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.

Descripción del proyecto

Mientras completamos este sencillo proyecto software vamos a ir aprendiendo algunos de los conceptos necesarios para este tema.

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

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

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:

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

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
```

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/.

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.

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":

En el siguiente paso, seleccionamos el tipo de proyecto, concretamente "quick-start":

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

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

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:

¡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.

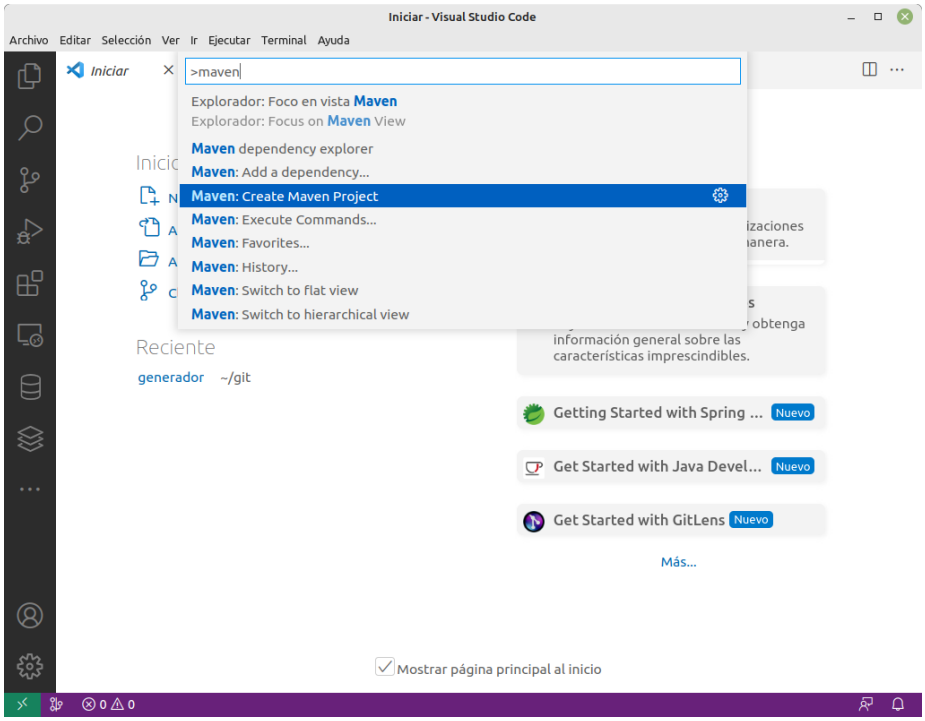

Figure 1: Creación del proyecto

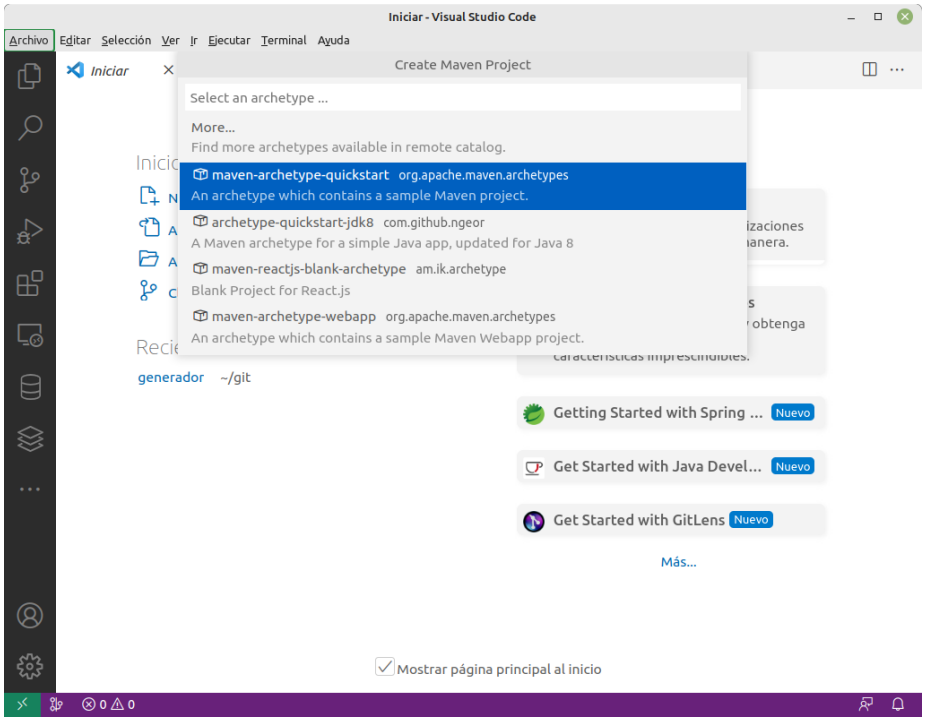

Figure 2: Selección del tipo de proyecto

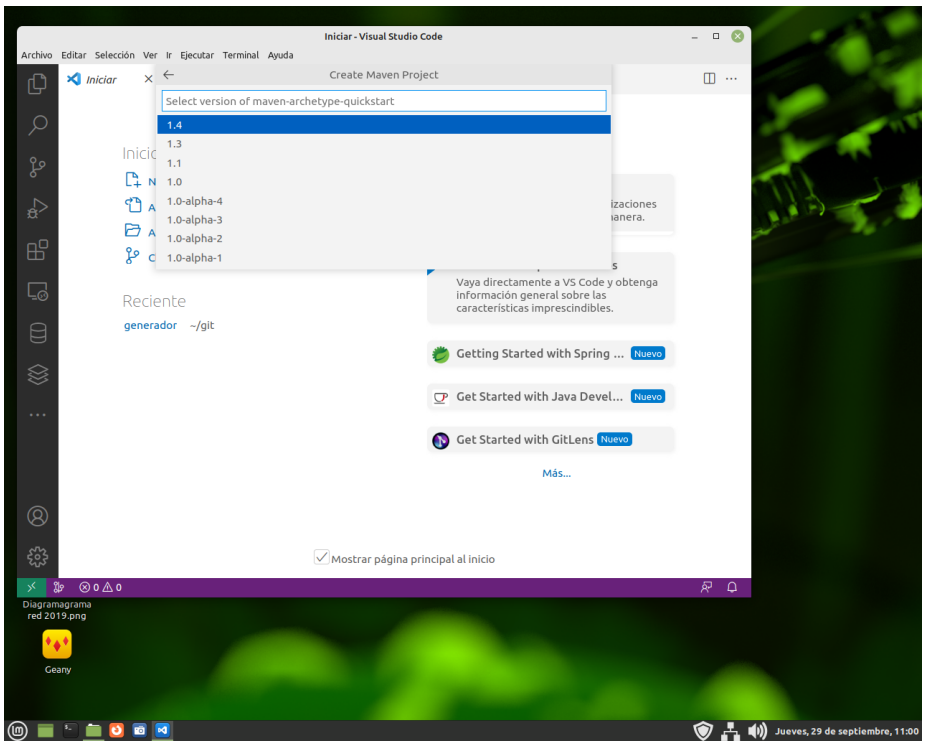

Figure 3: Selección de la versión de plugin de maven que vamos a usar

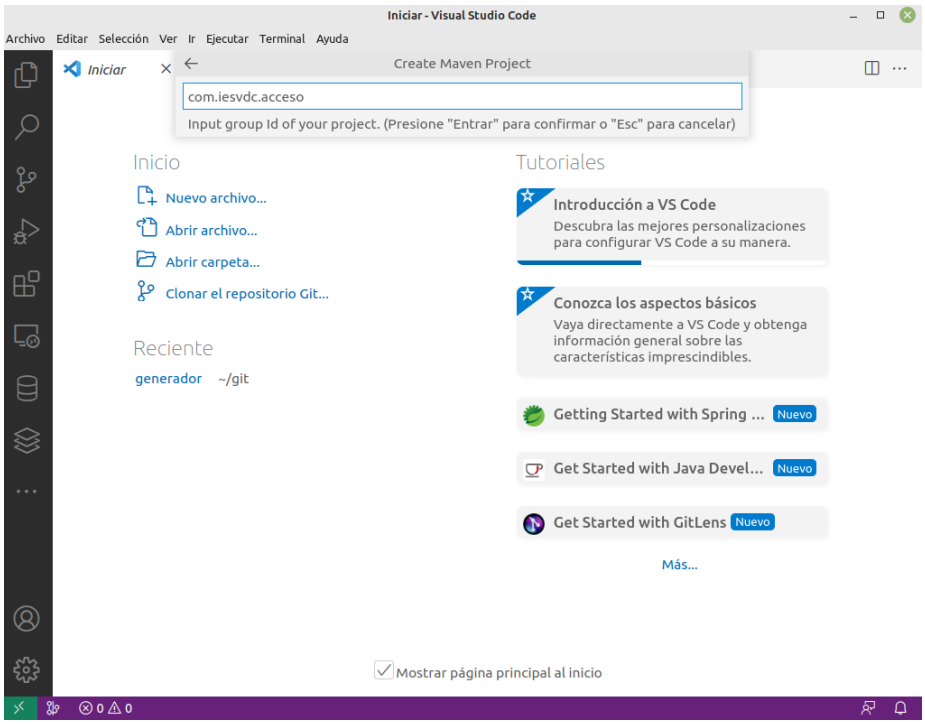

Figure 4: Introducimos el nombre del paquete

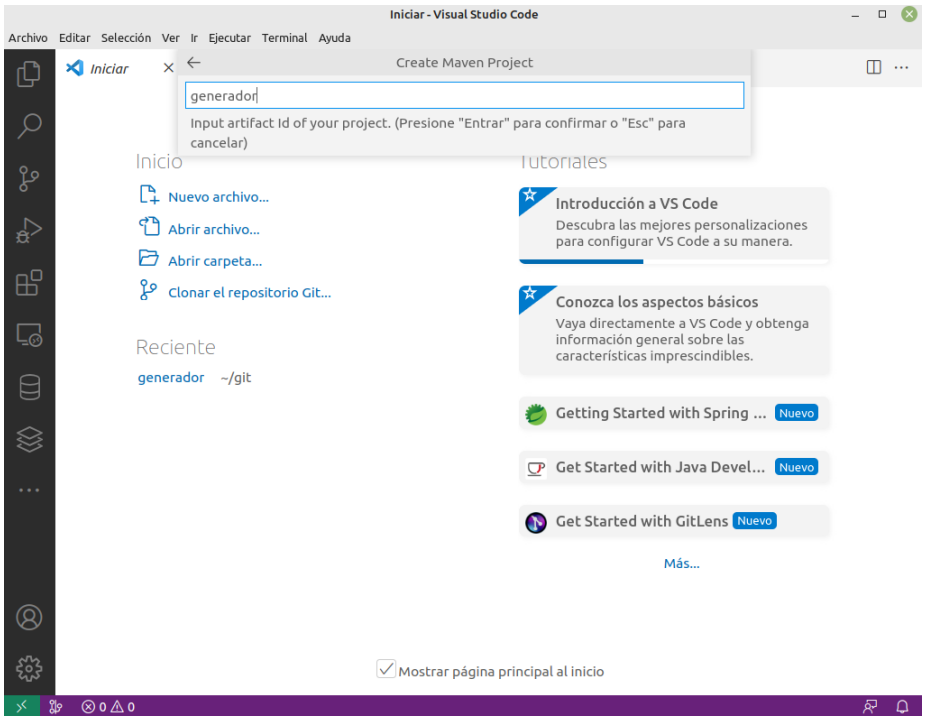

Figure 5: Introducimos el nombre del proyecto

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
```

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
```

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, ...).

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;
```

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

// constructores, getters and setters
```

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.
    public void load(){
        this.load("ciudades.csv");
    }

    /**
     * Devuelve una localidad de la lista al azar
     */
    Localidad getRandomLocalidad(){
        int tam = this.localidades.size();
        int donde = (int) Math.round(Math.random()*(tam-1));
        return this.localidades.get(donde);
    }
```

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:

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

import java.util.Objects;
```

```java
public class Persona{

    private String nombre;
    private String apellido1;
    private String apellido2;
    private String dni;
    private Sexo sexo;
    private Localidad localidad;


    public Persona() {
    }

    public Persona(
        String nombre,
        String apellido1,
        String apellido2,
        String dni,
        Sexo sexo,
        Localidad localidad) {
        this.nombre = nombre;
        this.apellido1 = apellido1;
        this.apellido2 = apellido2;
        this.dni = dni;
        this.sexo = sexo;
        this.localidad = localidad;
    }

    // getters, setters, toString...

}
```

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

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

```java
package com.iesvdc.acceso.modelos;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class Personas {
```

```
    private List<Persona> personas;
    // constructores, getters y setters...
```

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**.

```
package com.iesvdc.acceso.modelos;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;

public class PersonasGenerator extends Personas {

    private Localidades locs;
    private List<String> apellidos;
    private List<String> nombresHombre;
    private List<String> nombresMujer;
    private Dominios dominios;

    public PersonasGenerator(){
        super();
    }

    List<String> loadFileinArray(String filename) {
        List<String> array = null;
        Path fichero = Paths.get(filename);
        try {
            array = Files.readAllLines(fichero);
        } catch (Exception e) {
            System.out.println("::PERSONAS::loadFielinArray(): Error al cargar fichero: " +
                filename + "::" +
                e.getMessage());
        }
        return array;
    }

    public void load(String apellidosFilename,
        String nombresHombreFilename, String nombresMujerFilename) {
        this.locs = new Localidades();
        this.locs.load();
        this.dominios = new Dominios();
```

```
        this.dominios.load();
        this.apellidos = loadFileinArray(apellidosFilename);
        this.nombresMujer = loadFileinArray(nombresMujerFilename);
        this.nombresHombre = loadFileinArray(nombresHombreFilename);
    }

    public void load(){
        load("apellidos.txt","nombre_hombres.txt", "nombre_mujeres.txt");
    }

    private int dado(int tam){
        return ( (int) Math.round(Math.random()*(tam-1)));
    }

    private String getRandApellido(){
        return this.apellidos.get(dado(this.apellidos.size()));
    }

    private static char calcularLetra(int dni){
        String caracteres="TRWAGMYFPDXBNJZSQVHLCKE";
        int resto = dni%23;
        return caracteres.charAt(resto);
    }

    private String getRandDni(){
        int num = dado(100000000);
        return Integer.toString(num)+calcularLetra(num);
    }

    private String getRandNombre(Sexo sexo) {
        String salida="John Doe";
        switch (sexo){
            case HOMBRE:
                salida = this.nombresHombre.get(
                    dado(this.nombresHombre.size()));
                break;
            case MUJER:
                salida = this.nombresMujer.get(
                    dado(this.nombresMujer.size()));
                break;
            case X:
                if (dado(10)>5){
                    salida = this.nombresHombre.get(
                        dado(this.nombresHombre.size()));
                } else {
                    salida = this.nombresMujer.get(
```

```
                    dado(this.nombresMujer.size()));
            }
        }
        return salida;
    }

    private Sexo getRandSexo(){
        Sexo[] sexos = Sexo.values();
        int num = dado(sexos.length);
        return sexos[num];
    }

    private String getRandDominio(){
        return this.dominios.getDominios().get(
            dado(this.dominios.getDominios().size()));
    }

    private String generateEmail(String nombre, String ap1, String ap2){
        String salida = "";
        salida = nombre.substring(0, 1) +
            ap1.substring(0,3) +
            ap2.substring(0,3) + "@" +
            this.getRandDominio();
        return (salida.replaceAll("\\s+", "").toLowerCase());
    }

    public void generate(int cuantos){
        if (this.apellidos==null ||
            this.nombresHombre==null ||
            this.nombresMujer == null) {
                this.load();
            }
        for (int i=0; i<cuantos; i++){
            Sexo sexo = this.getRandSexo();
            String nombre = this.getRandNombre(sexo);
            String apellido1 = this.getRandApellido();
            String apellido2 = this.getRandApellido();
            this.add(
                new Persona(
                    nombre,
                    apellido1,
                    apellido2,
                    this.getRandDni(),
                    sexo,
                    this.locs.getRandomLocalidad(),
                    generateEmail(nombre, apellido1, apellido2))
```

```
            );
        }
    }

}
```

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.

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.

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 {
        Localidades locs = new Localidades();
        Localidades locs2 = new Localidades();
        locs.load();
        locs2.load();
        if (locs.equals(locs2)) {
            assertTrue(true);
```

```
        } else {
            assertTrue(false);
        }
    }

}
```

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:

```
package com.iesvdc.acceso.modelos;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlElement;

@XmlRootElement(name = "personas")
@XmlAccessorType (XmlAccessType.FIELD)
public class Personas {

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

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".

Marshaller XML

```java
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

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

/**
 * Ejemplo de Marshaller de personas.
 * Versión XML.
 */
public class MarshallerXML
{
    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 {
            jaxbContext = JAXBContext.newInstance(lista.getClass());
            Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
            jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
            jaxbMarshaller.marshal(lista, new File("personas.xml"));

        } catch (JAXBException e) {
            e.printStackTrace();
        }
```

```
    }
}
```

Unmarshaller XML

```
package com.iesvdc.acceso;

import java.io.File;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;

import com.iesvdc.acceso.modelos.Personas;

/**
 * Ejemplo de unmarshaller de personas.
 * Versión XML.
 */
public class UnMarshallerXML
{
    public static void main( String[] args )
    {
        Personas lista = new Personas();
        JAXBContext jaxbContext;

        try {

            jaxbContext = JAXBContext.newInstance(lista.getClass());
            Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
            Object objeto = jaxbUnmarshaller.unmarshal(new File("personas.xml"));
            lista = (Personas) objeto;
            System.out.println(lista.toString());

        } catch (JAXBException e) {
            e.printStackTrace();
        }

    }
}
```

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
            bootstrap al JAXBContext */
            System.setProperty(
                "javax.xml.bind.JAXBContextFactory",
                "org.eclipse.persistence.jaxb.JAXBContextFactory");

            jaxbContext = JAXBContext.newInstance(lista.getClass());

            Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
            jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

            //Para JSON
            jaxbMarshaller.setProperty(
                MarshallerProperties.MEDIA_TYPE, "application/json");
            jaxbMarshaller.setProperty(
                MarshallerProperties.JSON_INCLUDE_ROOT, true);

            jaxbMarshaller.marshal(lista, new File("personas.json"));

        } catch (JAXBException e) {
            e.printStackTrace();
        }

    }
}
```

Unmarshaller JSON

Para hacer el proceso inverso, el unmarshalling, es exactamente igual pero usando el objeto "Unmarshaller":

```
package com.iesvdc.acceso;

import java.io.File;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;
import org.eclipse.persistence.jaxb.UnmarshallerProperties;

import com.iesvdc.acceso.modelos.Personas;

/**
```

```
 * Ejemplo de unmarshaller de personas.
 * Versión JSON
 */
public class UnMarshallerJSON
{
    public static void main( String[] args )
    {
        Personas lista = new Personas();

        JAXBContext jaxbContext;

        try {

            System.setProperty(
                "javax.xml.bind.context.factory",
                "org.eclipse.persistence.jaxb.JAXBContextFactory");

            jaxbContext = JAXBContext.newInstance(lista.getClass());

            Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
            // Para JSON
            jaxbUnmarshaller.setProperty(
                UnmarshallerProperties.MEDIA_TYPE, "application/json");
            jaxbUnmarshaller.setProperty(
                UnmarshallerProperties.JSON_INCLUDE_ROOT, true);

            Object objeto = jaxbUnmarshaller.unmarshal(
                new File("personas.json"));
            lista = (Personas) objeto;
            System.out.println(lista.toString());

        } catch (JAXBException e) {
            e.printStackTrace();
        }
    }
}
```

Bibliografía

- http://juangualberto.github.io/acceso/tema01/gestin_de_ficheros_y_directorios_con_java.html
- Lista de todos los dominios de correo