IES Virgen del Carmen

JUNTA DE ANDALUCIA
CONSEJERÍA DE EDUCACIÓN

Departamento de Informática

Acceso a Datos: Gestión de ficheros y directorios

Juan Gualberto Gutiérrez Marín

Septiembre 2023

UNIÓN EUROPEA
Fondo Social Europeo
El FSE invierte en tu futuro

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

INFORMÁTICA - IES Virgen del Carmen - JAÉN

Índice

1 Tema 01: Gestión de ficheros y directorios con Java

- Ciclo formativo de 2º de Desarrollo de Aplicaciones Multiplataforma.
- Curso 2022-23.
- Esta documentación está disponible también en PDF en este enlace.

1.1 Resultados de aprendizaje y criterios de evaluación del tema.

En este tema trabajamos el siguiente RA:

RA 1. Desarrolla aplicaciones que gestionan información almacenada en ficheros identificando el campo de aplicación de los mismos y utilizando clases específicas. Criterios de evaluación:

- Se han utilizado clases para la gestión de ficheros y directorios.
- Se han valorado las ventajas y los inconvenientes de las distintas formas de acceso.
- Se han utilizado clases para recuperar información almacenada en ficheros.
- Se han utilizado clases para almacenar información en ficheros.
- Se han utilizado clases para realizar conversiones entre diferentes formatos de ficheros.
- Se han previsto y gestionado las excepciones.
- Se han probado y documentado las aplicaciones desarrolladas.

1.2 Contenidos del tema:

Manejo de ficheros:

- Clases asociadas a las operaciones de gestión de ficheros y directorios: creación, borrado, copia, movimiento, recorrido, entre otras.
- Formas de acceso a un fichero. Ventajas.
- Clases para gestión de flujos de datos desde/hacia ficheros. Flujos de bytes y de caracteres.
- Operaciones sobre ficheros secuenciales y aleatorios.
- Serialización/deserialización de objetos.
- Trabajo con ficheros: de intercambio de datos (XML y JSON, entre otros). Analizadores sintácticos (parser) y vinculación (binding). Conversión entre diferentes formatos.
- Excepciones: detección y tratamiento.
- Desarrollo de aplicaciones que utilizan ficheros.

1.3 Clases asociadas a las operaciones de gestión de ficheros y directorios: creación, borrado, copia, movimiento, recorrido

La entrada y salida en Java sigue el mismo modelo que en Unix (basada en flujos). Así, la E/S en Java se basa en clases como `FileInputStream`, `FileOutputStream`, `BufferedReader`, `BufferedWriter`, `Scanner`, entre otras.

Java siempre está evolucionando y mejorando, y en cada versión se van introducido características nuevas o bibliotecas adicionales para simplificar la E/S de archivos. Por lo tanto, siempre es recomendable verificar la documentación oficial de la versión específica de Java que estés utilizando para obtener información sobre las últimas características y mejores prácticas en cuanto a E/S de archivos.

Nosotros en clase haremos lo siguiente:

1. Utilizaremos las clases del paquete `java.nio.file`: A partir de Java 7, se introdujo el paquete `java.nio.file` que proporciona clases como `Path`, `Files`, y `FileSystems` que ofrecen una API más moderna y versátil para trabajar con archivos y directorios.

2. Usaremos try-with-resources: Para garantizar que los recursos se cierren adecuadamente después de su uso, utiliza la declaración `try-with-resources` al trabajar con objetos que implementan la interfaz `AutoCloseable`. Esto garantiza que los flujos de E/S se cierren automáticamente al salir del bloque `try`, lo que hace que el código sea más limpio y seguro.

```
try (BufferedReader reader = new BufferedReader(new FileReader("archivo.txt"))) {
    // Operaciones de lectura aquí
} catch (IOException e) {
    // Manejo de excepciones
}
```

3. Utilizamos las clases `Buffered`: Al realizar operaciones de E/S, es a menudo más eficiente utilizar clases que implementan el almacenamiento en búfer, como `BufferedReader` y `BufferedWriter`, para reducir el número de operaciones de E/S físicas.

4. Usamos `Files` para operaciones avanzadas: La clase `Files` del paquete `java.nio.file` proporciona métodos convenientes para realizar operaciones avanzadas en archivos, como copiar, mover, eliminar y más.

5. Utilizamos `java.util.Scanner` para entrada de texto: La clase `Scanner` es útil para la lectura de datos formateados desde un archivo, ya que permite leer y analizar diferentes tipos de datos (enteros, flotantes, cadenas, etc.) de manera sencilla.

6. Manejaremos adecuadamente las excepciones: Siempre maneja las excepciones de E/S de archivos de manera adecuada, ya que pueden ocurrir errores durante la lectura o escritura. Considera usar bloques `try-catch` para capturar y manejar estas excepciones de manera adecuada.

Estas simplemente son algunas buenas prácticas para realizar E/S de archivos en Java. Éstas no son la úna opción posible, la elección de la clase y el patrón o metodología específica dependerá de las necesidades particulares y del tipo de E/S que estés realizando en nuestro proyecto.

1.3.1 Guardando información en un archivo

Vamos a ponernos manos a la masa. Veamos un ejemplo de cómo crear un archivo en Java. Lee atentamente el código e intenta averiguar qué hace:

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class GuardarEnArchivo {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Uso: java GuardarEnArchivo <nombre_del_archivo>");
            System.exit(1);
        }

        String nombreArchivo = args[0];

        try {
            // Abre un archivo para escritura
            BufferedWriter writer = new BufferedWriter(new FileWriter(nombreArchivo));

            System.out.println("Escribe el contenido a guardar en el archivo (Ctrl+D para finalizar en Linux/Mac o Ctrl+Z en Windows):");

            Scanner scanner = new Scanner(System.in);
            while (scanner.hasNextLine()) {
                String linea = scanner.nextLine();
                writer.write(linea);
                writer.newLine();
            }

            // Cierra el archivo
            writer.close();
            System.out.println("Contenido guardado en el archivo '" + nombreArchivo + "'.");
        } catch (IOException e) {
            System.err.println("Error al escribir en el archivo: " + e.getMessage());
        }
```

```
34      }
35  }
```

Efectivamente, como habrás podido averiguar, este programa toma un argumento de línea de comandos que debe ser el nombre del archivo en el que deseas guardar el contenido de la entrada estándar. Luego, utiliza un bucle while para leer líneas de la entrada estándar y escribirlas en el archivo. El programa termina cuando se alcanza el final de la entrada estándar (por ejemplo, cuando se presiona Ctrl+D en Linux/Mac o Ctrl+Z en Windows).

1.3.2 Listando archivos y carpetas

Ahora vamos a complicarlo un poco más. Vamos a ver un ejemplo que, haciendo uso de recursividad, recorre todas las carpetas y subcarpetas de la carpeta que se pasa como parámetro:

```
1  package com.iesvdc.acceso;
2
3  import java.io.File;
4
5  public class ListarContenidoRecursivo {
6      public static void main(String[] args) {
7          if (args.length != 1) {
8              System.out.println("Uso: java ListarContenidoRecursivo <
                   ruta_de_la_carpeta>");
9              System.exit(1);
10         }
11
12         String rutaCarpeta = args[0];
13         listarContenidoRecursivo(new File(rutaCarpeta));
14     }
15
16     public static void listarContenidoRecursivo(File carpeta) {
17         if (carpeta.isDirectory()) {
18             System.out.println("Carpeta: " + carpeta.getAbsolutePath())
                   ;
19
20             File[] archivos = carpeta.listFiles();
21             if (archivos != null) {
22                 for (File archivo : archivos) {
23                     listarContenidoRecursivo(archivo);
24                 }
25             }
26         } else if (carpeta.isFile()) {
27             System.out.println("Archivo: " + carpeta.getAbsolutePath())
                   ;
28         }
29     }
30 }
```

Ejercicio de clase: Intenta modificar el comando anterior para que produzca una salida similar al comando *tree* de Linux (ese comando va metiendo tabuladores o espacios en las subcarpetas de manera que se ve el listado como un árbol, de ahí el nombre del comando).

1.3.3 Borrando archivos

Vamos a complicarlo un poco más, ahora vamos a modificar el programa anterior para que ahora tome dos parámetros: la ruta de la carpeta desde la cual se debe buscar y el nombre del archivo que se debe encontrar y eliminar si existe. Este programa utilizará la API de Java para trabajar con archivos y directorios. Aquí tienes el código:

```java
import java.io.File;

public class EliminarArchivo {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("Uso: java EliminarArchivo <ruta_de_la_carpeta> <nombre_del_archivo>");
            System.exit(1);
        }

        String rutaCarpeta = args[0];
        String nombreArchivo = args[1];

        File carpeta = new File(rutaCarpeta);

        if (!carpeta.isDirectory()) {
            System.err.println("La ruta especificada no es una carpeta válida.");
            System.exit(1);
        }

        File archivo = new File(carpeta, nombreArchivo);

        if (archivo.exists() && archivo.isFile()) {
            if (archivo.delete()) {
                System.out.println("El archivo '" + nombreArchivo + "' ha sido eliminado con éxito.");
            } else {
                System.err.println("No se pudo eliminar el archivo '" + nombreArchivo + "'.");
            }
        } else {
            System.err.println("El archivo '" + nombreArchivo + "' no existe en la carpeta especificada.");
        }
    }
}
```

Este programa toma dos argumentos de línea de comandos: la ruta de la carpeta (<ruta_de_la_carpeta>) y el nombre del archivo que se debe eliminar (<nombre_del_archivo>). A continuación, verifica si la ruta especificada es una carpeta válida, luego intenta eliminar el archivo especificado si existe.

Asegúrate de proporcionar la ruta de la carpeta y el nombre del archivo correctamente en la línea de comandos al ejecutar el programa.

1.4 Clases para gestión de flujos de datos desde/hacia ficheros. Flujos de bytes y de caracteres.

1.4.1 Copiando archivos byte a byte.

Para poder copiar el contenido de un archivo en otro byte a byte, utilizaremos las clases `FileInputStream` y `FileOutputStream` para leer y escribir bytes individualmente:

```
1  import java.io.FileInputStream;
2  import java.io.FileOutputStream;
3  import java.io.IOException;
4
5  public class CopiarArchivoByteAByte {
6      public static void main(String[] args) {
7          if (args.length != 2) {
8              System.out.println("Uso: java CopiarArchivoByteAByte <
                   archivo_origen> <archivo_destino>");
9              System.exit(1);
10         }
11
12         String archivoOrigen = args[0];
13         String archivoDestino = args[1];
14
15         try (FileInputStream input = new FileInputStream(archivoOrigen)
               ;
16              FileOutputStream output = new FileOutputStream(
                    archivoDestino)) {
17
18             int byteLeido;
19             while ((byteLeido = input.read()) != -1) {
20                 output.write(byteLeido);
21             }
22
23             System.out.println("El archivo se copió correctamente.");
24
25         } catch (IOException e) {
```

```
26                 System.err.println("Error al copiar el archivo: " + e.
                       getMessage());
27             }
28         }
29     }
```

Este programa toma dos argumentos de línea de comandos: el nombre del archivo de origen (<archivo_origen>) y el nombre del archivo de destino (<archivo_destino>). Luego, utiliza `FileInputStream` para leer bytes del archivo de origen y `FileOutputStream` para escribir esos bytes en el archivo de destino uno por uno. Asegúrate de proporcionar los nombres de los archivos correctamente en la línea de comandos al ejecutar el programa. El programa copiará el archivo de origen en el archivo de destino byte a byte.

1.5 Operaciones sobre ficheros secuenciales y aleatorios.

El problema de los archivos de texto es que ocupan mucho espacio y, por ejemplo en dispositivos móviles, esto es crítico (no queremos que se llene el almacenamiento de nuestros móviles). En este caso podemos recurrir a archivos binarios.

1.6 Serialización/deserialización de objetos.

Definimos la **serialización** de objetos como el proceso de copiar y almacenar un objeto de memoria volátil (memoria RAM) a memoria persistente (disco). El proceso de **deserialización** es lo contrario, tenemos un objeto almacenado en memoria persistente (un archivo, una base de datos…) y queremos copiarlo a memoria RAM.

A continuación vamos a ver cómo crear un programa en Java que cree varias mascotas y las almacene en un archivo binario. Para esto, primero necesitaremos definir una clase `Mascota` y luego usar `ObjectOutputStream` para escribir objetos de esta clase en un archivo binario:

```
1  import java.io.FileOutputStream;
2  import java.io.ObjectOutputStream;
3  import java.io.Serializable;
4  import java.time.LocalDate;
5  import java.io.IOException;
6
7  class Mascota implements Serializable {
8      private String nombre;
9      private String nombreDueño;
10     private String tipoAnimal;
11     private LocalDate fechaNacimiento;
12
13     public Mascota(String nombre, String nombreDueño, String tipoAnimal
           , LocalDate fechaNacimiento) {
```

```
14          this.nombre = nombre;
15          this.nombreDueño = nombreDueño;
16          this.tipoAnimal = tipoAnimal;
17          this.fechaNacimiento = fechaNacimiento;
18      }
19
20      @Override
21      public String toString() {
22          return "Nombre: " + nombre + "\nDueño: " + nombreDueño + "\
                nTipo de Animal: " + tipoAnimal + "\nFecha de Nacimiento: "
                + fechaNacimiento;
23      }
24  }
25
26  public class AlmacenarMascotas {
27      public static void main(String[] args) {
28          try (FileOutputStream fileOutput = new FileOutputStream("
                mascotas.dat");
29               ObjectOutputStream objectOutput = new ObjectOutputStream(
                    fileOutput)) {
30
31              // Crear tres objetos de mascota
32              Mascota mascota1 = new Mascota("Whiskers", "ObiJuan", "Gato
                    ", LocalDate.of(2020, 5, 10));
33              Mascota mascota2 = new Mascota("Rex", "Maria", "Perro",
                    LocalDate.of(2018, 7, 15));
34              Mascota mascota3 = new Mascota("Tweety", "Pedro", "Pájaro",
                     LocalDate.of(2019, 2, 28));
35              Mascota mascota4 = new Mascota("Yogi", "Mateo", "Oso Pardo"
                    , LocalDate.of(2019, 2, 28));
36
37              // Escribir las mascotas en el archivo binario
38              objectOutput.writeObject(mascota1);
39              objectOutput.writeObject(mascota2);
40              objectOutput.writeObject(mascota3);
41              objectOutput.writeObject(mascota4);
42
43              System.out.println("Las mascotas han sido almacenadas en el
                     archivo mascotas.dat.");
44
45          } catch (IOException e) {
46              System.err.println("Error al almacenar las mascotas: " + e.
                    getMessage());
47          }
48      }
49  }
```

En este programa, creamos una clase `Mascota` que implementa `Serializable`. Esto permite que las instancias de la clase se serialicen y deserialicen automáticamente cuando las escribimos en un archivo binario y las leemos de nuevo.

Luego, en el método `main`, creamos varios objetos `Mascota` y los escribimos en un archivo llamado "mascotas.dat" usando `ObjectOutputStream`. Ten en cuenta que hemos usado **`try`**`-with-resources` para asegurarnos de que los recursos se cierren correctamente. Fíjate que hemos usado LocalDate en vez de Date que está deprecada.

Después de ejecutar el programa, encontrarás el archivo "mascotas.dat" en el directorio de tu proyecto, y contendrá las tres mascotas serializadas en formato binario. Intenta probarlo en tu ordenador.

Ahora vamos a ver el proceso inverso, vamos a leer un archivo binario que contiene objetos `Mascota` serializados y los muestramos. Utilizaremos `ObjectInputStream` para deserializar los objetos y mostrar la información de las mascotas almacenadas. Aquí tienes el programa:

```
1  import java.io.EOFException;
2  import java.io.FileInputStream;
3  import java.io.ObjectInputStream;
4  import java.io.IOException;
5
6  public class LeerMascotas {
7      public static void main(String[] args) {
8          try (FileInputStream fileInput = new FileInputStream("mascotas.
               dat");
9               ObjectInputStream objectInput = new ObjectInputStream(
                    fileInput)) {
10
11              while (true) {
12                  try {
13                      // Leer y deserializar la siguiente mascota
14                      Mascota mascota = (Mascota) objectInput.readObject
                            ();
15                      System.out.println("\nInformación de la mascota:");
16                      System.out.println(mascota.toString());
17                  }catch (EOFException e) {
18                      System.err.println("Alcanzado el final del archivo"
                            );
19                      break; // Se alcanzó el final del archivo
20                  }
21              }
22
23          } catch (IOException | ClassNotFoundException e) {
24              System.err.println("Error al leer el archivo mascotas.dat:
                    " + e.getMessage());
25          }
26      }
27 }
```

Este programa utiliza `ObjectInputStream` para leer y deserializar objetos `Mascota` del archivo binario "mascotas.dat". Utiliza un bucle infinito que se detiene cuando se alcanza el final del archivo. Cada objeto `Mascota` se imprime en la consola con su información. El programa leerá el archivo

"mascotas.dat" que generamos anteriormente y mostrará la información de todas las mascotas almacenadas en ese archivo. Asegúrate de que el archivo "mascotas.dat" exista en el mismo directorio donde se encuentra el programa.

1.7 Situación de aprendizaje: Proyecto generador de personas (marshalling/unmarshalling)

Mientras completamos este sencillo proyecto software vamos a ir aprendiendo algunos conceptos más de este tema.

Se trata de hacer un generador de personas, que tengan aleatoriamente:

- Nombre
- Apellidos
- DNI con letra
- email
- Dirección con ciudad y CP

Posteriormente vamos a serializar los objetos creados en un archivo JSON y un archivo XML. Este proceso (convertir de objeto a JSON y/o XML) recibe el nombre de **marshalling**.

Para terminar haremos el proceso contrario. A partir de los archivos JSON y XML generados, los leemos y creamos los objetos en memoria.

¿Difícil? No tanto, es un problema sencillo. Primero vamos a crear un proyecto Maven y un ejemplo de ejecución (por ejemplo generar 1000 personas). Vamos a ver también cómo hacer pruebas con jUnit de las clases generadas y aprender a hacer el marshalling/unmarshalling de los objetos.

Partimos de la base que tenemos una *base de programación en Java*, sabemos qué es *maven* y los ciclos de vida de desarrollo con dicha herramienta.

Trabajaremos con Visual Studio Code y usaremos algunos de sus plugins para aligerar el tiempo de "fontanería". La documentación la pedimos en Markdown (tienes una guía aquí: https://docs.github.com/es/get-started/writing-on-github y en esta Web: https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet). Para pasar de Markdown a PDF puedes usar el comando:

```
1 $ pandoc Readme.md -o Readme.pdf
```

1.7.1 Preparación del entorno

Aunque es muy similar en Windows, vamos a indicar los pasos a seguir en un equipo con Ubuntu 22.04 LTS. En el caso de Windows, también es posible hacerlo en Ubuntu vía WSL y con las extensiones de

Visual Studio Code WSL y Remote Development.

Necesitamos tener instalado Java, para ello, desde una terminal tecleamos:

```
$ sudo apt install openjdk-17-jdk
```

En el caso de usar Microsoft Windows, una buena alternativa que nos ahorrará tiempo de *fontanería* es la OpenJDK compilada por Microsoft, disponible en su Web para desarrolladores. Para facilitar la instalación recomendamos usar el instalador (fichero *.msi). Si no quieres ensuciar el sistema operativo, también puedes hacerlo en Ubuntu dentro del subsistema Linux (WSL) y sería igual que hacerlo en Linux "puro".

La versión de Maven compatible con Java 17 hay que descargarla de internet, de su web oficial (https://maven.apache.org). En nuestro caso descargamos y descomprimimos en nuestro directorio personal, concretamente en **$HOME/usr** (usamos esta carpeta **usr** para albergar programas descargados como diferentes JDK, Tomcat, GlassFish, Netbeans, etc. sin necesidad de instalar y ensuciar el sistema operativo). En el caso de Windows, con elevación (permisos de administrador), puedes descompromirlo en *C:/Program Files* o, si no quieres hacerlo disponible para todos los usuarios, en un directorio en tu *HOME*.

Para que funcione Maven, en nuestro .bashrc o .zshrc hay que hacer estos cambios:

```
export PATH=$PATH:$HOME/usr/apache-maven-3.8.6/bin
export MAVEN_HOME=$HOME/usr/apache-maven-3.8.6
```

En Windows igualmente hay que añadir a las variables de entorno la ruta a donde lo tenemos descomprimido para que funcione.

Si no tenemos Visual Studio Code instalado, podemos instalarlo con snap (no sería tu caso si usas Ubuntu en WSL):

```
$ sudo snap install code
```

En Windows podemos descargarlo de su Web oficial (https://code.visualstudio.com/download) o bien con Chocolatey (https://chocolatey.org/) con el comando:

```
c:\> choco install vscode
```

1.7.2 Preparación de los datos

Los archivos los tienes en el repositorio, por si no quieres perder tiempo en prepararlos.

Nos descargamos del INE todos los apellidos de españoles con frecuencia igual o mayor de 20: https://www.ine.es/daco/daco42/nombyapel/apellidos_frecuencia.xls.

Convertimos el XLS a CSV, del CSV sacamos los apellidos y lo mandamos a un archivo de texto (en unbuntu tendremos que instalarnos el paquete catdoc para tener el comando xls2csv):

```
$ cd git/generador
$ xls2csv apellidos_frecuencia.xls > apellidos.csv
$ cat apellidos.csv | awk -F "," '{print $2 }' \
    | sed 's/\"//g' | sort > apellidos.txt
```

Hacemos lo mismo con los nombres, también disponible en la Web del INE.

Para las localidades, como sólamente nos piden ciudad, provincia y código postal, usaremos un CSV con todas las ciudades de España que podemos encontrar fácilmente buscando en Google. En nuestro caso particular usaremos el CSV descargado de https://códigospostales.es/listado-de-codigos-postales-de-espana/.

1.7.3 Implementando el generador

Para hacer el generador debemos crear un proyecto Java bien desde terminal, bien con nuestro IDE favorito. Las clases deben estar correctamente ordenadas en *packages* y a su vez, cada funcionalidad debe ir en su *package* correspondiente, por ejemplo, las clases *modelo* (las que modelan los datos base, personas, localidades, etc.) van en un paquete diferente de los tests.

1.7.3.1 Creación del proyecto El proyecto podemos crearlo directamente desde la terminal o desde Visual Studio en modo gráfico. Veamos primero cómo hacerlo en Visual Studio Code.

Nos instalamos las extensiones de Java (cuidado con las versiones preliminares, a veces no funcionan y hay que volver a versiones anteriores):

- Extension Pack for Java (incluye soporte para maven, ejecutar proyectos, etc.)
- Java Code Generators (para generar los constructores, getters y setters)
- Java Run (para que nos salga un texto “Run” sobre cualquier método “main” y ejecutar el código más fácilmente)

Ahora ya podemos crear el proyecto. Para ello usamos la paleta de comandos de Visual Studio Code (pulsamos Ctrl+shift+P) y escribimos maven, de la lista seleccionamos “Maven Project” y pulsamos intro:

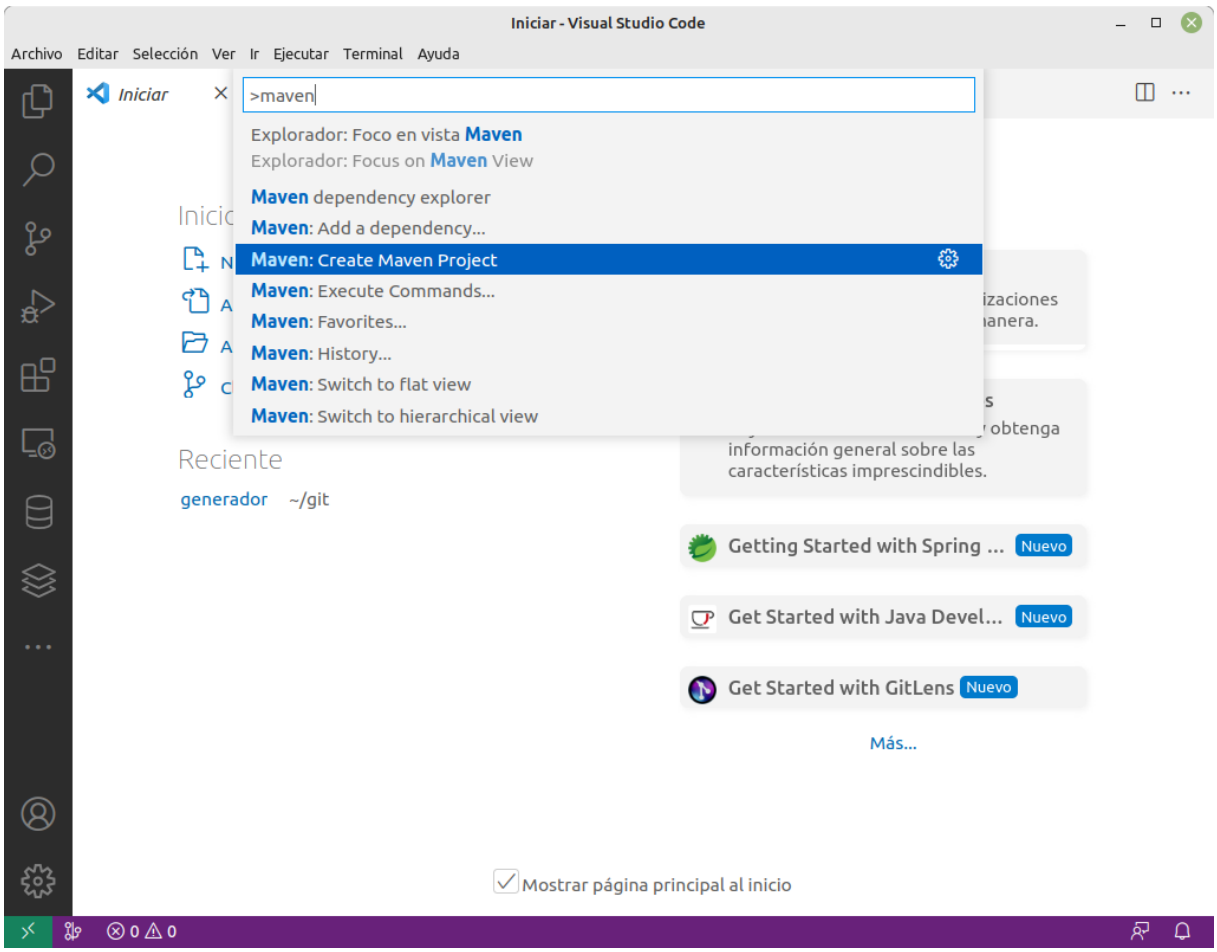

Figura 1: Creación del proyecto maven desde la paleta de comandos de Visual Studio Code

En el siguiente paso, seleccionamos el tipo de proyecto, concretamente “quickstart”:

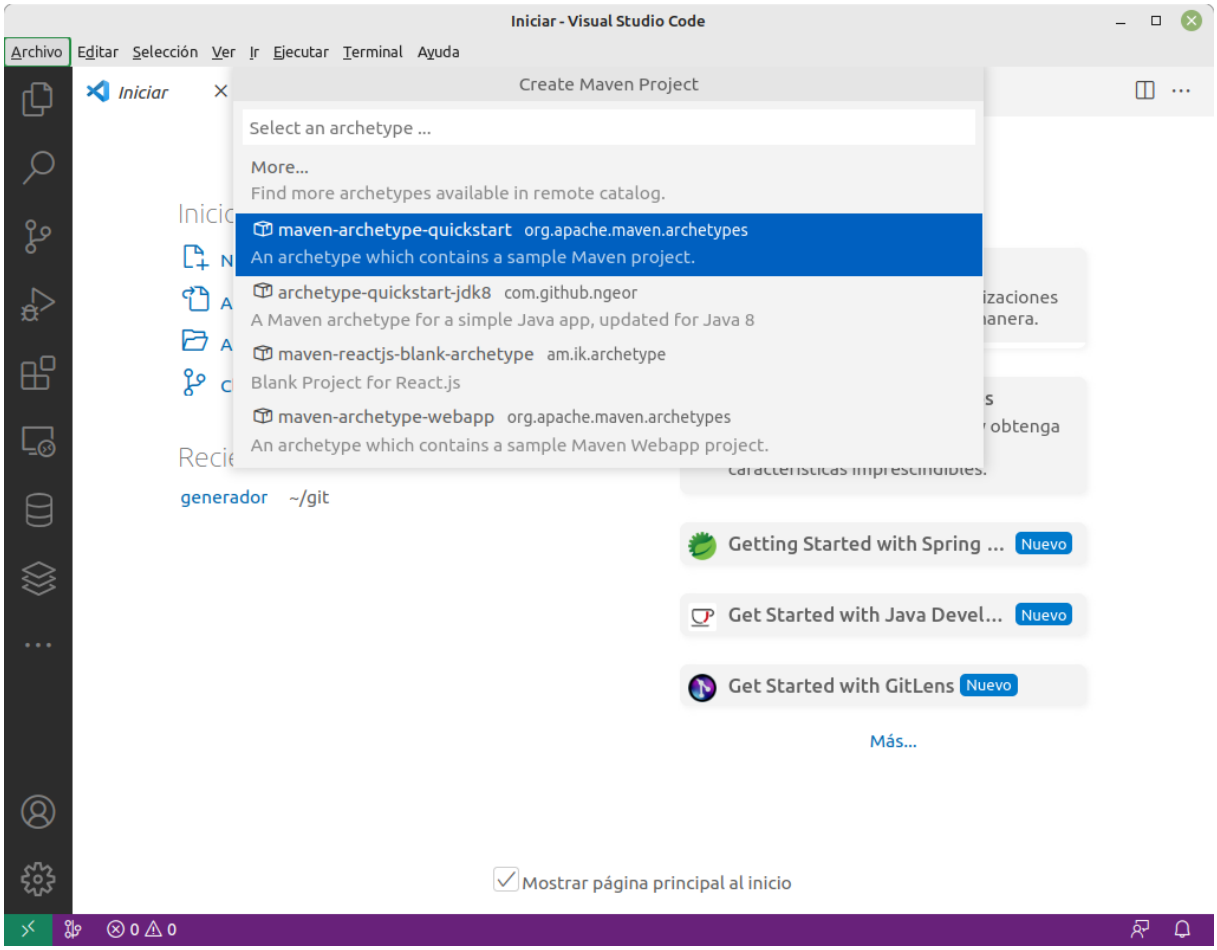

Figura 2: Selección del arquetipo de proyecto Maven: elegimos en el desplegable *quickstart*

A continuación mantenemos la versión 14 (última) del plugin de Maven:

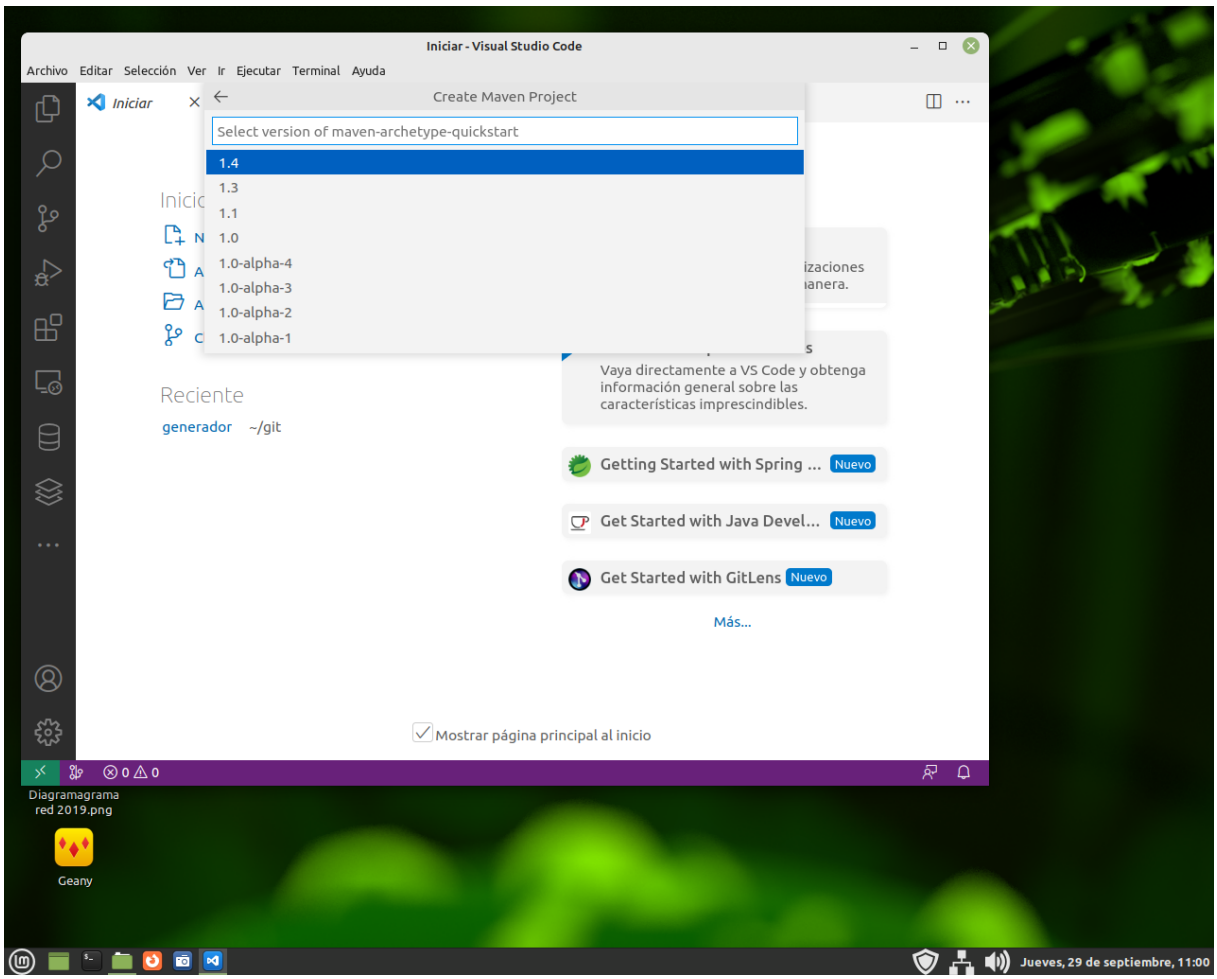

Figura 3: Selección de la versión de plugin de maven que vamos a usar, la última de la lista

Ahora ya podemos darle nombre al paquete (pondremos **com.iesvdc.acceso**):

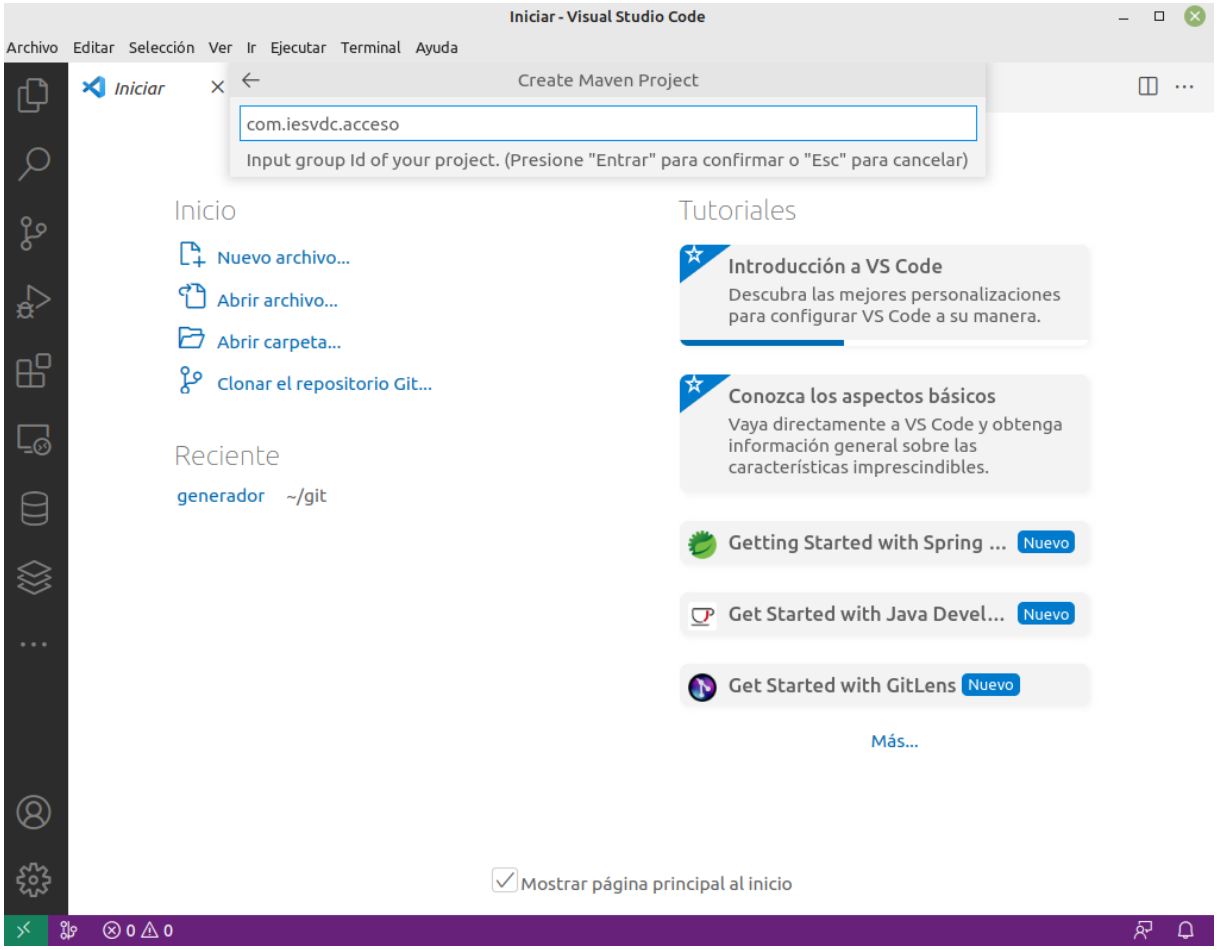

Figura 4: Introducimos el nombre del paquete en la entrada de texto.

Finalmente damos nombre al proyecto: **generador** y a continuación nos preguntará dónde generar la carpeta que contendrá el proyecto y seguidamente que si queremos abrirlo:

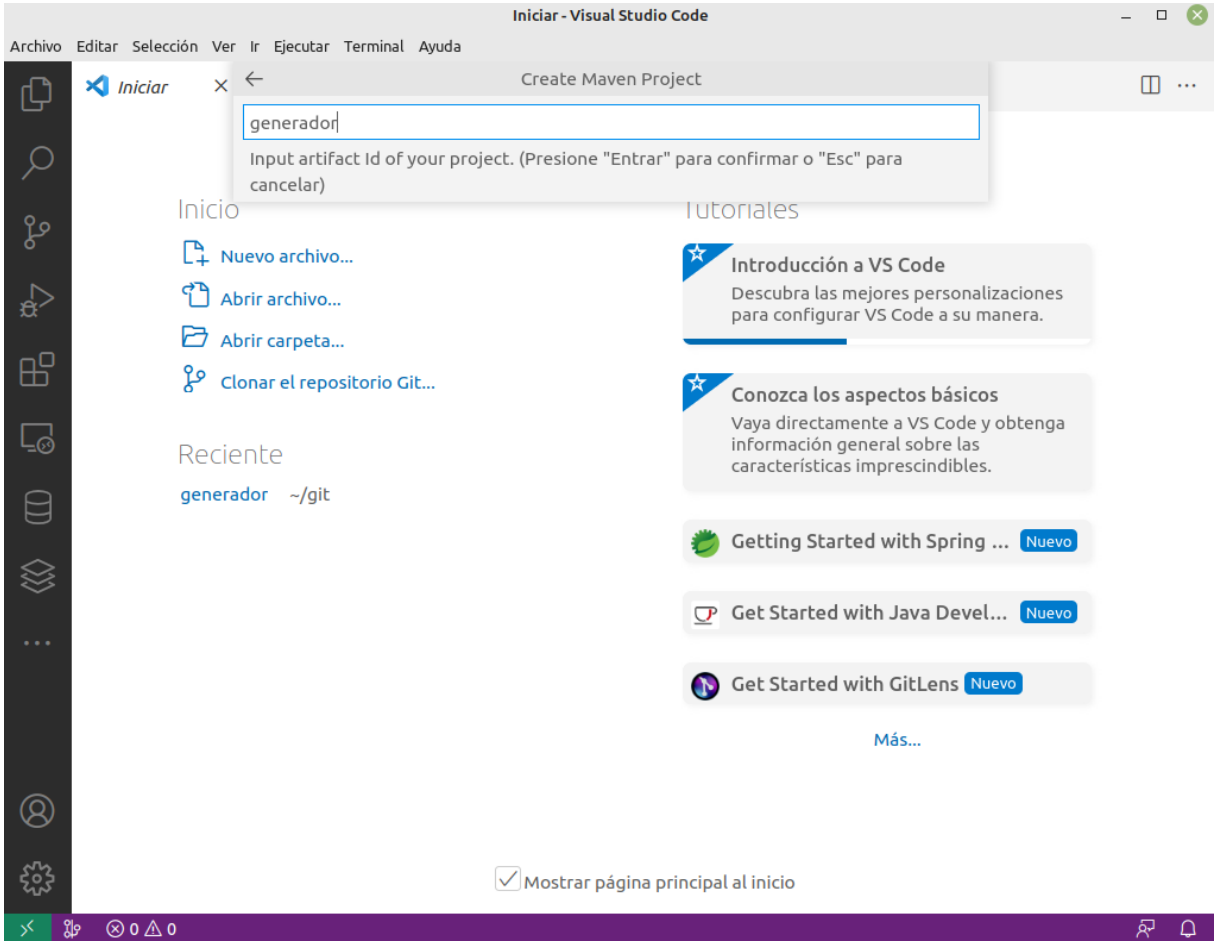

Figura 5: Introducimos el nombre del proyecto, que será el nombre de la aplicación también.

¡Ya podemos empezar a escribir código!

Si te preguntas porqué usar *maven* (o *gradle*), la razón es que este tipo de proyectos pueden ser compilados, testeados y ejecutados sin necesidad de ningún IDE, es decir, podemos probarlos más fácilmente o subirlos a un servidor o contenedor y ponerlos en producción prácticamente clonando el proyecto.

Para la generación en modo terminal, es posible hacerla con el comando (si lo copias, pon todo en una línea para que funcione, aquí lo partimos para que se vea más claro):

```
$ mvn archetype:generate
    -DgroupId=com.iesvdc.acceso
    -DartifactId=generador
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DinteractiveMode=false
```

1.7.3.2 Inicialización del repositorio

Ahora inicializamos el repositorio, añadimos los primeros archivos y cambiamos a la rama desarrollo:

```
git init
git add .gitignore Readme.md pom.xml src docs
echo .vscode >> .gitignore
echo target >> .gitignore
git commit -m "Creación del proyecto"
git branch dev
git checkout dev
```

1.7.3.3 Modelos

El primer paso es modelar las clases *base* que contienen nuestros objetos. Así, crearemos Persona y Personas, Localidad y Localidades… que se encargarán de hacer la *magia* gracias a ciertas anotaciones y un proceso llamado **introspección** que veremos más adelante.

¿Porqué crear una clase Personas o Localidad si simplemente será una lista de Persona o Localidad? Para hacer más legible el código y mucho más sencilla la generación y lectura de XML y JSON (marshalling/unmarshalling).

Es muy importante que nos falte el constructor vacío para que funcionen correctamente las bibliotecas que hacen el binding (conectar el objeto con su representación en XML, JSON, …).

1.7.3.3.1 Localidad

Creamos la clase Localidad. De momento es una clase más, pero modelar correctamente estas clases tomará gran importancia cuando trabajemos con herramientas ORM como Spring, donde aprenderemos el concepto de clases entidad o de POJO (Plain Old Java Object en el argot de Spring) para referirnos a clases como esta que serán persistidas en la base de datos directamente por dichas herraminetas ORM.

```
package com.iesvdc.acceso.modelos;

import java.util.Objects;

public class Localidad {
    private String ciudad;
    private Integer cp;
    private String provincia;

// constructores, getters and setters
```

1.7.3.3.2 Localidades

El listado de localidades es almacenado en una clase especial que se carga de un archivo. Los códigos postales varian a lo largo del tiempo, de hecho empresas como Correos cobran por ofreces un servicio de códigos postales actualizado. En nuestro caso simplemente lo vamos

a cargar de un archivo CSV, pero podría ser igualmente una API REST o una base de datos remota consultados periódicamente.

```java
package com.iesvdc.acceso.modelos;

import java.io.BufferedReader;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class Localidades {
    private List<Localidad> localidades;


    public Localidades() {
        this.localidades = new ArrayList<Localidad>();
    }

    // getters y setters...

    /**
     * Carga las localidades de un archivo
     */
    public void load(String filename){
        this.localidades = new ArrayList<Localidad>();
        Path fichero = Paths.get(filename);
        try (BufferedReader br = Files.newBufferedReader(fichero)) {
            String linea = br.readLine();
            // nos saltamos la primera línea que tiene las cabeceras
            if (linea!= null)
                linea = br.readLine();
            while (linea!=null) {
                String datos[] = linea.split(";");
                // Lodalidad: String ciudad, Integer cp, String
                   provincia
                // CSV: Provincia;Población;Código Postal
                this.localidades.add(
                    new Localidad(
                        datos[1],
                        Integer.parseInt(datos[2]),
                        datos[0]));
                linea = br.readLine();
            }
        } catch (Exception e) {
            System.out.println(e.getLocalizedMessage());
        }
    }
    // sobrecarga del método anterior sin parámetro.
```

```
48      public void load(){
49          this.load("ciudades.csv");
50      }
51
52      /**
53       * Devuelve una localidad de la lista al azar
54       */
55      Localidad getRandomLocalidad(){
56          int tam = this.localidades.size();
57          int donde = (int) Math.round(Math.random()*(tam-1));
58          return this.localidades.get(donde);
59      }
```

1.7.3.3.3 Persona Para almacenar información de personas, usamos la clase modelo *Persona*. Para el sexo usaremos un Enum. Queremos aprovechar para comentar que cuando implementemos ACLs (listas de control de acceso) para usuarios (similar a Persona), el tipo de usuario debería ser un enum o una lista de enum (o simplemente un String) para que sea más fácil la gestión de roles. Si usamos herencia en estos casos (especialización: Hombre, Mujer, Indefinido que heredan de Persona) compicamos enormemente la lógica de la aplicación sin sentido. La herencia hay que usarla cuando realmente es necesaria, donde realmente tenemos una especialización que parte de un tipo base al que añadimos nuevas funcionalidades y atributos (no hay herencia si mantenemos los mismos atributos/métodos o los fabricamos sintéticamente para justificarla sin que nos los pidan los requisitos).

Fichero Persona.java:

```
1  package com.iesvdc.acceso.modelos;
2
3  import java.util.Objects;
4
5  public class Persona{
6
7      private String nombre;
8      private String apellido1;
9      private String apellido2;
10     private String dni;
11     private Sexo sexo;
12     private Localidad localidad;
13
14
15     public Persona() {
16     }
17
18     public Persona(
19         String nombre,
20         String apellido1,
21         String apellido2,
22         String dni,
```

```
23          Sexo sexo,
24          Localidad localidad) {
25          this.nombre = nombre;
26          this.apellido1 = apellido1;
27          this.apellido2 = apellido2;
28          this.dni = dni;
29          this.sexo = sexo;
30          this.localidad = localidad;
31      }
32
33      // getters, setters, toString...
34
35  }
```

Para ampliar: Investiga qué es Lombok para Java y piensa para qué lo usarías con la clase *Persona*.

1.7.3.3.4 Personas Personas es la clase que contiene la lista de *Persona* y que nos ayudará en el proceso de marshalling/unmarshalling.

```
1  package com.iesvdc.acceso.modelos;
2  import java.util.ArrayList;
3  import java.util.List;
4  import java.util.Objects;
5
6  public class Personas {
7
8      private List<Persona> personas;
9      // constructores, getters y setters...
```

1.7.3.3.5 PersonasGenerator Creamos una clase diferente, que hereda de personas sus métodos y propiedades para especializarnos en la tarea de generar listas de personas de manera automatizada, para crear datos de "*mockeo*" o datos falsos para probar nuestras aplicaciones. Jugando con las anotaciones podríamos haber usado sólo la clase Personas, pero **especializando** queda el **código** más legible y es más **reutilizable**.

```
1  package com.iesvdc.acceso.modelos;
2  import java.nio.file.Files;
3  import java.nio.file.Path;
4  import java.nio.file.Paths;
5  import java.util.List;
6
7  public class PersonasGenerator extends Personas {
8
9      private Localidades locs;
10     private List<String> apellidos;
11     private List<String> nombresHombre;
```

```
12      private List<String> nombresMujer;
13      private Dominios dominios;
14
15      public PersonasGenerator(){
16          super();
17      }
18
19      List<String> loadFileinArray(String filename) {
20          List<String> array = null;
21          Path fichero = Paths.get(filename);
22          try {
23              array = Files.readAllLines(fichero);
24          } catch (Exception e) {
25              System.out.println("::PERSONAS::loadFielinArray(): Error al
                    cargar fichero: " +
26                  filename + "::" +
27                  e.getMessage());
28          }
29          return array;
30      }
31
32      public void load(String apellidosFilename,
33          String nombresHombreFilename, String nombresMujerFilename) {
34          this.locs = new Localidades();
35          this.locs.load();
36          this.dominios = new Dominios();
37          this.dominios.load();
38          this.apellidos = loadFileinArray(apellidosFilename);
39          this.nombresMujer = loadFileinArray(nombresMujerFilename);
40          this.nombresHombre = loadFileinArray(nombresHombreFilename);
41      }
42
43      public void load(){
44          load("apellidos.txt","nombre_hombres.txt", "nombre_mujeres.txt"
                );
45      }
46
47      private int dado(int tam){
48          return ( (int) Math.round(Math.random()*(tam-1)));
49      }
50
51      private String getRandApellido(){
52          return this.apellidos.get(dado(this.apellidos.size()));
53      }
54
55      private static char calcularLetra(int dni){
56          String caracteres="TRWAGMYFPDXBNJZSQVHLCKE";
57          int resto = dni%23;
58          return caracteres.charAt(resto);
59      }
60
```

```
61      private String getRandDni(){
62          int num = dado(100000000);
63          return Integer.toString(num)+calcularLetra(num);
64      }
65
66      private String getRandNombre(Sexo sexo) {
67          String salida="John Doe";
68          switch (sexo){
69              case HOMBRE:
70                  salida = this.nombresHombre.get(
71                      dado(this.nombresHombre.size()));
72                  break;
73              case MUJER:
74                  salida = this.nombresMujer.get(
75                      dado(this.nombresMujer.size()));
76                  break;
77              case X:
78                  if (dado(10)>5){
79                      salida = this.nombresHombre.get(
80                          dado(this.nombresHombre.size()));
81                  } else {
82                      salida = this.nombresMujer.get(
83                          dado(this.nombresMujer.size()));
84                  }
85          }
86          return salida;
87      }
88
89      private Sexo getRandSexo(){
90          Sexo[] sexos = Sexo.values();
91          int num = dado(sexos.length);
92          return sexos[num];
93      }
94
95      private String getRandDominio(){
96          return this.dominios.getDominios().get(
97              dado(this.dominios.getDominios().size()));
98      }
99
100     private String generateEmail(String nombre, String ap1, String ap2)
            {
101         String salida = "";
102         salida = nombre.substring(0, 1) +
103             ap1.substring(0,3) +
104             ap2.substring(0,3) + "@" +
105             this.getRandDominio();
106         return (salida.replaceAll("\\s+", "").toLowerCase());
107     }
108
109     public void generate(int cuantos){
110         if (this.apellidos==null ||
```

```
111             this.nombresHombre==null ||
112             this.nombresMujer == null) {
113                 this.load();
114             }
115         for (int i=0; i<cuantos; i++){
116             Sexo sexo = this.getRandSexo();
117             String nombre = this.getRandNombre(sexo);
118             String apellido1 = this.getRandApellido();
119             String apellido2 = this.getRandApellido();
120             this.add(
121                 new Persona(
122                     nombre,
123                     apellido1,
124                     apellido2,
125                     this.getRandDni(),
126                     sexo,
127                     this.locs.getRandomLocalidad(),
128                     generateEmail(nombre, apellido1, apellido2))
129             );
130         }
131     }
132
133 }
```

1.7.4 Probando clases

Según si disponemos del código o no, en general podemos clasificar las pruebas en:

- **Estáticas**: se realizan sin ejecutar el código de la aplicación.Puede referirse a la revisión de documentos, ya que no se hace una ejecución de código. Esto se debe a que se pueden realizar "pruebas de escritorio" con el objetivo de seguir los flujos de la aplicación.
- **Dinámicas**: pruebas que para su ejecución requieren la ejecución de la aplicación. Debido a la naturaleza dinámica de la ejecución de pruebas es posible medir con mayor precisión el comportamiento de la aplicación desarrollada. Las pruebas dinámicas permiten el uso de técnicas de caja negra y caja blanca con mayor amplitud.
  - *De caja negra*: es estudiado desde el punto de vista de las entradas que recibe y las salidas o respuestas que produce, sin tener en cuenta su funcionamiento interno.
  - *De caja blanca*: se centran en los detalles procedimentales del software, por lo que su diseño está fuertemente ligado al código fuente. El ingeniero de pruebas escoge distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa y cerciorarse de que se devuelven los valores de salida adecuados.

Otra clasificación de las pruebas es según lo que verifican: son las llamadas **pruebas funcionales**. Una prueba funcional es una prueba basada en la ejecución, revisión y retroalimentación de las

funcionalidades previamente diseñadas para el software (requisitos funcionales). Hay distintos **tipos** como por ejemplo:

- **Pruebas unitarias**: La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto. Luego, con las Pruebas de Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.
- **Pruebas de componentes**: La palabra componente nos hace pensar en una unidad o elemento con propósito bien definido que, trabajando en conjunto con otras, puede ofrecer alguna funcionalidad compleja. Un componente es una pieza de software que cumple con dos características: no depende de la aplicación que la utiliza y, se puede emplear en diversas aplicaciones. Así estas pruebas están un paso justo por encima de las unitarias, cuando integramos varios componentes en lo que podría ser una API.
- **Pruebas de integración**: se realizan en el ámbito del desarrollo de software una vez que se han aprobado las pruebas unitarias y lo que prueban es que todos los elementos unitarios que componen el software, funcionan juntos correctamente probándolos en grupo. Se centra principalmente en probar la comunicación entre los componentes y sus comunicaciones ya sea hardware o software.
- **Pruebas de sistema**: son un tipo de pruebas de integración, donde se prueba el contenido del sistema completo.
- **Pruebas de humo**: son aquellas pruebas que pretenden evaluar la calidad de un producto de software previo a una recepción formal, ya sea al equipo de pruebas o al usuario final, es decir, es una revisión rápida del producto de software para comprobar que funciona y no tiene defectos que interrumpan la operación básica del mismo.
- **Pruebas alpha**: realizadas cuando el sistema está en desarrollo y cuyo objetivo es asegurar que lo que estamos desarrollando es probablemente correcto y útil para el cliente.
- **Pruebas beta**: cuando el sistema está teóricamente correcto y pasa a ejecutarse en un entorno real. Es la fase siguiente a las pruebas Alpha.
- **Pruebas de regresión**: cualquier tipo de pruebas de software que intentan descubrir errores (bugs), carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del software, causados por la realización de un cambio en el programa. Se evalúa el correcto funcionamiento del software desarrollado frente a evoluciones o cambios funcionales.

1.7.4.1 jUnit

JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

1.7.4.2 Test de Localidades En este ejemplo veremos cómo comprobar si funcionan algunos de los métodos de Localidades. En el pom.xml hemos de indicar a Maven que descargue jUnit. Mira en el raíz del proyecto el fichero **pom.xml** y verás esta dependencia:

```
...
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
...
</dependencies>
```

```
package com.iesvdc.acceso.modelos;

import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class LocalidadesTest {

    @Test
    public void testGetLocalidades() throws Exception {
        Localidad loc1, loc2;
        loc1 = new Localidad("JAEN", 23002, "JAEN");
        Localidades locs = new Localidades();
        locs.add(loc1);
        loc2 = new Localidad("JAEN", 23005, "JAEN");
        locs.add(loc2);
        if (locs.getLocalidades().size()==2) {
            Localidad loc3, loc4;
            loc3 = locs.getLocalidades().get(0);
            loc4 = locs.getLocalidades().get(1);
            if ( loc1.equals(loc3) && loc2.equals(loc4) )
                assertTrue(true);
            else
                assertTrue(false);
        } else
            assertTrue(false);
    }

    @Test
    public void testLoad() throws Exception {
```

```
30          Localidades locs = new Localidades();
31          Localidades locs2 = new Localidades();
32          locs.load();
33          locs2.load();
34          if (locs.equals(locs2)) {
35              assertTrue(true);
36          } else {
37              assertTrue(false);
38          }
39      }
40
41  }
```

1.7.5 Marshalling y unmarshalling

Hacemos el parseo (del inglés parsing) a un fichero XML cuando lo cargamos en un árbol en la memoria del ordenador o bien lo vamos leyendo y ejecutando instrucciones según las etiquetas que entran.

Hacer binding de un objeto a XML es "conectar" propiedades de ese objeto a etiquetas del fichero XML. Al proceso de volcar o serializar el objeto a un XML se le da el nombre de marshalling, al proceso contrario unmarshalling.

Resumiendo, se denomina "marshalling" al proceso de volcar la representación en memoria de un objeto a un formato que permita su almacenamiento o transmisión, siendo "unmarshalling" el proceso contrario. Basicamente esto seria serialización y deserialización.

JAXB es la tecnología de java que provee un API y una herramienta para ligar el esquema XML a una representación en código java. Básicamente esta API nos provee un conjunto de Annotations y clases para hacer el binding entre nuestra estructura en los objetos Java y el XML, o mejor aún si nosotros poseemos el XSD del XML podes utilizar la herramienta xjc (Viene por defecto en la distrubución con la JDK de Oracle) para que nos genere la clases del dominio.

Fíjate en las anotaciones que comienzan con @ en el siguiente código fuente:

```
1  package com.iesvdc.acceso.modelos;
2  import java.util.ArrayList;
3  import java.util.List;
4  import java.util.Objects;
5
6  import javax.xml.bind.annotation.XmlAccessType;
7  import javax.xml.bind.annotation.XmlAccessorType;
8  import javax.xml.bind.annotation.XmlRootElement;
9  import javax.xml.bind.annotation.XmlElement;
10
11 @XmlRootElement(name = "personas")
12 @XmlAccessorType (XmlAccessType.FIELD)
13 public class Personas {
```

```
14
15      @XmlElement(name="persona")
16      private List<Persona> personas;
17      ...
```

Con XmlRootElement estamos indicando que, en caso de *marshalling*, se trata de un elemento raíz. Con XmlElement indicamos que es un elemento de XML y además le damos un nombre (en caso contrario tomaría como nombre el mismo de la variable).

Aunque parezca algo relativamente sencillo, este tipo de tecnología es muy necesaria y utilizada, por ejemplo, cuando el frontend (ej. una APP móvil o una Web cargada en el navegador) se comunica con el backend (una API o servicio REST). Cuando mandamos datos desde una APP al servidor esos datos se serializan, es decir, se les hace un "marshalling" para transmitirlos. Una vez en el servidor, para convertirlos en objetos del servicio que estamos atacando y que corre en dicho servidor, hay que deserializarlo o hacerles el "unmarshalling".

1.7.5.1 Marshaller XML

```
1  import javax.xml.bind.JAXBContext;
2  import javax.xml.bind.JAXBException;
3  import javax.xml.bind.Marshaller;
4
5  import com.iesvdc.acceso.modelos.Personas;
6  import com.iesvdc.acceso.modelos.PersonasGenerator;
7
8  /**
9   * Ejemplo de Marshaller de personas.
10  * Versión XML.
11  */
12 public class MarshallerXML
13 {
14     public static void main( String[] args )
15     {
16         Personas lista = new Personas();
17         PersonasGenerator pg = new PersonasGenerator();
18         pg.generate(10);
19
20         lista = new Personas(pg.getPersonas());
21         JAXBContext jaxbContext;
22
23         try {
24             jaxbContext = JAXBContext.newInstance(lista.getClass());
25             Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
26             jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT
                   , true);
27             jaxbMarshaller.marshal(lista, new File("personas.xml"));
28
29         } catch (JAXBException e) {
30             e.printStackTrace();
```

```
31          }
32      }
33  }
```

1.7.5.2 Unmarshaller XML

```
1  package com.iesvdc.acceso;
2
3  import java.io.File;
4
5  import javax.xml.bind.JAXBContext;
6  import javax.xml.bind.JAXBException;
7  import javax.xml.bind.Unmarshaller;
8
9  import com.iesvdc.acceso.modelos.Personas;
10
11 /**
12  * Ejemplo de unmarshaller de personas.
13  * Versión XML.
14  */
15 public class UnMarshallerXML
16 {
17     public static void main( String[] args )
18     {
19         Personas lista = new Personas();
20         JAXBContext jaxbContext;
21
22         try {
23
24             jaxbContext = JAXBContext.newInstance(lista.getClass());
25             Unmarshaller jaxbUnmarshaller = jaxbContext.
                   createUnmarshaller();
26             Object objeto = jaxbUnmarshaller.unmarshal(new File("
                   personas.xml"));
27             lista = (Personas) objeto;
28             System.out.println(lista.toString());
29
30         } catch (JAXBException e) {
31             e.printStackTrace();
32         }
33
34     }
35 }
```

1.7.5.3 Marshaller JSON Para poder serializar o deserializar el objeto a o desde JSON (marshalling/unmarshalling de JSON), usamos la librería Eclipse Link MOxY. **MOxY** permite a los programadores hacer el binding de clases a XML y/o JSON gracias a la información de mapeo que se proporciona en forma de anotaciones (las mismas que vismos en JAXB).

Tenemos que añadir a nuestro *pom.xml* inicial la siguiente dependencia:

```
...
  <dependencies>
    ...
    <dependency>
      <groupId>org.eclipse.persistence</groupId>
      <artifactId>org.eclipse.persistence.moxy</artifactId>
      <version>2.5.2</version>
    </dependency>
  </dependencies>
...
```

Ahora ya podemríamos generar JSON directamente desde el marshaller. Previamente tendremos que establecer EclipseLink bien en el jaxb.properties (en el raiz del paquete) o bien desde el “main” para hacer el bootstrap al JAXBContext y que se use para el binding esta biblioteca en vez de la nativa de Java que usamos con anterioridad.

```
package com.iesvdc.acceso;

import java.io.File;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import org.eclipse.persistence.jaxb.MarshallerProperties;

import com.iesvdc.acceso.modelos.Personas;
import com.iesvdc.acceso.modelos.PersonasGenerator;

/**
 * Ejemplo de generador de personas.
 * JSON version.
 */
public class MarshallerJSON
{
    public static void main( String[] args )
    {
        Personas lista = new Personas();
        PersonasGenerator pg = new PersonasGenerator();
        pg.generate(10);

        lista = new Personas(pg.getPersonas());
        JAXBContext jaxbContext;

        try {
            /* Necesitamos establecer EclipseLink bien el el jaxb.
               properties
            (en el raiz del paquete) o bien desde el "main" para hacer
               el
```

```
31              bootstrap al JAXBContext */
32              System.setProperty(
33                  "javax.xml.bind.JAXBContextFactory",
34                  "org.eclipse.persistence.jaxb.JAXBContextFactory");
35
36              jaxbContext = JAXBContext.newInstance(lista.getClass());
37
38              Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
39              jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT
                    , true);
40
41              //Para JSON
42              jaxbMarshaller.setProperty(
43                  MarshallerProperties.MEDIA_TYPE, "application/json");
44              jaxbMarshaller.setProperty(
45                  MarshallerProperties.JSON_INCLUDE_ROOT, true);
46
47              jaxbMarshaller.marshal(lista, new File("personas.json"));
48
49          } catch (JAXBException e) {
50              e.printStackTrace();
51          }
52
53      }
54  }
```

1.7.5.4 Unmarshaller JSON Para hacer el proceso inverso, el unmarshalling, es exactamente igual pero usando el objeto “Unmarshaller”:

```
1  package com.iesvdc.acceso;
2
3  import java.io.File;
4
5  import javax.xml.bind.JAXBContext;
6  import javax.xml.bind.JAXBException;
7  import javax.xml.bind.Unmarshaller;
8  import org.eclipse.persistence.jaxb.UnmarshallerProperties;
9
10 import com.iesvdc.acceso.modelos.Personas;
11
12 /**
13  * Ejemplo de unmarshaller de personas.
14  * Versión JSON
15  */
16 public class UnMarshallerJSON
17 {
18     public static void main( String[] args )
19     {
20         Personas lista = new Personas();
```

```
21
22          JAXBContext jaxbContext;
23
24          try {
25
26              System.setProperty(
27                  "javax.xml.bind.context.factory",
28                  "org.eclipse.persistence.jaxb.JAXBContextFactory");
29
30              jaxbContext = JAXBContext.newInstance(lista.getClass());
31
32              Unmarshaller jaxbUnmarshaller = jaxbContext.
                    createUnmarshaller();
33              // Para JSON
34              jaxbUnmarshaller.setProperty(
35                  UnmarshallerProperties.MEDIA_TYPE, "application/json");
36              jaxbUnmarshaller.setProperty(
37                  UnmarshallerProperties.JSON_INCLUDE_ROOT, true);
38
39              Object objeto = jaxbUnmarshaller.unmarshal(
40                  new File("personas.json"));
41              lista = (Personas) objeto;
42              System.out.println(lista.toString());
43
44          } catch (JAXBException e) {
45              e.printStackTrace();
46          }
47      }
48  }
```

1.8 Bibliografía

- http://juangualberto.github.io/acceso/tema01/gestin_de_ficheros_y_directorios_con_java.html
- Lista de todos los dominios de correo
- Documentación oficial de la JDK API 17