Ficheros con JAVA

Juan Carlos Pérez Rodríguez

Sumario

Introducción

Cuando desarrollas programas, en la mayoría de ellos los usuarios pueden pedirle a la aplicación que realice cosas y pueda suministrarle datos con los que se quiere hacer algo. Una vez introducidos los datos y las órdenes, se espera que el programa manipule de alguna forma esos datos, para proporcionar una respuesta a lo solicitado.

Además, normalmente interesa que el programa guarde los datos que se le han introducido, de forma que si el programa termina, los datos no se pierdan y puedan ser recuperados en una sesión posterior. La forma más normal de hacer esto es mediante la utilización de ficheros, que se guardarán en un dispositivo de memoria no volátil (normalmente un disco).

Por tanto, sabemos que el almacenamiento en variables o vectores (arrays) es temporal, los datos se pierden en las variables cuando están fuera de su ámbito o cuando el programa termina. Las computadoras utilizan ficheros para guardar los datos, incluso después de que el programa termine su ejecución. Se suele denominar a los datos que se guardan en ficheros datos persistentes, porque existen, persisten más allá de la ejecución de la aplicación. Los ordenadores almacenan los ficheros en unidades de almacenamiento secundario como discos duros, discos ópticos, etc. En esta unidad veremos cómo hacer con Java estas operaciones de crear, actualizar y procesar ficheros.

A todas estas operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como Entrada/Salida (E/S).

Distinguimos dos tipos de E/S: la E/S estándar que se realiza con el terminal del usuario y la E/S a través de ficheros, en la que se trabaja con ficheros de disco.

Excepciones

Cuando se trabaja con archivos, es normal que pueda haber errores, por ejemplo: podríamos intentar leer un archivo que no existe, o podríamos intentar escribir en un archivo para el que no tenemos permisos de escritura. Para manejar todos estos errores debemos utilizar excepciones. Las dos excepciones más comunes al manejar archivos son:

FileNotFoundException: Si no se puede encontrar el archivo.

IOException: Si no se tienen permisos de lectura o escritura o si el archivo está dañado.

Stream

La clase Stream representa un flujo o corriente de datos, es decir, un conjunto secuencial de bytes, como puede ser un archivo, un dispositivo de entrada/salida (en adelante E/S), memoria, un conector TCP/IP (Protocolo de Control de Transmisión/Protocolo de Internet), etc.

Cualquier programa realizado en Java que necesite llevar a cabo una operación de entrada salida lo hará a través de un stream.

Un flujo es una abstracción de aquello que produzca o consuma información. Es una entidad lógica.

Las clases y métodos de E/S que necesitamos emplear son las mismas independientemente del dispositivo con el que estemos actuando, luego, el núcleo de Java, sabrá si tiene que tratar con el teclado, el monitor, un sistema de archivos o un socket de red liberando al programador de tener que saber con quién está interactuando.

La vinculación de un flujo al dispositivo físico la hace el sistema de entrada y salida de Java.

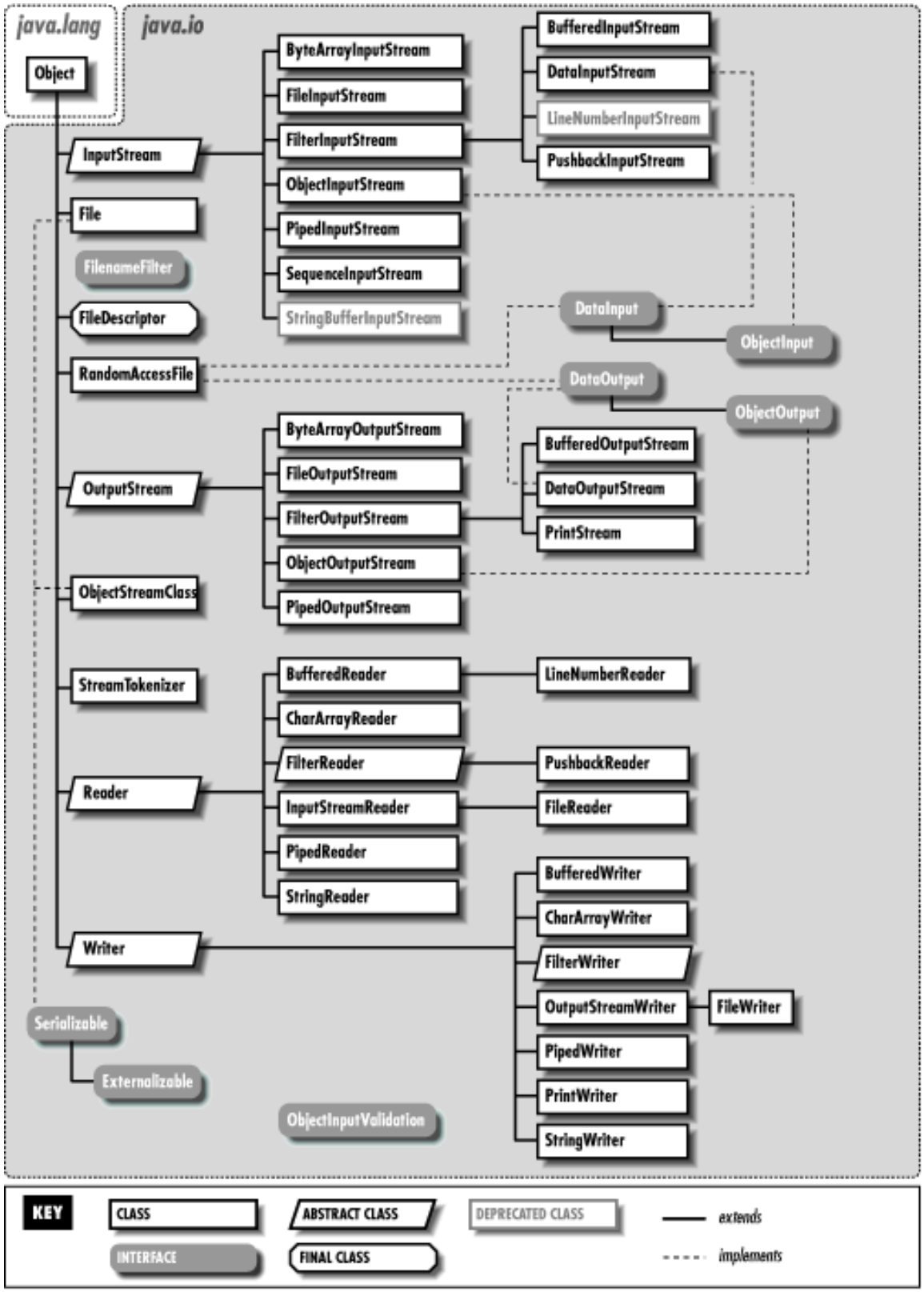
En resumen, será el flujo el que tenga que comunicarse con el sistema operativo concreto y "entendérselas" con él. De esta manera, no tenemos que cambiar absolutamente nada en nuestra aplicación, que va a ser independiente tanto de los dispositivos físicos de almacenamiento como del sistema operativo sobre el que se ejecuta. Esto es primordial en un lenguaje multiplataforma y tan altamente portable como Java.

Existen dos tipos de flujos, flujos de bytes (byte streams) y flujos de caracteres (character streams).

- **Los flujos de caracteres** (16 bits) se usan para manipular datos legibles por humanos (por ejemplo un fichero de texto). Vienen determinados por dos clases abstractas: **Reader y Writer.** Dichas clases manejan flujos de caracteres Unicode. De ellas derivan subclases concretas que implementan los métodos definidos destacados los métodos read() y write() que, en este caso, leen y escriben caracteres de datos respectivamente.

- **Los flujos de bytes (8 bits)** se usan para manipular datos binarios, legibles solo por la maquina (por ejemplo un fichero de programa). Su uso está orientado a la lectura y escritura de datos binarios. El tratamiento del flujo de bytes viene determinado por dos clases abstractas que son **InputStream y OutputStream**. Estas dos clases son las que definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan read() y write() que leen y escriben bytes de datos respectivamente.

Dentro de estas hay bastantes clases distintas para gestionar fichero la siguiente imagen lo ilustra



Según lo que queramos hacer usaremos unas u otras

El tipo más básico que tenemos en Java es **InputStream, OutputStream.** Ambas estaban pensadas para leer byte (así que son apropiadas para tratar con ficheros binarios) stas corrientes de bits, no representan ni textos ni objetos, sino datos binarios puros. Poseen numerosas subclases; de hecho casi todas las clases preparadas para la lectura y la escritura, derivan de estas.

Los métodos más importantes son **read** (leer) y **write** (escribir), que sirven para leer un byte del dispositivo de entrada o escribir un byte respectivamente.

Vamos a empezar trabajando con texto y luego veremos los binarios. Así pues las anteriores no parecen las más apropiadas

Stream de Texto

Lo primero que tenemos que tener en cuenta es que estamos hablando de Streams Un stream puede ser la consola (cuando hacemos: System.out) otro puede ser una conexión de red y otros pueden ser ficheros. Así que ahora estamos viendo algo más genérico que los ficheros de texto

Al principio cuando se trabajaba en ASCII con un byte (256bits) almacenábamos un carácter así que inputstream y outputstream se acomodaban a leer/escribir un carácter

Pero luego vino Unicode y hacía falta más espacio, InputStream y OutputStream no eran tan convenientes pues para almacenar un carácter unicode directamente. Surgieron **Reader/Writer** que nos sirven para leer los textos

Observar de nuevo el gráfico anterior Todas las clases que devien de Reader y las que deriven de Writer estarán pensadas para ficheros de texto

Pero todo en Java caminaba sobre las clases base InputStream y OutputStream así que hubo que adaptar estas nuevas clases a las anteriores. Así surgieron: **InputStreamReader** y **OutputStreamWriter**

Bien, veamos la siguiente sentencia:

```
OutputStreamWriter osw = new OutputStreamWriter(System.out);
```

Con ella estamos tomando la forma de escribir en la salida estandar

(System.out es un OutputStream que como dijimos antes era el tipo básico en Java para escritura)

Con el siguiente código se imprimiría en la salida estandar (consola) la frase hola mundo:

```
        OutputStreamWriter osw = new OutputStreamWriter(System.out);
        try {
            osw.write("Hola Mundo!".toCharArray());
            osw.close();
        } catch (IOException ex) {

        }
```

Está dentro de try-catch porque puede OutputStreamWriter contempla las excepciones de entrada/salida (IOException)

Como vemos cerramos el flujo después de haber terminado (os.close())

Y la lectura nos quedaría:

```
        InputStreamReader isr = new InputStreamReader(System.in);
        char c = (char)isr.read();
```

La alternativa para leer algo más que un único carácter:

```
InputStreamReader isr = new InputStreamReader(System.in);
char charArray[] = new char[1024];
isr.read(charArray);
isr.close();
System.out.println(new String(charArray));
```

Ahora bien, el teclado ya de por sí va sobre un buffer y nos vendría bien disponer de una clase que tomara toda una línea de texto. Así pues surge **BufferedReader**

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String texto = br.readLine();
```

La anterior es la forma más fiable de leer un texto de teclado

Vale , entonces hasta ahora hemos visto:

InputStream

- InputStreamReader
  - BufferedReader

OutputStream

- OutputStreamWriter
  - BufferedWriter

Lo que hemos visto vale para textos puros. Pero, ¿ y si queremos leer del stream un número que nos haya escrito el usuario por teclado ? Para eso tendremos que usar los parse o usar Scanner

Por último estamos acostumbrados a darle un formato a lo que queremos imprimir. BufferedWriter no tiene utilidades de formato pero tenemos una clase que está a más alto nivel que sí lo soporta: **PrintWriter**

```
        OutputStreamWriter osw = new OutputStreamWriter(System.out);
        BufferedWriter br = new BufferedWriter(osw);
        PrintWriter pw = new PrintWriter(br);
        pw.format("el valor de PI: %.3f ",3.1415926);
        pw.close();
```

Adicionalmente PrintWriter nos permite hacer uso de nuestros conocidos: println(), print()

Ficheros de Texto

Mediante las clases **FileInputStream y FileOutputStream** se puede obtener un InputStream o un OutputStream y a estos aplicarles todo lo anterior que hemos hablado para ficheros de texto

Así lo anterior que imprimimos en pantalla podríamos guardar en fichero mediante:

```
FileOutputStream fos = new FileOutputStream("lico.txt");
OutputStreamWriter osw = new OutputStreamWriter(fos);
BufferedWriter bw = new BufferedWriter(osw);
PrintWriter pw = new PrintWriter(bw);
pw.format("el valor de PI: %.3f ",3.1415926);
pw.close();
```

Pero si se quiere se puede condensar dos de las anteriores en una Ya que se creó la clase **FileWriter** (y **FileReader** como equivalente para leer) que nos haría el efecto de las clases anteriores: FileOutputStream y OutputStreamWriter

El siguiente código escribe lo puesto por el usuario hasta encontrar un intro sin texto

```
File f=new File("salida.txt");
try {
	FileWriter fw=new FileWriter(f);
	BufferedReader br=new BufferedReader(new
	InputStreamReader(System.in));
	String texto=" ";
	while(texto.length()>0){
		texto=br.readLine();
		fw.write(texto+"\r\n");
	}
	fw.close();
}
catch (IOException e) {
	e.printStackTrace();
}
```

Archivos binarios

Para archivos binarios se pueden utilizar las clases **DataInputStream** y **DataOutputStream**. Estas clases están mucho más preparadas para escribir datos de todo tipo

El proceso sería:

(1) Crear un objeto FileOutputStream a partir de un objeto File que posea la ruta al archivo que se desea escribir (para añadir usar el segundo parámetro del constructor indicando true)

(2) Crear un objeto DataOutputStream asociado al objeto anterior. Esto serealiza mediante el constructor de DataOutputStream.

(3) Usar el objeto del punto 2 para escribir los datos mediante los métodos write Tipo donde tipo es el tipo de datos a escribir (int, double, ...). A este método se le pasa como único argumento los datos a escribir.

(4) Se cierra el archivo

Ejemplo:

```
File f=new File("prueba.out"); Random r=new Random();
double d=18.76353;
try{
	FileOutputStream fis=new FileOutputStream(f);
	DataOutputStream dos=new DataOutputStream(fis);
	for (int i=0;i<234;i++){ //Se repite 233 veces
		dos.writeDouble(r.nextDouble());//No aleatorio
	}
	dos.close();
}
catch(FileNotFoundException e){
	System.out.println("No se encontro el archivo");
}
catch(IOException e){
	System.out.println("Error al escribir");
}
```

Y ahora la lectura:

```
boolean finArchivo=false;//Para provocar bucle infinito
try{
```

```
      FileInputStream fis=new FileInputStream(f);
      DataInputStream dis=new DataInputStream(fis);
      double d;
      while (!finArchivo){
            d=dis.readDouble();
            System.out.println(d);
      }
      dis.close();
}
catch(EOFException e){
      finArchivo=true;
}
catch(FileNotFoundException e){
      System.out.println("No se encontra el archivo");
}
catch(IOException e){
      System.out.println("Error al leer");
}
```

Observar que estamos capturando una excepción que detecta el final del fichero y así salimos del bucle

Archivos de acceso aleatorio

Hasta ahora los archivos se están leyendo secuencialmente. Es decir desde el inicio hasta el final. Pero es posible leer datos de una zona concreta del archivo.

RandomAccessFile permite eso mismo, ya que permite leer archivos en forma aleatoria. Es decir, se permite leer cualquier posición del archivo en cualquier momento. Los archivos anteriores son llamados secuenciales, se leen desde el primer byte hasta el último.

Ejemplo

```
RandomAccessFile rafFichero = new RandomAccessFile("cuba.txt", "rwd");
if( rafFichero.length() > 1 )
     rafFichero.seek(rafFichero.length());
rafFichero.writeUTF("Estamos añadiendo texto al final del fichero");
```

En el ejemplo anterior se ve el uso de seek() que es el método que diferencia especialmente al poder ubicarnos en una posición en concreto del fichero y leer o escribir a partir de esa posición

Serialización

Es una forma automática de guardar y cargar el estado de un objeto. Se basa en la interfaz Serializable (en el paquete java.io) que es la que permite esta operación. Si una clase implementa esta interfaz puede ser guardado y restaurado directamente en un archivo

Cuando se desea utilizar un objeto para ser almacenado con esta técnica, debe ser incluida la instrucción implements **Serializable** la cabecera de clase. Esta interfaz no posee métodos, pero es un requisito obligatorio para hacer que un objeto sea serializable.

La clase ObjectInputStream y la clase ObjectOutputStream se encargan de realizar este procesos. Son las encargadas de escribir o leer el objeto de un archivo. Son herederas de InputStream y OutputStream, de hecho son casi iguales a DataInput/OutputStream sólo que incorporan los métodos readObject y writeObject que son los que permiten grabar directamente objetos.

Ejemplo:

```
        Persona persona = new Persona("yepo",43,180,new DNI(12345678));
        try(
             FileOutputStream fos = new FileOutputStream(new File("aquel.txt"));
             BufferedOutputStream bos = new BufferedOutputStream(fos);
             ObjectOutputStream oos = new ObjectOutputStream(bos);
           ){

            oos.writeObject(persona);

        } catch (FileNotFoundException ex) {
            Logger.getLogger(FicherosBinariosObjetos.class.getName()).log(Level.
SEVERE, null, ex);
        } catch (IOException ex) {
            Logger.getLogger(FicherosBinariosObjetos.class.getName()).log(Level.
SEVERE, null, ex);
        }
```

Y la lectura:

```
        ArrayList<Persona> personas = new ArrayList<Persona>();
        try(
               FileInputStream fis = new FileInputStream(new File("aquel.txt"));
               BufferedInputStream bis = new BufferedInputStream(fis);
               ObjectInputStream ois = new ObjectInputStream(bis);
           ){
            boolean finDeFichero = false;
```

```
            Persona p;
            do{
                try{
                    p = (Persona)ois.readObject();
                    personas.add(p);
                    System.out.println(p);
                }catch(EOFException ex){
                    finDeFichero = true;
                }
            }while(!finDeFichero);
        } catch (FileNotFoundException ex) {}
```

Como se puede ver es muy cómodo porque nos despreocupamos bastante. Simplemente escribimos el objeto y luego lo leemos

En el anterior ejemplo hemos hecho uso de un try que no hemos usado hasta ahora:

Try with resources

Java 7 permite declarar recursos en el try. Un recurso es un objeto que necesita ser cerrado. Try-with resources cerrará de forma automática los recursos que implementen Closeable al terminar el try, tanto si va bien como si da algún error. Con este try ya no es necesario el uso de finally.

```
// Con Java 7
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}


// Antes de Java 7

static String readFirstLineFromFileWithFinallyBlock(String path)
                                                    throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

XML

La serialización de objetos resulta interesante si bien tiene como problemática que cuando fue creado se pensó para transmitir los objetos por líneas de comunicación con bastantes fallos. Al crearse se hizo que fuera tolerante a fallos pero para eso introdujo mucha redundancia. Adicionalmente tiene la dificultad de ser algo que únicamente podemos realizar desde Java. Si guardamos un fichero en XML tenemos la ventaja de que se puede trabajar desde diferentes lenguajes de programación

JAXB (no confundir con la interfaz de acceso JAXP) es una librería de (un)-marshalling. El concepto de serialización es el proceso de almacenar un conjunto de objetos en un fichero.

Unmarshalling es justo el proceso contrario: convertir en objetos el contenido de un fichero. Para el caso concreto de XML y Java, unmarshalling es convertir el contenido de un archivo XML en una estructura de objetos Java.

De manera resumida, JAXB convierte el contenido de un documento XML en una estructura de clases Java:

- El documento XML debe tener un esquema XML asociado (fichero.xsd), por tanto el contenido del XML debe ser válido con respecto a ese esquema.

- Una vez JAXB crea en tiempo de diseño (no durante la ejecución) la estructura de clases, el proceso de unmarshalling (creación de objetos de las clases creadas con el contenido del XML) y marshalling (almacenaje de los objetos como un documento XML) es sencillo y rápido, y se puede hacer en tiempo de ejecución.

JAXB hace use de anotaciones para saber como proceder con cada componente de una clase

Así pues veamos algunas de las más relevantes:

```
@XmlRootElement(name="casa")
public class Casa {}
```

De la forma anterior JAXB sabe que vamos a grabar y leer la clase casa como elemento de nuestro xml

```
@XmlElement(name="direccion")
String direccion;
```

Con xmlelement le decimos que tiene que incorporar en la transformación el elemento definido y tendrá el nombre puesto en name

```
@XmlRootElement(name="propietario")
public class Propietario {

    @XmlElementWrapper
    @XmlElement(name="casa")
    private ArrayList<Casa> casas;
```

Al poner xmlelementwrapper le decimos que el atributo casas es una colección del objeto xmleelment casa. Pondremos estas etiquetas encima de la definición de nuestra colección de datos (en este caso arraylist) y debemos crear un método get para casas: getCasas()

También podemos decidir si un atributo de la clase se convierte en un elemento en XML o en un atributo. Para ello en lugar de @XmlElement con @XmlAttribute

Necesitaremos **un constructor por defecto** (si no hemos puesto uno sin parámetros y tenemos alguno creado con parámetros tendremos que agregar uno sin parámetros) para poder hacer el proceso

El siguiente es un ejemplo:

```java
@XmlRootElement(name="propietario")
public class Propietario {
    @XmlElementWrapper
    @XmlElement(name="casa")
    private ArrayList<Casa> casas;

    public ArrayList<Casa> getCasas() {
        return casas;
    }

    @XmlElement(name="nombre")
    String nombre;

    @XmlElement(name="apellidos")
    String apellidos;

    @XmlElement(name="nacimiento")
    GregorianCalendar nacimiento;

    //la librería xml precisa de un constructor sin parámetros
    public Propietario(){}

       public Propietario(String nombre, String apellidos, GregorianCalendar
nacimiento, Casa casa) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.nacimiento = nacimiento;
        this.casas = new ArrayList<Casa>();
        casas.add(casa);
    }
}
```

Ahora Casa:

```
@XmlRootElement(name="casa")
public class Casa {

    @XmlElement(name="direccion")
    String direccion;

    @XmlElement(name="metros")
    int metros;

    @XmlElement(name="planta")
    int planta;

    @XmlElement(name="ascensor")
    boolean ascensor;

    @XmlElement(name="precio")
    double precio;

    public Casa(){} //necesitamos un constructor por defecto para marshall

      public Casa(String direccion, int metros, int planta, boolean ascensor,
double precio) {
        this.direccion = direccion;
        this.metros = metros;
        this.planta = planta;
        this.ascensor = ascensor;
        this.precio = precio;
    }
}
```

Los siguientes 2 métodos nos permiten pasar de un objeto Almacen a una String formateada XML y viceversa:

```
  static Almacen stringXMLtoAlmacen(String string) {
        JAXBContext contexto;
        Marshaller marshaller;
        StringReader sr = new StringReader(string);
        Almacen almacen = null;
        try {
            contexto =  JAXBContext.newInstance(Almacen.class);
            marshaller  = contexto.createMarshaller();
                        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
Boolean.TRUE);
            Unmarshaller unmarshaller = contexto.createUnmarshaller();
            almacen = (Almacen) unmarshaller.unmarshal(sr);
        } catch (JAXBException ex) {
            System.out.println(ex);
        }finally{
            return almacen;
        }
    }


    static String almacenToStringXML(Almacen almacen) {
        JAXBContext contexto;
        Marshaller marshaller;
        OutputStream os=null;
        StringWriter sw = new StringWriter();
        try {
            contexto =  JAXBContext.newInstance(almacen.getClass());
            marshaller  = contexto.createMarshaller();
         marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
            marshaller.marshal(almacen, sw);
        } catch (JAXBException ex) {
            System.out.println(ex);
        }finally{
            return sw.toString();
        }
    }
```

Paths y Files

Ambas clases surgen a partir de Java 7 y en conjunto podrían sustituir y mejorar lo que aporta la clase File

La clase File representa el nombre de un fichero o de un directorio que existe en el sistema de ficheros. Los métodos de File permiten obtener toda la información sobre las características del fichero o directorio. Ya en algún ejemplo del tema lo hemos utilizado

Por medio de Paths podemos eliminar uno de los problemas cuando estamos con los sistemas de ficheros de los diferentes sistemas operativos. Por ejemplo si escribimos:

```
Path path = Paths.get("proyectos", "pro", "almacenamiento");
System.out.println(path.toString());
System.out.println(path.toAbsolutePath().toString());
```

Observaremos que nos da un texto con slash: "/" o backslash: "\" según lo correspondiente al sistema operativo y nos encadena los textos. Así en linux el AbsolutePath nos queda:
/home/user/proyectos/pro/almacenamiento

Algo que suele ser interesante es tomar el path en la carpeta de usuario:

```
Path path = Paths.get(System.getProperty("user.home"), "prueba.txt");
```

Para poder mantener la compatibilidad con lo anterior se puede convertir en File:

```
path.toFile()
```

La clase java.nio.file.Files (Files) es el otro punto de entrada a la librería de ficheros de Java. Es la que nos permite manejar ficheros reales del disco desde Java.

Esta clase tiene métodos estáticos para el manejo de ficheros , lo que permite crear y borrar ficheros y directorios, comprobar su existencia en el disco, comprobar sus permisos, moverlos de un directorio a otro y lo más importante, leer y escribir el contenido de dichos ficheros

```
        Path path = Paths.get("./ficheroPreguntar.txt");

        System.out.println("path = " + path);
        System.out.println ("   exists = " + Files.exists (path));
        System.out.println ("   readable  = " + Files.isReadable(path));
        System.out.println ("   writeable = " + Files.isWritable(path));
        System.out.println ("   executeable = " + Files.isExecutable(path));
```

Files.newBufferedReader

Un buffer es una estructura de datos que permite el acceso por trozos a una colección de datos.

Los buffers son útiles para evitar almacenar en memoria grandes cantidades de datos, en lugar de ello, se va pidiendo al buffer porciones pequeñas de los datos que se van procesando por separado.

También resultan muy útiles para que la aplicación pueda ignorar los detalles concretos de eficiencia de hardware subyacente, la aplicación puede escribir al buffer cuando quiera, que ya se encargará el buffer de escribir a disco siguiendo los ritmos más adecuados y eficientes.

El siguiente código nos lee mediante Buffer línea a línea un fichero (APROPIADO PARA FICHEROS GRANDES)

```
        Path path = Paths.get("fichero.txt");

        try (BufferedReader reader = Files.newBufferedReader(path) ){
            String  line;
            while ( (line = reader.readLine ()) != null )
                System.out.println (line);

        } catch (IOException e) {
            System.err.println ("ERROR: " + e);
        }
```

Files.readAllLines

Si nuestro fichero es relativamente pequeño (cuidado! Cargar un fichero de 300MB puede desbordar máquinas de 4GB de RAM) una opción bastante interesante es cargar todas las filas de texto en un array en memoria:

```
        ArrayList<String> lineas = null;
        Path path = Paths.get(System.getProperty("user.home"), "prueba.txt");
        try {
            lineas = new ArrayList(Files.readAllLines(path));
        } catch (IOException ex) {
            System.out.println(ex);
        }
```

Files.copy

Con Files.copy podemos copiar muy fácilmente un fichero o un directorio (el directorio copia la estructura no el contenido) Lo hace de una forma muy eficiente y nos despreocupamos de la apertura, cierre y demás circunstancias de manejo de ficheros

```
Files.copy(originalPath, copied, StandardCopyOption.REPLACE_EXISTING);
```

Files.newBufferedWriter

Si vamos a escribir en ficheros largos o si queremos aislarnos de la lentitud de hardware podemos usar un buffer de escritura

El siguiente ejemplo ilustra la escritura en fichero del texto puesto por el usuario en teclado hasta escribir la palabra: “fin” (adicionalmente se ve el uso de printwriter envolviendo el buffer)

```
        Path path = Paths.get("fichero.txt");
        boolean finalizar = false;
        //para ficheros de texto escritura
        //Acceso eficiente mediante buffer
        System.out.println("Introducir el texto a escribir");
        System.out.println("Se finalizará introduciendo una línea: Fin");

        try(BufferedWriter bw =  Files.newBufferedWriter(path,
                StandardOpenOption.APPEND,
                StandardOpenOption.CREATE
                )){
            Scanner sc = new Scanner(System.in);
            PrintWriter pw = new PrintWriter(bw);
            do{
                String texto = sc.nextLine();
                if(texto.toLowerCase().equals("fin")){
                    finalizar = true;
                }else{
                    bw.write(texto);
```

```
                    bw.newLine();

                    pw.printf("%s \n", texto);
                }
            }while(!finalizar);

        } catch (IOException ex) {
            System.out.println(ex);
        }
```

El ejemplo anterior nos muestra que podemos abrir el fichero para agregar, crear, etc

Finalmente decir que Files permite el uso de Stream mediante lines: Files.lines() y Files.newBufferedReader.lines()

En el siguiente ejemplo pasamos a minúscula un stream desde buffer:

```
String fileName = "lines.txt";
ArrayList<String> list = null;
try (Stream<String> str= Files.newBufferedReader(Paths.get(fileName)).lines()) {
      list = (ArrayList)str
                .map(String::toLowerCase)
                .collect(Collectors.toList());
} catch (IOException e) {
      e.printStackTrace();
}
```

Java FX FileChooser

Para el manejo de ficheros desde una aplicación gráfica hay un elemento específico que nos permite seleccionar (y así obtener el path) el lugar en el sistema de ficheros para la ubicación del fichero.

Ubicándonos en el evento (por ejemplo de un botón open o save)

```
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Abrir fichero");
        File selectedFile = fileChooser.showOpenDialog(null);
        Path pathFile=null;
        if (selectedFile != null) {
            pathFile = selectedFile.toPath();
            System.out.println("File selected: " + pathFile.toString());
        }
```

Tableview

Para mostrar datos en java fx puede ser útil un control llamado tableview

Es importante que la clase que vamos a mostrar en la tabla de datos tenga SimpleProperty para que pueda vincular un atributo con su columna en la tabla, además debe ser pública la clase y tener los getter y setter correspondientes

Ejemplo de una clase que vamos a usar para “inyectar” sus atributos en un tableview

```
public class EAfx{
    SimpleIntegerProperty mes;
    SimpleDoubleProperty intereses;

    public int getMes() {
        return mes.get();
    }

    public void setMes(int mes) {
        this.mes.set(mes);
    }

    public double getIntereses() {
```

```
        return intereses.get();
    }

    public void setIntereses(double intereses) {
        this.intereses.set(intereses);
    }
}
```

Supongamos ahora que la tableview tiene dos columnas:

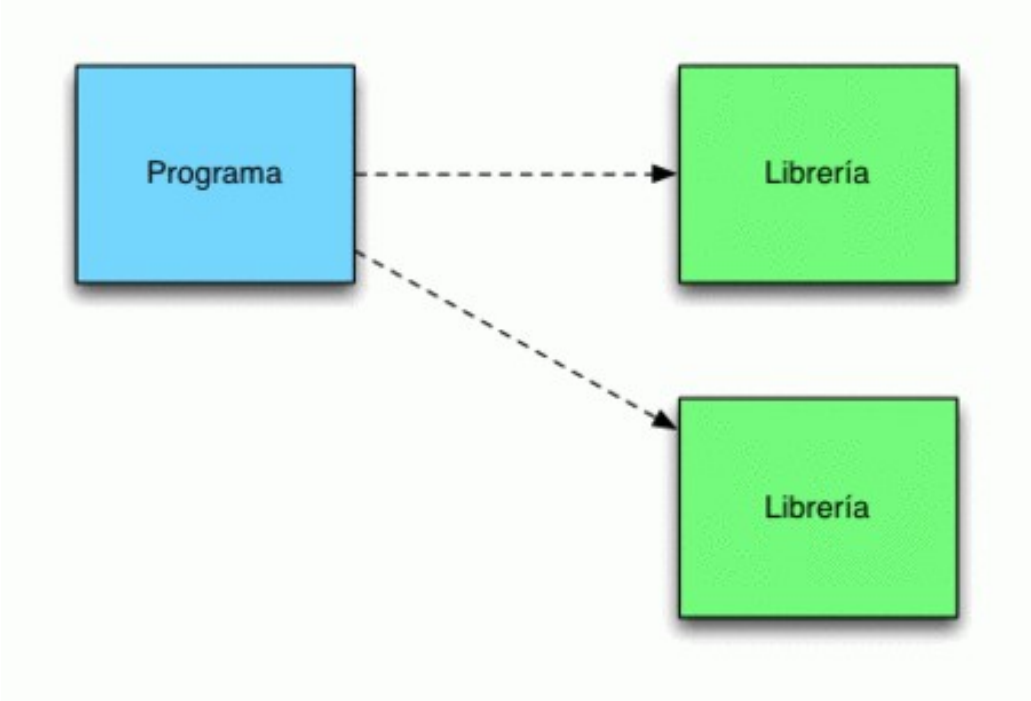
```
    @FXML
    private TableColumn<EAfx, String> mes;
    @FXML
    private TableColumn<EAfx, String> intereses;
```

y ahora para inyectar los datos de un array de objetos EAfx en el tableview:

```
mes.setCellValueFactory(new  PropertyValueFactory<EAfx,String>("mes"));
intereses.setCellValueFactory(new  PropertyValueFactory<EAfx, String>("intereses"));
tableView.setItems(FXCollections.observableArrayList(arrayfx));
```

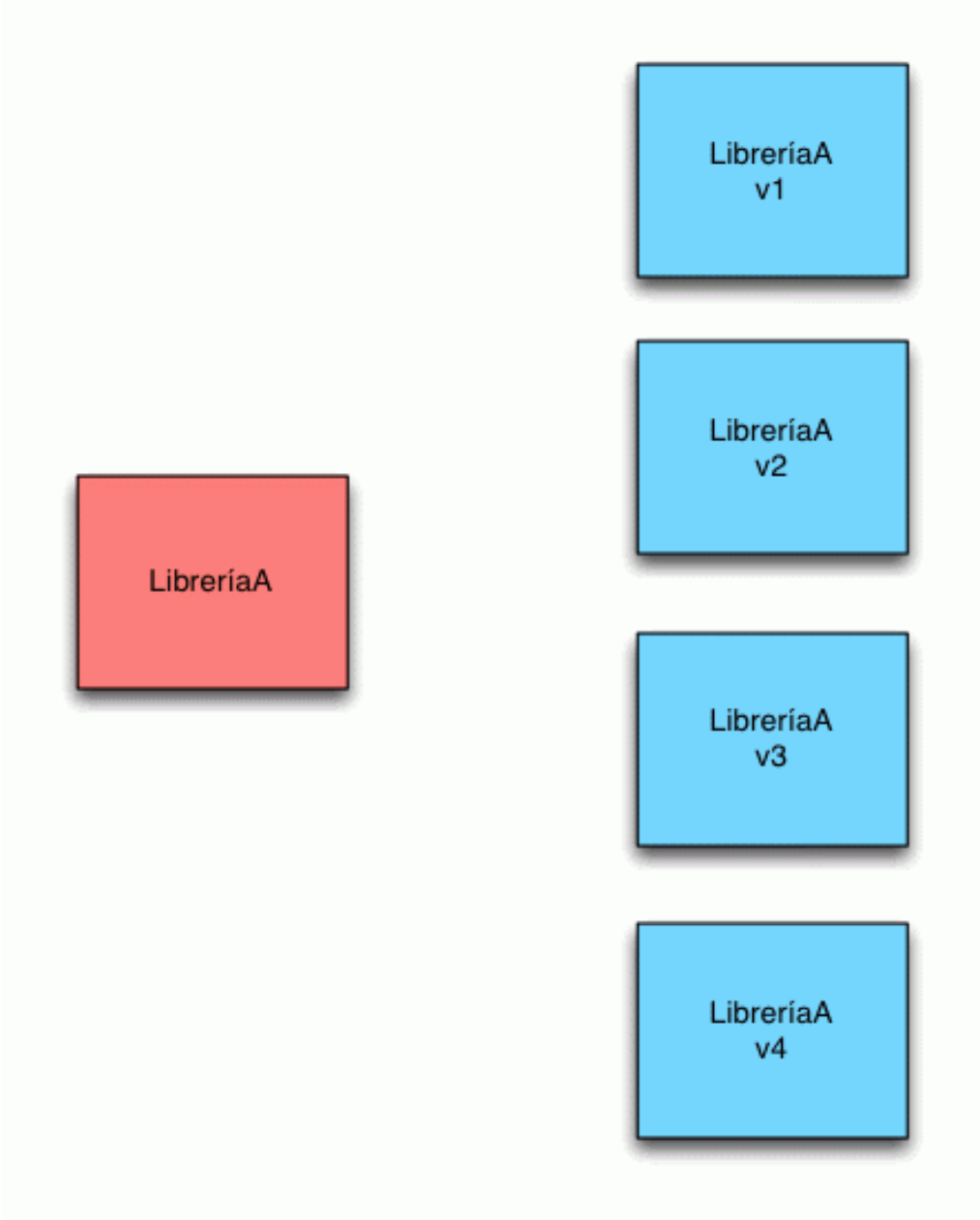
Maven

Normalmente cuando nosotros trabajamos con Java/JavaEE el uso de librerías es algo común como en cualquier otro lenguaje de programación.

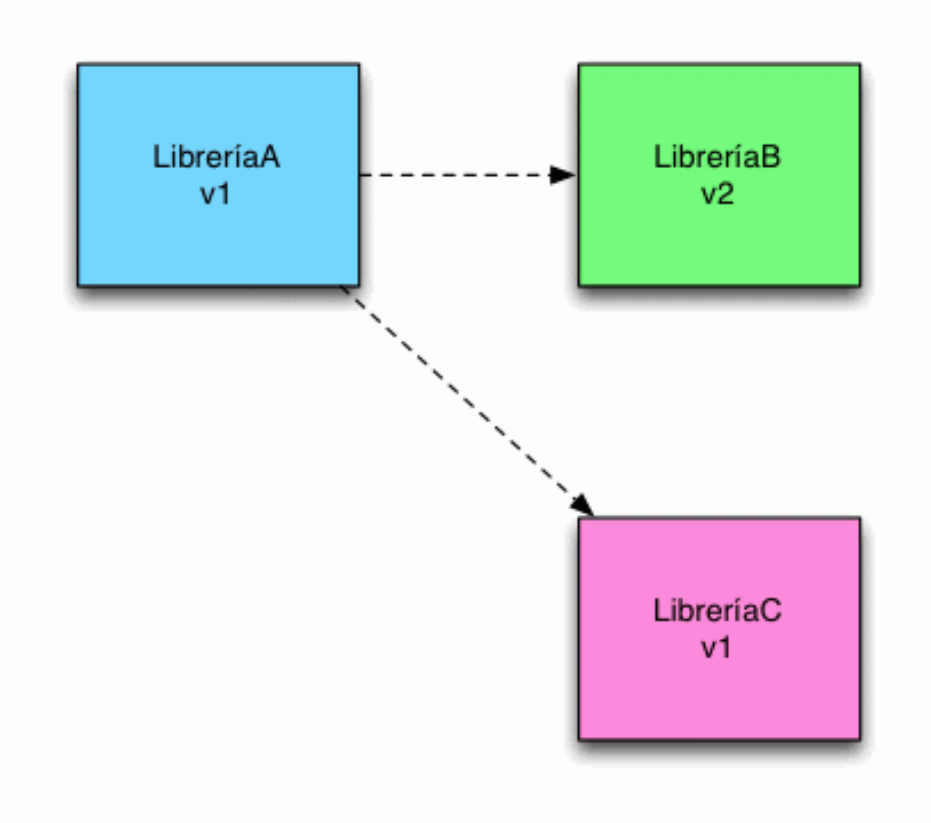


Librerías y limitaciones

El concepto de librería es un concepto que a veces es limitado. Por ejemplo nosotros podemos querer utilizar la librería A en nuestro proyecto. Sin embargo no nos valdrá con simplemente querer utilizar la librería sino que ademas necesitaremos saber que versión exacta de ella necesitamos.



¿Es esto suficiente?. Lamentablemente no lo es, una librería puede depender de otras librerías para funcionar de forma correcta. Así pues necesitamos más información para gestionarlo todo de forma correcta.



Maven y Artefactos

Si vamos a una herramienta más evolucionada, llegamos a maven. Maven, con comandos simples, nos crea una estructura de directorios para nuestro proyecto con sitio para los fuentes, los iconos, ficheros de configuración y datos, etc, etc. Si a maven le indicamos qué jar externos necesitamos, es capaz de ir a buscarlos a internet y descargarlos por nosotros. Sin necesidad prácticamente de configurar nada, maven sabe como borrar los .class, compilar, generar el jar, generar el javadoc y generar un documentación web con montones de informes (métricas, código duplicado, etc). Maven se encarga de pasar automáticamente nuestros test de prueba cuando compilamos. Incluso maven nos genera un zip de distribución en el que van todos los jar necesarios y ficheros de configuración de nuestro proyecto.

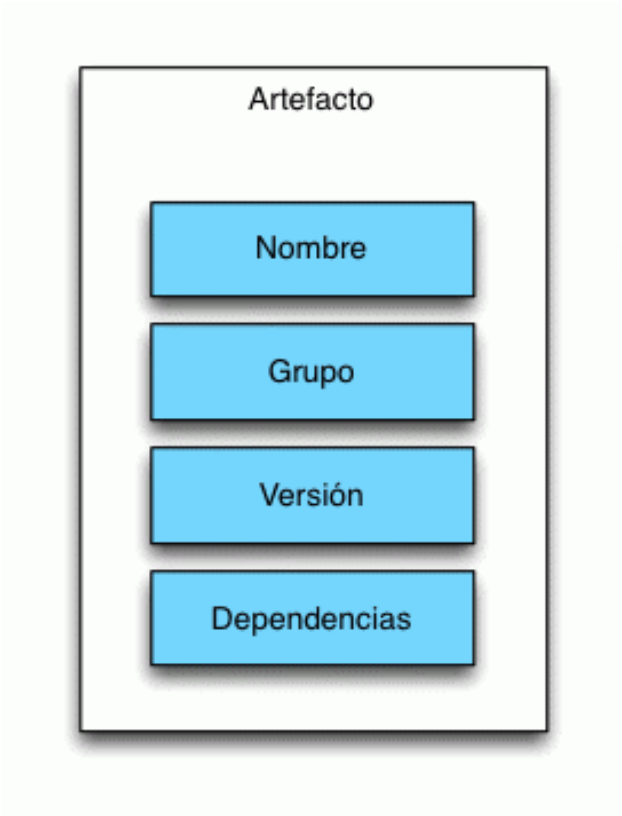
Una de las grandes ventajas de **maven** son los repositorios (almacenes) de ficheros jar que se crea.

Si miras en http://www.ibiblio.org/maven2/ tienes el repositorio oficial de jars de maven. Ahí están los **groupId** de casi todos los jar de libre distribución que puedas encontrar en internet. Tienes el log4j, commons-logging, JFreeChart, mysql-connector, etc, etc. **Maven** es capaz de bajarse cualquiera de estos jar si tu proyecto lo necesita.

Todo lo que se baje **maven** de internet lo mete en un repositorio (almacen) local en tu pc, de forma que si lo necesita una segunda vez, no necesita descargárselo nuevamente de internet. Este directorio, habitualmente está en

- **$HOME/.m2** en unix/linux
- **C:\Documents and Settings\usuario\.m2** en windows

Maven solventa esta problema a traves **del concepto de Artefacto. Un Artefacto puede verse como una librería con esteroides** (aunque agrupa mas conceptos). Contiene las clases propias de la librería pero ademas **incluye toda la información necesaria para su correcta gestión (grupo, versión, dependencias etc).**



Artefactos y POM

Para definir un **Artefacto** necesitamos crear un **fichero POM.xml (Proyect Object Model)** que es el encargado de almacenar toda la información que hemos comentado anteriormente:

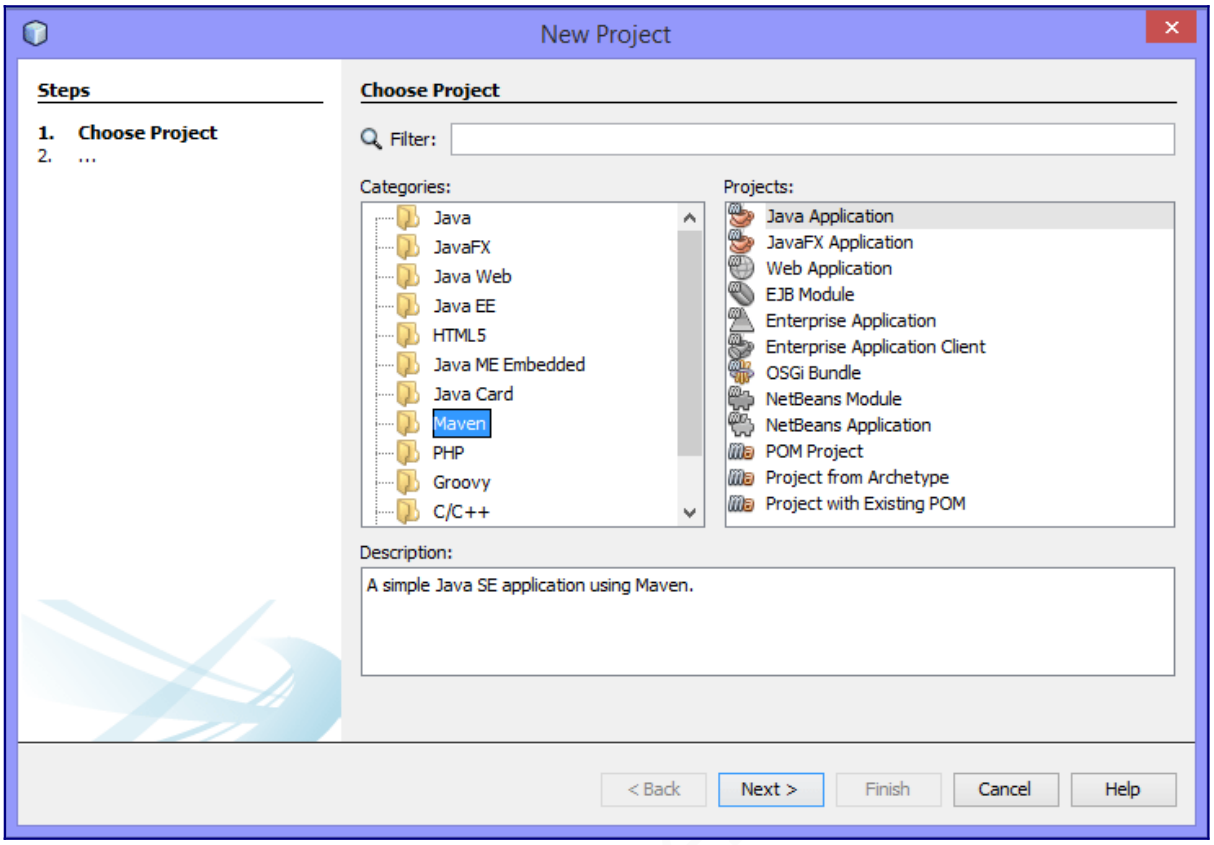
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance"
```

```
                            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>
 <groupId>com.genbetadev.proyecto1</groupId>
 <artifactId>proyecto1</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 <packaging>jar</packaging>
 <dependencies>

 <dependency>
 <groupId>log4j</groupId>
 <artifactId>log4j</artifactId>
 <version>1.2.17</version>
 </dependency>

</dependencies>
</project>
```

La estructura del fichero puede llegar a ser muy compleja y puede llegar a depender de otros POM. En este ejemplo **estamos viendo el fichero más sencillo posible.** En el se define el nombre del Artefacto (artifactID) el tipo de empaquetado (jar) y también las dependencias que tiene (log4j). **De esta manera nuestra librería queda definida de una forma mucho más clara.**

Creación de un proyecto Maven

Lo primero que haremos será ir a Netbens y crear un nuevo proyecto. Debemos tener cuidado de escoger Maven/Java Application a la hora de escoger el tipo de proyecto.



Llamaré mi proyecto **MavenTest**.

Ej. agregar las dependencias de JFreeCharts. Podemos buscar los repositorios que necesitamos en la siguiente dirección:

http://mvnrepository.com/

En el caso de JFreeCharts podemos encontar las dependencias en la siguiente dirección:

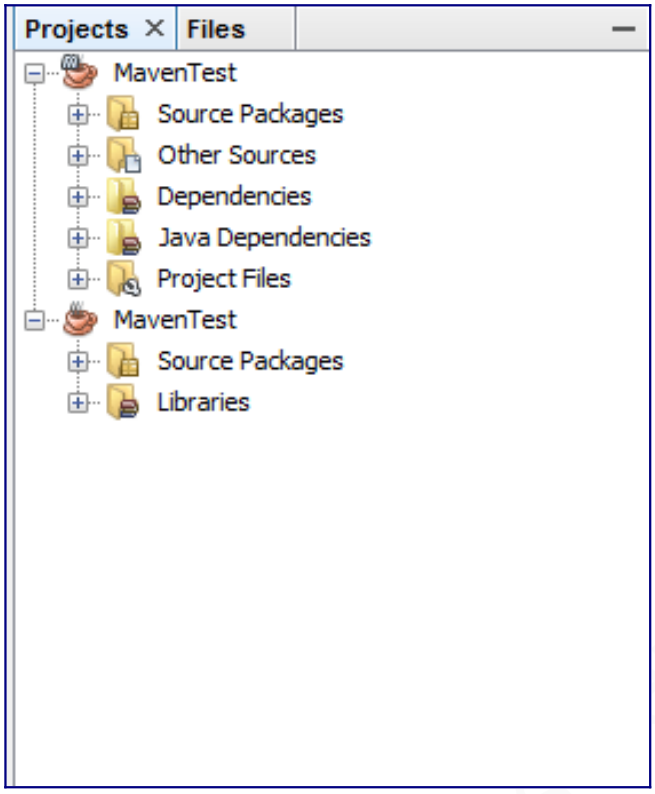
http://mvnrepository.com/artifact/org.jfree/jfreechart

Ej para la versión 1.0.19 . La dependencia es la siguiente:

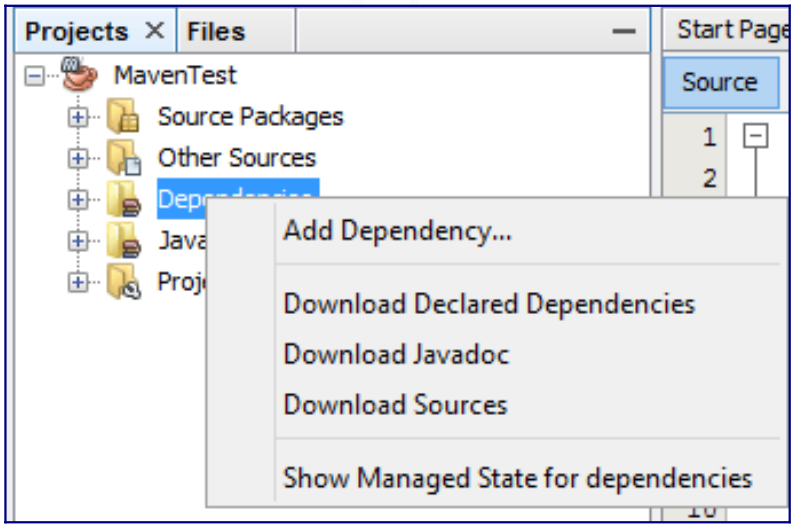
```
<dependency>
    <groupId>org.jfree</groupId>
    <artifactId>jfreechart</artifactId>
    <version>1.0.19</version>
</dependency>
```

Existen dos formas de agregar esto a nuestro proyecto, ya sea a través de la interfaz gráfica de usuario o colocando esta dependencia directamente en el archivo **pom.xml**

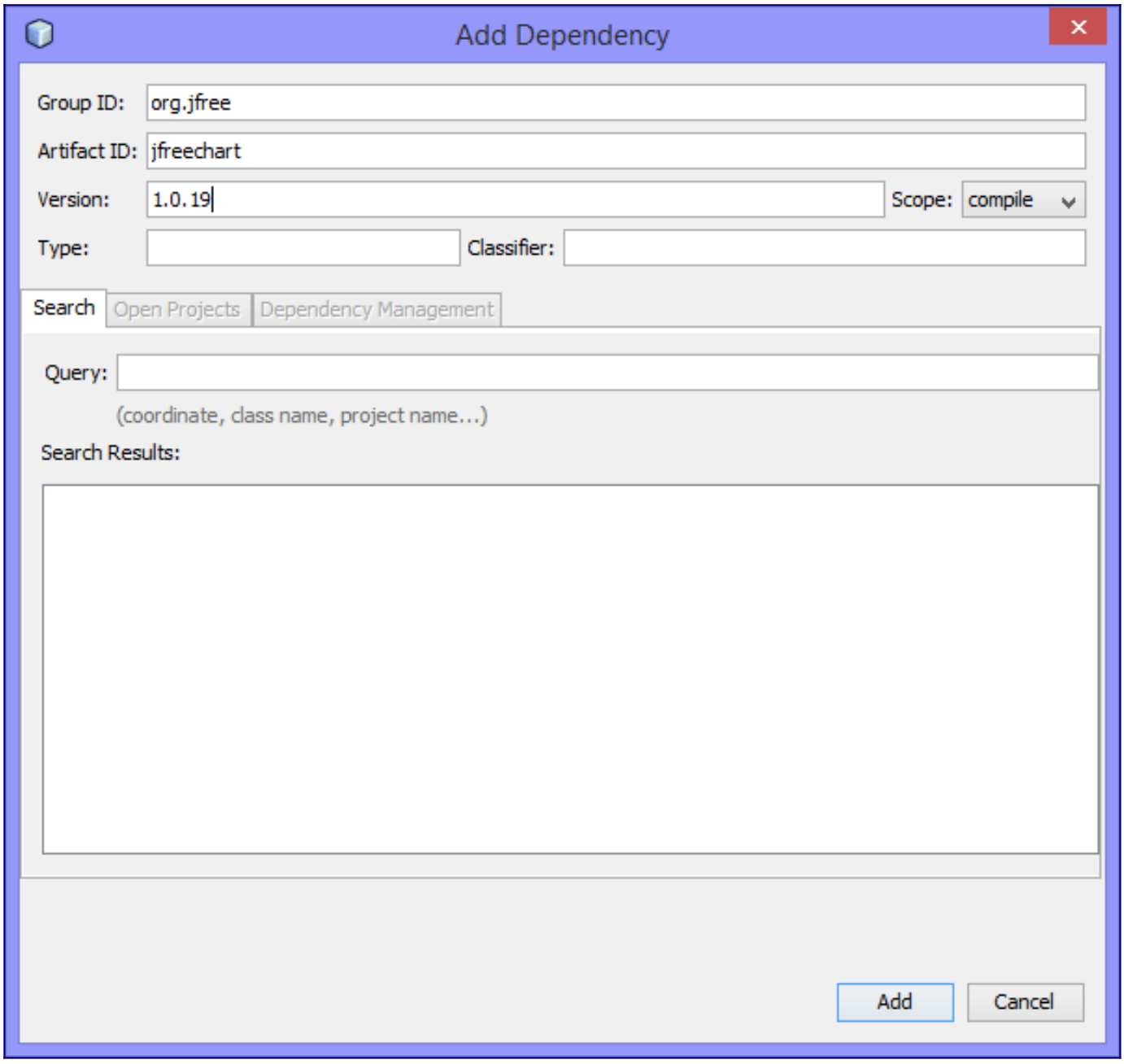
Lo haremos a través de la interfaz gráfica de usuario. El árbol de nuestro proyecto es ligeramente diferente al de una aplicación común de Java.



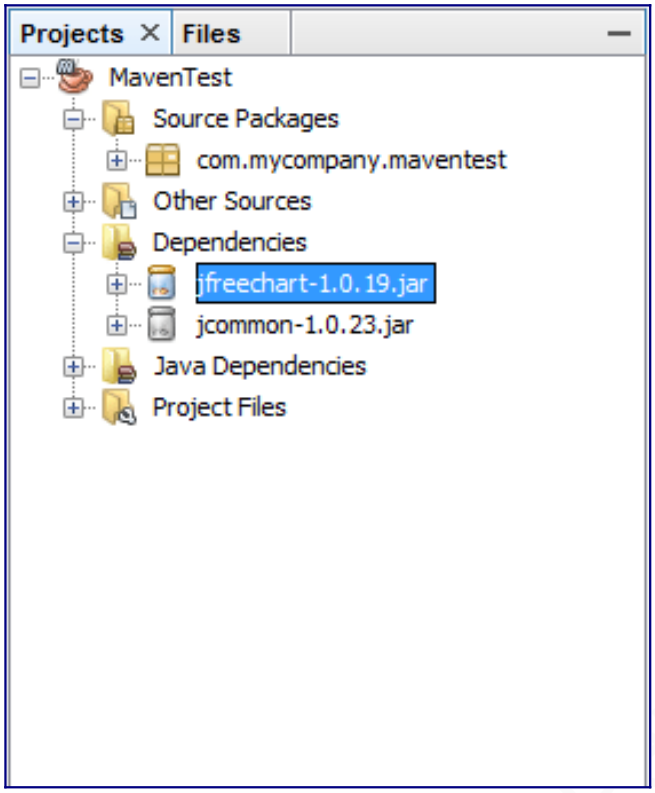
Vemos que en el caso del proyecto Maven tenemos las carpetas Java Dependencies, Dependencies y Project Files. Vamos a darle clic derecho a la carpeta Dependencies y escogeremos la opción Add Dependency.



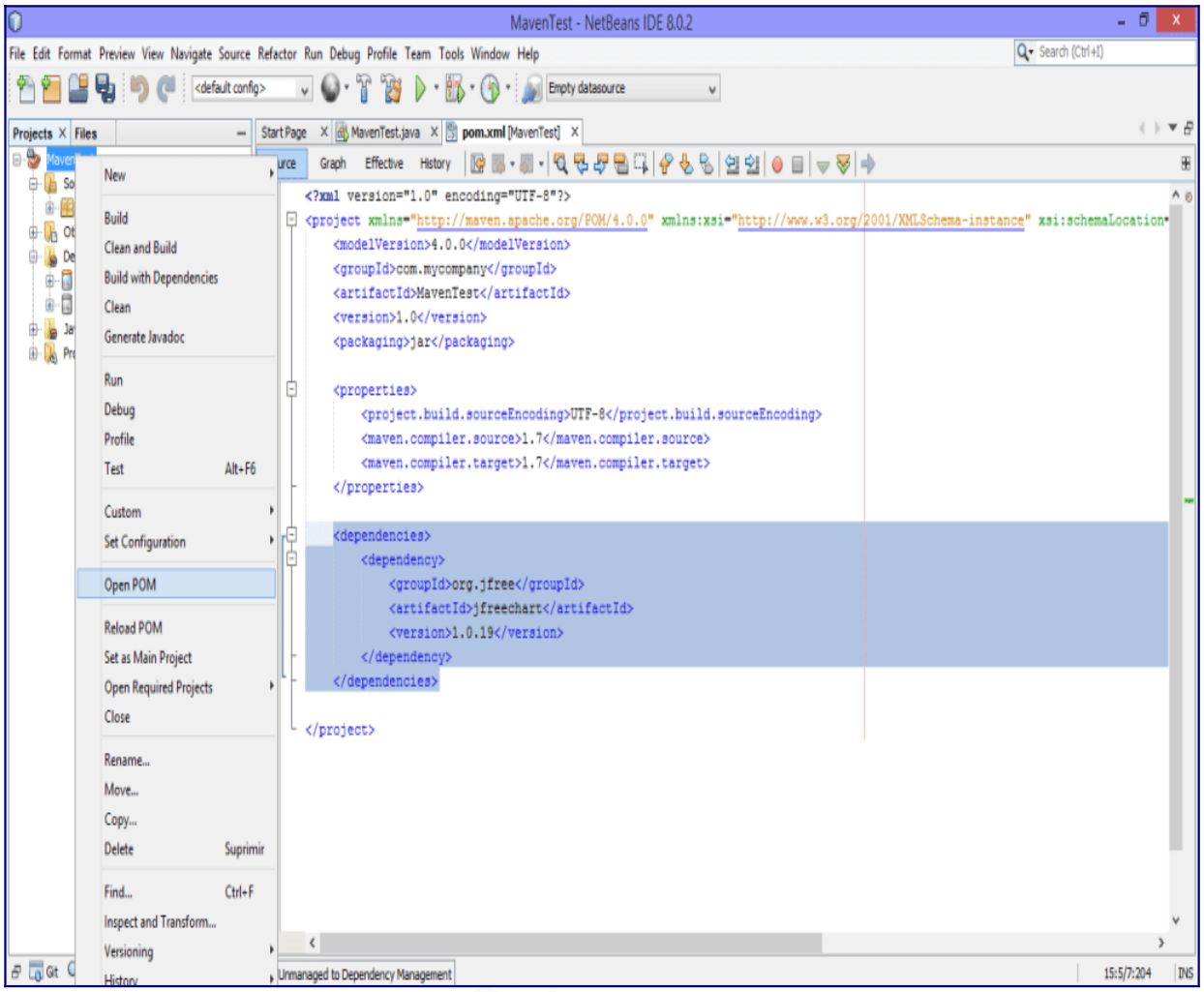
Nos aparece una ventana donde colocaremos la información de JfreeCharts.



Al hacer clic en “Add” se agregará a nuestro proyecto todo lo que necesitamos para utilizar JFreeCharts. Al final del proceso de descarga tendremos nuestro proyecto con sus respectivas dependencias ya instaladas y listas para utilizar.



Si verificamos el archivo pom.xml (clic derecho sobre el proyecto, Open POM) veremos lo siguiente:



Si fuésemos a agregar las dependencias de forma manual basta con copiar el código de dependencias en este archivo y listo. Existen cientos de librerías en los repositorios de Maven. Es posible también utilizar GitHub para hostear repositorios y llamarlos desde Maven. Esta herramienta es sumamente útil y nos facilitará muchísimo la creación de proyectos en Java.

Así pues para trabajar con Maven

File → New Project → Maven → JavaFX Application

Para compilar con lambdas tenemos que poner en pom.xml que estamos en Java 8 (1.8) (Tu Proyecto → Project Files->pom.xml)

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <compilerArguments>
            <bootclasspath>${sun.boot.class.path}${path.separator}${java.home}/lib/jfxrt.jar</
bootclasspath>
        </compilerArguments>
    </configuration>
</plugin>
```


Dependencias para itext.version (para que las descargue Maven)

```
<dependencies>
<dependency>
   <groupId>com.itextpdf</groupId>
   <artifactId>itextpdf</artifactId>
   <version>5.5.10</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.itextpdf.tool/xmlworker -->
<dependency>
```

```
    <groupId>com.itextpdf.tool</groupId>
    <artifactId>xmlworker</artifactId>
    <version>5.5.10</version>
</dependency>
</dependencies>
```


¿ y cómo haríamos si queremos tener un único jar sin usar Maven ?

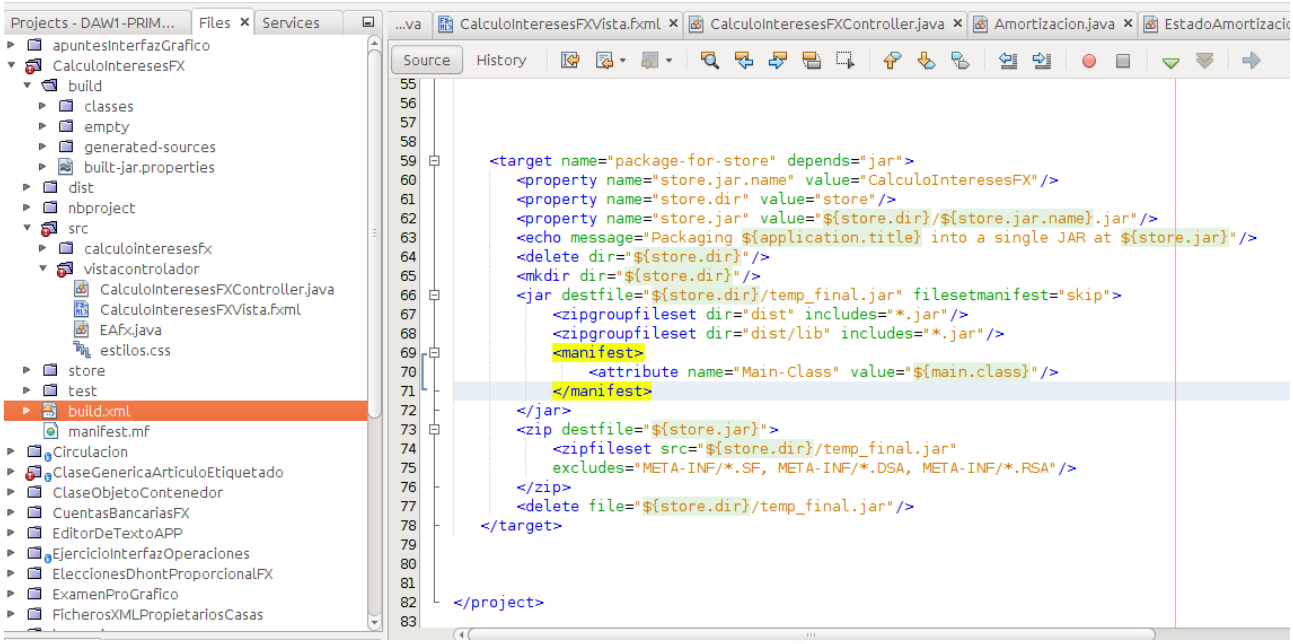
Usaríamos Ant:

primero incorporamos las librerías

libraries → Add jar folder

en la ventana que se abre se pone la ruta donde tenemos el jar externo

Luego en la pestaña “Files” buscamos build.xml:



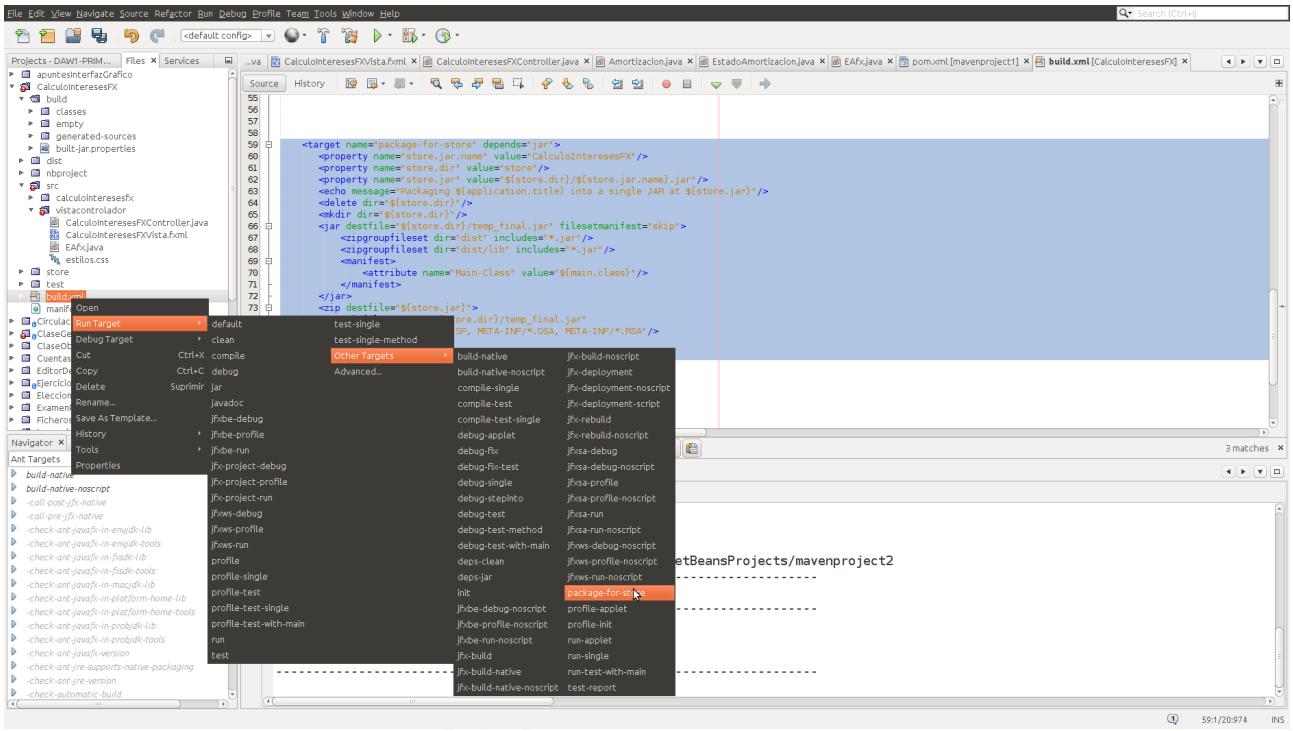
y allí le agregamos (justo antes del final del fichero antes de que cierre: </project> Observar la imagen de encima)

```
<target name="package-for-store" depends="jar">
  <property name="store.jar.name" value="CalculoInteresesFX"/>
  <property name="store.dir" value="store"/>
  <property name="store.jar" value="${store.dir}/${store.jar.name}.jar"/>
  <echo message="Packaging ${application.title} into a single JAR at ${store.jar}"/>
  <delete dir="${store.dir}"/>
  <mkdir dir="${store.dir}"/>
  <jar destfile="${store.dir}/temp_final.jar" filesetmanifest="skip">
    <zipgroupfileset dir="dist" includes="*.jar"/>
    <zipgroupfileset dir="dist/lib" includes="*.jar"/>
    <manifest>
      <attribute name="Main-Class" value="${main.class}"/>
    </manifest>
  </jar>
  <zip destfile="${store.jar}">
    <zipfileset src="${store.dir}/temp_final.jar"
    excludes="META-INF/*.SF, META-INF/*.DSA, META-INF/*.RSA"/>
  </zip>
  <delete file="${store.dir}/temp_final.jar"/>
</target>
```

Debemos fijarnos que hay que cambiar lo que hemos marcado en Amarillo en el texto anterior por el nombre del proyecto

El siguiente paso es:

botón derecho sobre build.xml → Run target → Other targets → package for store



Finalmente pulsamos en el icono de clean and build que estamos acostumbrados ya en todo proyecto (Mayuscula + F11)

nos genera una carpeta store (userhome/NetbeansProjects/nombredelproyecto/store) que contiene el jar de la aplicación con las librerías

Incluyendo librerías externas para crear un PDF desde Java

Usando Maven con las dependencias para itexpdf podemos “traernos” los jar desde internet a nuestro proyecto. Para usarlas veamos un ejemplo:

```
Document document = new Document();
PdfWriter pdfWriter;
try {

    pdfWriter = PdfWriter.getInstance(document, new FileOutputStream(fichero));
      document.open();
      String k = "<html><body><h1>Hello world!</h1></body></html>";
      System.out.println(k);
//            HTMLWorker hw = new HTMLWorker(document);
//            hw.parse(new StringReader(k));


     XMLWorkerHelper.getInstance().parseXHtml(pdfWriter, document, new StringReader(k));

      document.close();

      System.out.println("PDF Created!");


 } catch (FileNotFoundException ex) {
      ex.printStackTrace();
 } catch (DocumentException ex) {
      ex.printStackTrace();
 } catch (IOException ex) {
      ex.printStackTrace();
 }
```

Con lo anterior generamos un PDF con el formato que tenga el String HTML

Otra alternativa es que el Sistema tenga una impresora PDF entonces tenemos una mejor opción en JavaFX para imprimir cualquier nodo (cualquier elemento gráfico como una tableview, un chart, etc) en un PDF (o cualquier otra impresora del sistema)

```
    public  void printPDFfx(){

        PrinterJob job = PrinterJob.createPrinterJob();
        job.setPrinter(Printer.getAllPrinters()
                              .stream()
                              .filter(p->p.getName().equals("PDF"))
                              .findFirst()
                              .get());
        if( job != null){
            boolean ok = job.printPage(tableView);
            if(ok){
                job.endJob();
            }
        }
    }
```

Observar que en el código anterior se ha buscado una impresora del sistema con el nombre: “PDF” habría que poner ahí el nombre apropiado