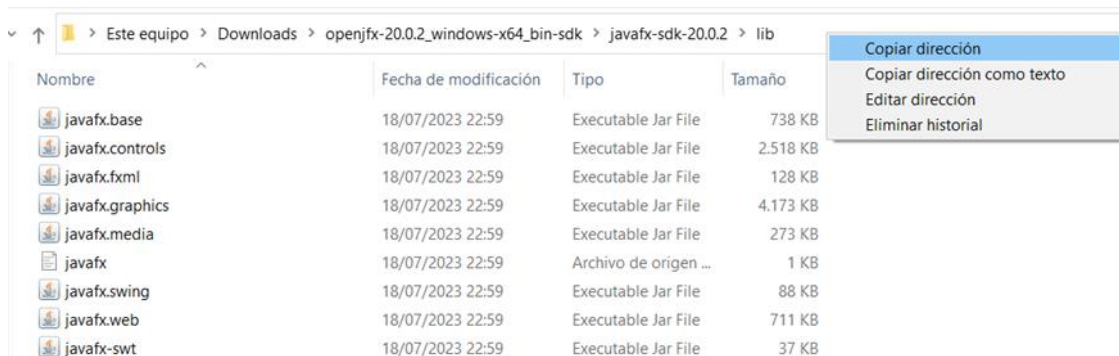
Fuente: Elaboración propia.



Run Configuration-Arguments para JavaFX.

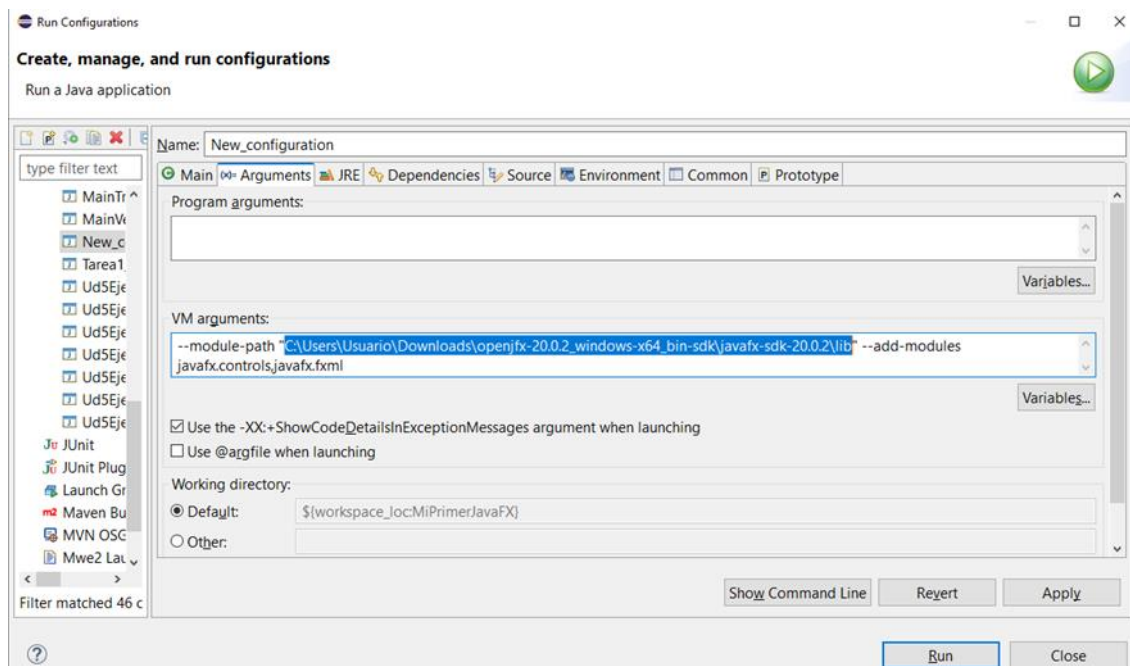
Fuente: Elaboración propia.

Aquí, debes cambiar el path (lo que aparece sombreado) por el tuyo, para ello localiza la carpeta que descargaste anteriormente y accede a la carpeta `lib`, a continuación, copia la dirección haciendo un clic derecho con el ratón sobre el camino de directorios que aparece en la barra superior, como muestra la imagen.



Run Configuration-cambiando path VM-para JavaFX.

Fuente: Elaboración propia.

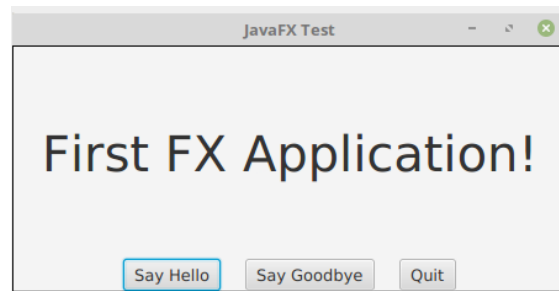


Run Configuration-nuevo path VM-para JavaFX.

Fuente: Elaboración propia.

A continuación, clic en **Apply** y **Run**

Si todo va bien debe aparecer una ventana (que ahora mismo no tiene nada en su interior), llamada JavaFX Application Window que es la interfaz principal que permite a los usuarios interactuar con la aplicación y ver su contenido gráfico.

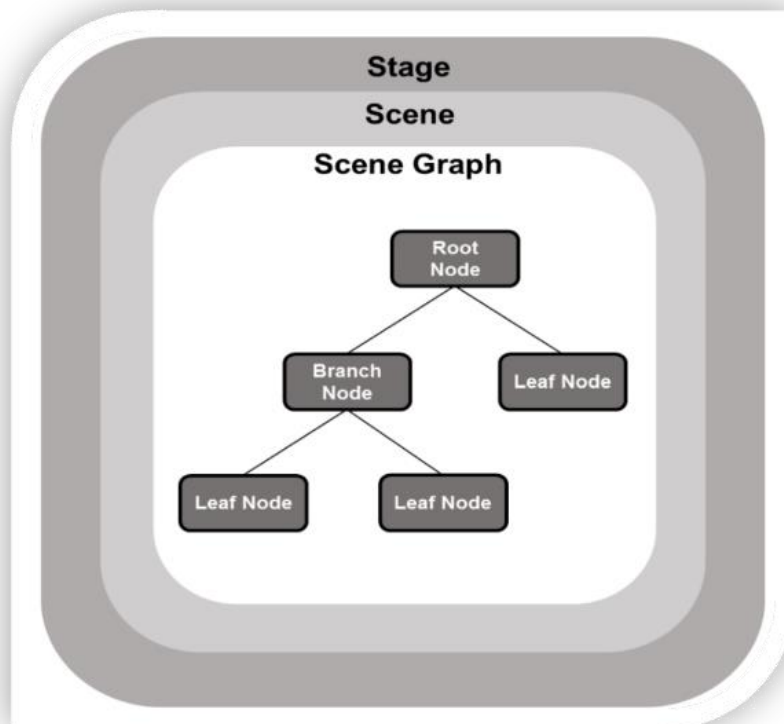


JavaFx Application Windows.

Fuente: <https://math.hws.edu/javanotes/c6/s1.html>

## 8.8 Componentes de una aplicación JavaFX

En general, una aplicación JavaFX tendrá tres componentes principales que son Stage, Scene y Nodes como se muestra en la imagen.



JavaFx Application Windows

Fuente: [https://www.tutorialspoint.com/javafx/javafx\\_tutorial.pdf](https://www.tutorialspoint.com/javafx/javafx_tutorial.pdf)



**Ventana principal:** El Stage representa la ventana principal de la aplicación JavaFX. Es la ventana que aparece cuando se ejecuta la aplicación y en la que se muestra la interfaz gráfica. Se pueden configurar diversas propiedades del Stage, como su título, dimensiones (ancho y alto), posición en la pantalla, visibilidad y comportamiento de cierre (por ejemplo, si se puede cerrar o minimizar). El Stage puede manejar eventos de ventana, como eventos de cierre (cuando el usuario cierra la ventana), eventos de cambio de tamaño, eventos de maximización y minimización, entre otros.

**Escena (Scene):** Cada Stage debe tener asociada una Scene, que representa el contenido visible dentro del Stage(nodos y elementos gráficos que se muestran en la ventana).

La escena es el contenedor principal que almacena todos los elementos gráficos de la interfaz. Representa el contenido visible de la ventana y actúa como un lienzo donde se colocan todos los elementos gráficos, como botones, etiquetas, campos de texto, etc. Una aplicación JavaFX puede tener una o varias escenas, pero solo una escena está activa y visible en un momento dado.

**Scene Graph:** Es una representación jerárquica de la estructura de la interfaz gráfica de usuario de una aplicación. En JavaFX, cada nodo gráfico (como botones, etiquetas, paneles, etc.) es representado por una clase específica que hereda de la clase base `javafx.scene.Node`. Estos nodos se organizan en una jerarquía de padres e hijos, donde cada nodo puede tener cero o más nodos hijos y un nodo puede tener solo un nodo padre.

**Concepto de evento:** Un evento se produce normalmente cuando un usuario interactúa con la aplicación y es una notificación o señal que se genera en respuesta a ese evento. Cuando ocurre un evento se desencadenan **controladores de evento (handlers)** que responden a estos eventos y realizan acciones específicas.

Estos eventos pueden estar relacionados con acciones, como hacer clic en un botón, mover el mouse sobre un elemento, presionar una tecla, cambiar el valor de un campo de entrada, etc. Algunos ejemplos de eventos comunes en JavaFX son:

- **ActionEvent:** Se genera cuando se realiza una acción, como hacer clic en un botón.
- **MouseEvent:** Se genera cuando ocurren acciones del mouse, como hacer clic, mover el mouse o soltar un botón.
- **KeyEvent:** Se genera cuando se presiona o suelta una tecla del teclado.

- **DragEvent:** Se genera durante una operación de arrastre y soltar (drag-and-drop).
- **WindowEvent:** Se genera cuando se realizan acciones en la ventana de la aplicación, como minimizar, maximizar o cerrar la ventana.
- **InputEvent:** Es la clase base para eventos de entrada, como MouseEvents y KeyEvents.



### ¿SABÍAS QUE...?

Como se ha comentado en la unidad hay otras alternativas para construir interfaces de usuario, una de las más conocidas y veterana es Swing, si quieres trabajar con ella, te puede servir saber que **Stage** es similar a **Jframe** y **Scene** similar a **Jpanel** en **Swing**.

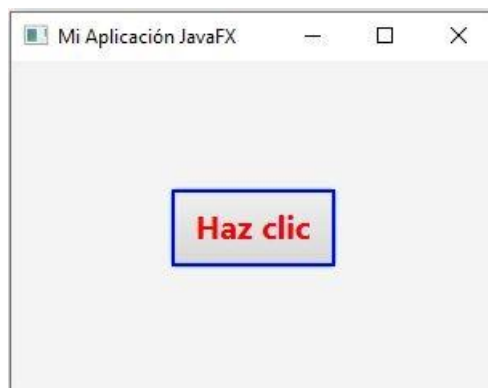
Para centrar un botón en una ventana gráfica con JavaFX puedes utilizar **StackPane** que es un contenedor que coloca sus elementos secundarios uno encima del otro, similar a una pila. **StackPane** es útil cuando necesitas superponer elementos gráficos y controlar su posición relativa fácilmente. **Pos.CENTER** es una constante de alineación que permite posicionar los elementos gráficos en diferentes lugares dentro del contenedor. (Ej: **Pos.TOP\_LEFT**, **Pos.TOP\_CENTER**, **Pos.CENTER\_LEFT**, etc)

```
package application;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.geometry.Pos;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
        try {
            // Crear un botón
            Button button = new Button("Haz clic");
            // Aplicar estilos CSS para el texto del botón
            button.setStyle("-fx-font-weight: bold;
```

```
        -fx-font-size: 20px;-fx-border-width: 2px;-  
        fx-border-color: blue; -fx-text-fill:  
        red;");  
    // Crear un StackPane para centrar el botón en la  
    ventana  
  
    StackPane stackPane = new StackPane();  
    stackPane.getChildren().add(button);  
    // Crear una escena y agregar el botón a la escena  
    Scene scene = new Scene(stackPane, 300, 200);  
    // Centrar el contenido del StackPane  
    StackPane.setAlignment(button, Pos.CENTER);  
    // Asignar la escena al escenario principal  
    primaryStage.setScene(scene);  
    // Establecer el título de la ventana  
    primaryStage.setTitle("Mi Aplicación JavaFX");  
  
    // Mostrar la ventana  
    primaryStage.show();  
} catch (Exception e) {  
    e.printStackTrace();  
}  
}  
  
public static void main(String[] args) {  
    launch(args);  
}  
}
```

Cuando se ejecute la aplicación aparecerá la siguiente pantalla:



Interfaz gráfica con un botón centrado y con estilo

Fuente: Elaboración propia.

A continuación, se explica brevemente el propósito algunas clases que se han importado:

**javafx.scene.control.Label:** La clase `Label` representa una etiqueta de texto que se utiliza para mostrar texto en la interfaz gráfica.

**javafx.scene.control.TextField:** La clase `TextField` representa una caja de texto de una sola línea que permite a los usuarios introducir texto en la interfaz gráfica.

**javafx.scene.layout.VBox:** La clase `VBox` es un tipo de contenedor que organiza los nodos hijos de manera vertical, es decir, uno debajo del otro. Es útil para alinear los elementos en una columna.



#### PARA SABER MÁS

Cambia el ancho, el alto, el padding, el margen, etc, de cada uno de los elementos que aparezcan en la interfaz gráfica que hayas construido.



#### ENLACE DE INTERÉS

Para ampliar la información sobre la API JavaFX, accede a este enlace:





### ENLACE DE INTERÉS

En este enlace puedes encontrar documentación y ejemplos sobre cada uno de los eventos y clases que se pueden utilizar con JavaFX:



Hay muchas posibilidades con JavaFX para desarrollar una interfaz de usuario. Como por ejemplo el uso de **FXML** y **Scene Builder**.

**FXML** es un formato de marcado basado en **XML** (Extensible Markup Language) y se utiliza para describir los elementos gráficos y su disposición en la interfaz de usuario de una aplicación JavaFX. En un archivo FXML, se pueden definir nodos y controles gráficos, como botones, etiquetas, campos de texto, tablas y otros elementos, junto con sus atributos, propiedades y eventos asociados. Además, se pueden definir disposiciones y tamaños, lo que permite un diseño más flexible y adaptativo. FXML es especialmente útil cuando se trabaja con herramientas visuales como **Scene Builder** que permite diseñar la interfaz gráfica arrastrando y soltando elementos como botones, etiquetas, campos de texto, tablas y otros controles en un lienzo de diseño.



### VÍDEO DE INTERÉS

Amplia información sobre el uso de FXML en el diseño de interfaces gráficas con JavaFX y otro IDE distinto a Eclipse:





### **NOTA DE INTERÉS**

Aunque en el temario de todas las unidades estamos utilizando Eclipse como IDE para el desarrollo en Java, como programadores, es fundamental conocer diferentes entornos de desarrollo y saber cómo configurarlos.

## RESUMEN FINAL

Cualquier programa que se desarrolle en Java y que tenga la necesidad de recibir o enviar datos lo hará a través de lo que se ha definido como un flujo (*stream*). La vinculación de un stream con un dispositivo físico concreto la va a realizar Java. Por lo tanto, las clases y los métodos que utilicemos van a ser las mismas sin tener en cuenta el dispositivo con el cual vamos a interactuar. Java se va a encargar de realizar esa tarea y será el que se comunique con el teclado, el monitor o cualquier otro dispositivo.

En esta unidad se ha comenzado analizando el concepto de flujo de entrada/salida que es fundamental en programación y se refiere a la transferencia de datos entre un programa y sus fuentes o destinos de datos.

Posteriormente se han clasificado los flujos dependiendo de si están orientados a bytes, a caracteres o a líneas. Cada uno está diseñado para manipular diferentes tipos de datos de manera adecuada.

En los **flujos de bytes** hemos usado `InputStream` y `OutputStream` para operar con secuencias de bytes (información binaria), muy útiles cuando se desea leer o escribir datos sin procesar, como imágenes, archivos binarios o datos que no tienen una codificación de caracteres específica.

En los **flujos de caracteres** se ha utilizado `Reader` y `Writer`. Estos flujos operan con datos que están codificados en caracteres (por ejemplo, texto) y son adecuados para leer y escribir información que tiene una representación legible por humanos. Manejan automáticamente la codificación y decodificación de caracteres, lo que facilita la lectura y escritura de texto en diferentes formatos.

En los **flujos de líneas**, las clases `BufferedReader` y `BufferedWriter` nos han servido para leer líneas completas de texto, en lugar de caracteres individuales, además el uso de un búfer mejora el rendimiento al leer o escribir grandes cantidades de datos en bloques.

Por otro lado, se ha visto el **manejo de ficheros** con la clase `File` para mostrar un listado de los ficheros y directorios de un determinado directorio y la clase `RandomAccessFile` que es una herramienta valiosa cuando necesitas acceder, leer y escribir datos en archivos de manera aleatoria y no secuencial. Es especialmente útil cuando trabajas con archivos que almacenan datos con estructuras complejas o cuando requieres realizar operaciones precisas en registros específicos.

Para finalizar, se han sentado las bases de la tecnología **JavaFX**, que es una tecnología de interfaz gráfica de usuario (GUI) en Java que permite a los desarrolladores crear aplicaciones con una interfaz visual moderna y atractiva. Es una biblioteca rica en características que proporciona una variedad de controles y componentes para diseñar interfaces interactivas entre los que cabe destacar **FXML** y **Scene Builder** para el desarrollo de interfaces de una forma más rápida.