

PROGRAMACIÓN en JAVA

Creado por: Iñigo Chueca



Iñigo Chueca López



2025-04-07

Índice

1. Conceptos Básicos del Lenguaje JAVA

- 1.1. Introducción Conceptos básicos
- 1.2. Instalación de JAVA
- 1.3. Estructura de un programa Java
- 1.4. Ejemplo Creación de un Programa Básico
- 1.5. Actividad 1 - Creación de una aplicación básica
- 1.6. Ejemplos Prácticos con Solución
- 1.7. Tipos de Datos variables y expresiones
- 1.8. Ejemplos de Operadores
- 1.9. Actividad 2 - Operadores
- 1.10. Actividad de Repaso
- 1.11. Actividad 22 -Tipos de Datos y Operadores
- 1.12. Ejercicios Adicionales
- 1.13. Entrada Salida de información por pantalla
- 1.14. Actividad 2-3
- 1.15. Ejercicios Adicionales
- 1.16. Estructuras de Control Secuencial y Alternativa
- 1.17. Actividad 3 - Alternativas
- 1.18. Estructuras de Control Repetitivas
- 1.19. Actividad 4 - Repetitivas
- 1.20. Ejemplos Prácticos de Uso de Repetitivas
- 1.21. Ejercicios Adicionales
- 1.22. 2 - Repetitivas y Alternativas
- 1.23. Ejercicio Adicional
- 1.24. Ejercicio Adicional-II
- 1.25. Programación Modular
- 1.26. Actividad 5 - Programación Modular
- 1.27. Estructuras de Control Control de Excepciones

2. Programación Orientada a Objetos con Java

- 2.1. Desarrollo de Clases Propiedades
- 2.2. Actividad 6 - Propiedades
- 2.3. Desarrollo de Clases Métodos
- 2.4. Actividad 7 - Métodos
- 2.5. Constructores Métodos de Acceso y Otros Métodos Habituales
- 2.6. Actividad 8 - Constructores
- 2.7. Actividad 9 - Creación de Clases
- 2.8. Algunas clases útiles de Java
- 2.9. Actividad 10 - String Date y GregorianCalendar
- 2.10. Actividad 11 - Formateadores
- 2.11. Herencia
- 2.12. Actividad 12 - Herencia
- 2.13. Interfaces
- 2.14. Actividad 13 - Interfaces
- 2.15. Testeando nuestra aplicación - TDD Test Driven Design
- 2.16. Actividad Adicional
- 2.17. Github Classroom

3. Matrices y Colecciones en JAVA

- 3.1. Introducción a las Matrices
- 3.2. Operaciones Básicas con Matrices Desordenadas Inserción eliminación búsqueda y listado
- 3.3. Actividad 14 - Matrices desordenadas
- 3.4. Actividad 15 - Operaciones con matrices desordenadas
- 3.5. API de Colecciones
- 3.6. Interfaces Collection y List e implementaciones Vector ArrayList
- 3.7. Actividad 16 - Uso de Colecciones
- 3.8. Operaciones Básicas con Matrices Colecciones Ordenadas
- 3.9. Actividad 17 - Matrices Ordenadas Búsqueda
- 3.10. Actividad 17-2 - Matrices Ordenadas Inserción y Eliminación
- 3.11. Ejemplo Con Expresión Lambda Avanzado
- 3.12. Ordenación

- 3.13. Actividad 22 - Ordenación de matrices
- 3.14. Ejemplo Examen
- 3.15. Matrices Multidimensionales Matrices Anidadas y clase Arrays
- 3.16. Clase Arrays Ordenación de Objetos Interfaces Comparable Comparator y Expresiones Lambda
- 3.17. Interfaz Set y SortedSet e implementación con clases HashSet y TreeSet
- 3.18. Interfaz Map y SortedMap e implementación con clases HashMap y TreeMap
- 3.19. Actividad 23 - Colecciones
- 3.20. Introducción a Streams Java 8
- 3.21. Actividad 24 - Streams

4. Corrientes de Datos

- 4.1. Introducción
- 4.2. Corrientes de Datos con Ficheros FileStreams Clases File y JFileChooser
- 4.3. Corrientes de Datos con Ficheros FileStreams DataStreams
- 4.4. Corrientes de Datos con Ficheros Buffers PrintWriter RandomAccess
- 4.5. Serialización de Objetos
- 4.6. Actividad 25 - Corrientes de Datos
- 4.7. Ficheros Comprimidos ZIP
- 4.8. Sockets corrientes a través de TCP IP Threads Runnable
- 4.9. Acceso a Internet conexión a API REST
- 4.10. Hacer una aplicación ejecutable

5. Interfaces de Usuario Swing

- 5.1. Introducción Creando la primera ventana
- 5.2. Añadiendo elementos controles
- 5.3. BorderLayout
- 5.4. FlowLayout GridLayout NullLayout
- 5.5. Propiedades Básicas de Componentes y Controles JLabel JTextField JFormattedTextField JButton
- 5.6. Creación de Interfaces Complejos
- 5.7. Actividad 26 - Creación de Interfaces Complejos
- 5.8. Look And Feel
- 5.9. Gestión de Eventos
- 5.10. Actividad 27 - Gestión de Eventos
- 5.11. Ejemplo Práctico Creación de un Formulario de Contactos
- 5.12. Creación de una Aplicación Distribuible
- 5.13. Actividad 28- I - Creación de un Formulario De Productos
- 5.14. Modificación de la Aplicación de Contactos
- 5.15. Actividad 28- II - Creación de un Formulario De Productos
- 5.16. Más Gestores de Distribución GridBagLayout
- 5.17. GridBagLayout Ejemplo Complejo
- 5.18. Actividad 29 - GridBagLayout
- 5.19. Más Gestores de Distribución CardLayout BoxLayout NullLayout
- 5.20. Listas Desplegables JComboBox
- 5.21. Listas JList
- 5.22. Deslizadores JSlider
- 5.23. Actividad 30 - Uso de JComboBox JList y JSlider
- 5.24. Paneles JTabbedPane JSplitPane
- 5.25. Tablas de Datos JTable
- 5.26. Actividad 31 - Uso de JTable
- 5.27. Árboles JTree
- 5.28. Actividad 32 - Creación Formulario Complejo

6. Acceso a Bases de Datos

- 6.1. Introducción
- 6.2. Ejecución de Consultas DDL y DML
- 6.3. Actividad 33 Creación de Tablas
- 6.4. Ejecución de Consultas de Selección
- 6.5. Actividad 34 - ResultSets
- 6.6. Sentencias Preparadas PreparedStatement
- 6.7. Actividad 35 - PreparedStatement
- 6.8. Actividad 36
- 6.9. ResultSets Navegables y Actualizables
- 6.10. Transacciones
- 6.11. Procedimientos Almacenados
- 6.12. Actividad ODS

7. Capa de Persistencia de JAVA JPA

- 7.1. Introducción a JPA
- 7.2. Primeros Pasos Maven e Hibernate
- 7.3. Primeros Pasos EclipseLink
- 7.4. EntityManager Acceso y modificación de objetos
- 7.5. Consultas Básicas JPQL
- 7.6. Actividad 37 - Conceptos Básicos JPA
- 7.7. Relaciones entre Entidades OneToMany y ManyToOne
- 7.8. Trabajo con Entidades Relacionadas
- 7.9. Actividad 38 - Creación de relaciones
- 7.10. Relaciones entre Entidades - ManyToMany
- 7.11. Relaciones entre Entidades - OneToOne

8. Proyecto TMDB

- 8.1. Librerías y Encriptación
- 8.2. Login y Registro de Usuarios - DAO y Servicio
- 8.3. Test DAO y Servicio
- 8.4. Login y Registro de Usuarios - Vistas
- 8.5. Testando una Aplicación Gráfica
- 8.6. Formulario Principal
- 8.7. ActividadTMDB

9. Crear una Aplicación Ejecutable

- 9.1. Creación de una Aplicación Ejecutable

Apuntes del Curso de Java

Generado el 2025-04-07

1. Conceptos Básicos del Lenguaje JAVA

1.1. Introducción Conceptos básicos

Introducción a la Programación y Java

Resolución de Problemas mediante Ordenador

La resolución de problemas mediante el ordenador sigue generalmente estos pasos:

1. Definición o **análisis del problema**
2. **Diseño de un algoritmo** que resuelva el problema
3. **Adaptación** del algoritmo a un lenguaje de programación (programa)
4. **Ejecución y validación** del programa
5. **Mantenimiento y optimización** (paso adicional importante)

Algoritmo

Un **algoritmo** es una secuencia ordenada de pasos, sin ambigüedades, que conducen a la solución de un problema dado. Se expresa en lenguaje natural, por ejemplo, en castellano. Características esenciales de un algoritmo:

1. **Preciso**: Indica el orden exacto de realización de cada paso.
2. **Definido**: Proporciona los mismos resultados si se siguen los mismos pasos con los mismos datos de entrada.
3. **Finito**: Tiene un número limitado de pasos y termina en algún momento.
4. **Eficiente**: Utiliza de manera óptima los recursos disponibles.

Programa

Un **programa** es la traducción de un algoritmo a un lenguaje de programación específico. Este proceso se denomina **codificación**. Aunque un algoritmo debería ser independiente del lenguaje de programación, en la práctica, su representación puede variar según el lenguaje elegido. La elección del lenguaje de programación depende de varios factores:

- Naturaleza del problema a resolver
- Eficiencia requerida
- Plataforma de destino
- Experiencia del equipo de desarrollo
- Requisitos del proyecto

Lenguaje de Programación

Un **lenguaje de programación** es un conjunto de reglas sintácticas y semánticas que permiten escribir programas. Ejemplos incluyen Python, Java, C++, JavaScript, entre otros.

Tipos de Lenguajes de Programación

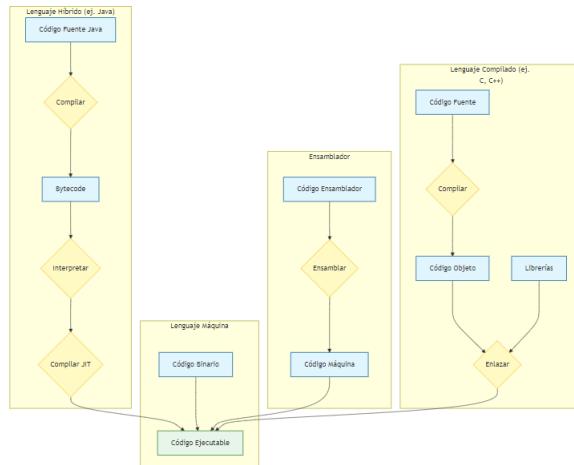
1. **Lenguaje máquina**: Compuesto por instrucciones binarias directamente ejecutables por el procesador.
2. **Lenguaje de bajo nivel** (ensamblador): Utiliza códigos nemotécnicos que representan instrucciones de máquina. Es específico para cada arquitectura de procesador.
3. **Lenguaje de alto nivel**: Más cercano al lenguaje humano, independiente de la máquina y portable entre diferentes arquitecturas. Ejemplos: Python, Java, C++.

Proceso de Traducción

Para que un programa escrito en un lenguaje de alto nivel pueda ser ejecutado por un ordenador, debe ser traducido a lenguaje máquina. Existen tres principales métodos:

1. **Ensambladores**: Traducen lenguaje de bajo nivel a lenguaje máquina.
2. **Compiladores**: Traducen lenguaje de alto nivel a lenguaje máquina, generando un archivo ejecutable. Ejemplos: C, C++.
3. **Intérpretes**: Traducen y ejecutan el código línea por línea. Ejemplos: Python, Ruby.
4. **Enfoque híbrido**: Algunos lenguajes, como Java, utilizan un enfoque de compilación a bytecode e interpretación posterior.

En función del proceso de traducción tendremos que realizar una serie de tareas u otras:



Tipos de Programación

Programación Estructurada

La programación estructurada es un paradigma que busca mejorar la claridad, calidad y tiempo de desarrollo utilizando únicamente tres estructuras de control:

1. **Secuencial:** Ejecución de instrucciones en orden.
2. **Selectiva (Condicional):** Ejecución de instrucciones basada en condiciones.
3. **Repetitiva (Bucles):** Ejecución repetida de instrucciones.

Programación Modular

La programación modular consiste en dividir un programa complejo en subprogramas más pequeños y manejables llamados **módulos**. Cada módulo realiza una tarea específica y puede ser desarrollado, probado y mantenido de forma independiente.

Programación Orientada a Objetos (POO)

La POO es un paradigma que organiza el código en objetos, que son instancias de clases. Los conceptos fundamentales de la POO incluyen:

1. **Clases:** Plantillas que definen las propiedades y comportamientos de los objetos.
2. **Objetos:** Instancias de clases que representan entidades del mundo real o conceptual.
3. **Encapsulamiento:** Ocultamiento de los detalles internos de un objeto.
4. **Herencia:** Mecanismo para crear nuevas clases basadas en clases existentes.
5. **Polimorfismo:** Capacidad de objetos de diferentes clases de responder al mismo mensaje de diferentes maneras.

Java

Java es un lenguaje de programación versátil y poderoso, desarrollado originalmente por Sun Microsystems en 1995 y ahora propiedad de Oracle Corporation.

Características principales de Java:

1. **Portabilidad:** "Write Once, Run Anywhere" (WORA)
2. **Orientado a objetos**
3. **Seguro**
4. **Robusto**
5. **Multihilo**
6. **Independiente de la plataforma**
7. **Alto rendimiento**

Evolución de Java a lo largo de los años:

Versión	Año	Características Principales
Java 1.0	1996	Primeras versiones públicas. Applets, AWT (GUI básica), JDK 1.0.
Java 1.1	1997	Inner classes, JDBC (conexión a bases de datos), RMI (llamadas remotas), Reflection API.
Java 1.2	1998	Llamada "Java 2". Colecciones (List, Set, Map), JIT Compiler, Swing (GUI mejorada), palabra clave <code>strictfp</code> .
Java 1.3	2000	Mejoras de rendimiento (HotSpot JVM), JNDI (acceso a directorios), JavaSound.
Java 1.4	2002	<code>assert</code> , Expresiones regulares (<code>java.util.regex</code>), NIO (I/O no bloqueante), API Logging (<code>java.util.logging</code>).

Versión	Año	Características Principales
Java 5	2004	Gran actualización: Genéricos, Enums, Autoboxing/Unboxing, Anotaciones, varargs, for-each, java.util.concurrent.
Java 6	2006	Soporte para scripting (JavaScript), JDBC 4.0, JAX-WS (web services), Compiler API.
Java 7	2011	Try-with-resources, Strings en switch, NIO.2 (Path, Files), Fork/Join Framework, Diamante (<>).
Java 8 (LTS)	2014	Revolución funcional: Lambdas, Stream API, Optional, métodos default en interfaces, Date/Time API (java.time).
Java 9	2017	Sistema modular (Project Jigsaw), JShell (REPL), HTTP/2 Client, Factory Methods para colecciones (List.of(), etc.).
Java 10	2018	Inferencia de tipos locales (var), mejoras en GC (Garbage Collector).
Java 11 (LTS)	2018	HTTP Client estándar, var en lambdas, elimina módulos Java EE y CORBA.
Java 12	2019	Switch expressions (preview), Shenandoah GC (recolección de basura de bajo tiempo de pausa).
Java 13	2019	Text blocks (""" . . . """), Dynamic CDS (mejora de rendimiento).
Java 14	2020	Records (clases inmutables, preview), instanceof con pattern matching, NullPointerExceptions descriptivas.
Java 15	2020	Sealed Classes (clases selladas, preview), Text blocks estándar.
Java 16	2021	Records estándar, Pattern Matching para instanceof, Stream.toList().
Java 17 (LTS)	2021	Sealed Classes estándar, Pattern Matching for Switch (preview), eliminación de Applets y RMI.
Java 18	2022	Simple Web Server, UTF-8 por defecto, @snippet para documentación.
Java 19	2022	Virtual Threads (hilos ligeros, preview), Structured Concurrency (preview).
Java 20	2023	Scoped Values (conurrencia), Record Patterns (desestructuración de objetos).
Java 21 (LTS)	2023	Virtual Threads (final), Pattern Matching for Switch (final), Generational ZGC (mejoras en GC).

Notas:

- **LTS:** Versiones con soporte a largo plazo (Long-Term Support), como Java 8, 11, 17 y 21.
- **Preview Features:** Algunas funcionalidades se lanzan en modo "preview" para recibir feedback antes de ser estándar.
- **Cambio de versión:** A partir de Java 10, las versiones se lanzan cada 6 meses, pero solo las LTS tienen soporte extendido.

Desarrollo con Java

Java permite desarrollar una amplia gama de aplicaciones:

1. Aplicaciones de escritorio
2. Aplicaciones web
3. Aplicaciones empresariales
4. Aplicaciones móviles (Android)
5. Sistemas embebidos
6. Internet de las cosas (IoT)

Para comenzar a programar en Java, es necesario familiarizarse con su sintaxis básica, que incluye:

- **Estructura** de un programa Java
- **Comentarios**
- **Variables y tipos de datos**
- **Operadores y expresiones**
- **Estructuras de control** (if, switch, for, while)
- **Métodos y clases**

1.2. Instalación de JAVA

Como ya hemos visto el lenguaje de programación Java fue desarrollado con la idea de disponer de un lenguaje de programación que permitiera crear aplicaciones que pudieran ejecutarse en diferentes sistemas operativos. Para ello, la solución fue **partir de un programa escrito en JAVA que se compila a un lenguaje intermedio**, denominado **bytecode**. La idea es que **dicho lenguaje intermedio es independiente del sistema operativo**. Proporcionando un intérprete para cada uno de los sistemas operativos disponibles se consigue que **el mismo programa pueda ser ejecutado en diferentes sistemas operativos** (por ejemplo, en Windows, Mac, en UNIX / Linux, Solaris, etc.). Este intérprete es lo que se denomina **JVM** (Java Virtual Machine) y es el encargado de **traducir el bytecode al código máquina adecuado al S.O. en el que se ejecuta** y, junto con las librerías, **ejecutar el código** correspondiente. Esta idea implica que **necesitamos instalar ese intérprete** (junto con una serie de librerías y programas adicionales) en cada equipo y que debemos instalar la versión adecuada al sistema operativo que estamos usando. A esta aplicación junto con sus elementos asociados es a lo que se denomina el **JRE** (Java Runtime Engine). Básicamente el **JRE** incluye:

- La **aplicación** que permite ejecutar el bytecode (java, javaw) junto con otras aplicaciones asociadas

- Las **librerías** donde están las clases estándar de Java (lo que se llama **Java API**). Aquí están las clases que usan los desarrolladores para crear las aplicaciones.

Si lo que queremos es **desarrollar aplicaciones** necesitamos lo que se llama el **JDK** (Java Developers Kit) que, incluye, además de las librerías, código para **compilar** los programas (`javac`), **ejecutarlos** (`java/javaw`), **depurarlos** (`javap`), generar **documentación** (`javadoc`), etc.

Versiónes

El 16 de Abril de 2019 cambió sustancialmente la licencia de Java de modo que la última versión gratuita de Java JDK es la versión 8. A partir de esta versión hay dos opciones:

- Emplear la versión open source de java denominada **Oracle Open JDK**. No dispone de soporte
- Emplear la versión comercial del JDK Esta versión es **gratuita para uso personal** pero de pago por licencias para uso comercial. Dispone de soporte oficial por parte de Oracle

Instalación

Si queremos descargar la última versión estable de **OpenJDK** desde <https://adoptopenjdk.net/> que nos proporciona versiones ya compiladas para los diferentes sistemas operativos:

The screenshot shows the AdoptOpenJDK website's download section. At the top, it says "Prebuilt OpenJDK Binaries for Free!". Below that, a note states: "Java™ is the world's leading programming language and platform. AdoptOpenJDK uses infrastructure, build and test scripts to produce prebuilt binaries from OpenJDK™ class libraries and a choice of either OpenJDK or the Eclipse OpenJ9 VM. All AdoptOpenJDK binaries and scripts are open source licensed and available for free." There are two main sections for selecting a version: "1. Elija una versión" (Choose a version) and "2. Elija una JVM". Under "1. Elija una versión", there are radio buttons for "OpenJDK 8 (LTS)", "OpenJDK 11 (LTS)", and "OpenJDK 15 (Latest)". Under "2. Elija una JVM", there are radio buttons for "HotSpot" (selected) and "OpenJ9". A large blue button labeled "Último lanzamiento" (Latest Release) is prominently displayed, with the text "jdk-15.0.1+9" underneath. Below this button are links for "Otras plataformas" (Other platforms) and "Archivo de lanzamientos y compilaciones nocturnas" (Nightly build archive). At the bottom of the page, a note says "AdoptOpenJDK now also distributes OpenJDK upstream builds! (Built by Red Hat)".

y lo instalamos con las opciones por defecto. Si queremos emplear la versión oficial (para uso personal) deberemos ir a la página <https://www.oracle.com/java/technologies/downloads/>:

The screenshot shows the Oracle Java Downloads page. At the top, there's a navigation bar with links for Oracle, Products, Industries, Resources, Customers, Partners, Developers, Events, a search bar, and buttons for View Accounts and Contact Sales. The main header is "Java Downloads". Below it, there's a section for "Java 18 and Java 17 available now". It mentions that Java 17 LTS is the latest long-term support release for the Java SE platform, and that Java 18 and Java 17 binaries are free to use in production and free to redistribute, at no cost, under the Oracle No-Fee Terms and Conditions. It also notes that Java 18 will receive updates until September 2022, and Java 17 until at least September 2024. There are buttons for "Java 18" and "Java 17". Below this, there's a section for "Java SE Development Kit 18.0.2.1 downloads". It thanks users for downloading the Java Platform, Standard Edition Development Kit (JDK). It explains that the JDK is a development environment for building applications and components using the Java programming language. It also states that the JDK includes tools for developing and testing programs written in the Java programming language and running on the Java platform. At the bottom of this section, there are links for "Linux", "macOS", and "Windows".

Es posible tener diferentes versiones de Java instaladas simultáneamente. De hecho para el desarrollo de aplicaciones de escritorio desde Oracle recomiendan emplear la versión 8 de Java dado que el JRE suele venir incluido en los Sistemas Operativos modernos.

Instalación de NetBeans

Para las clases vamos a emplear el IDE (Entorno de Desarrollo Integrado) NetBeans que podéis instalar desde <https://netbeans.apache.org/download/index.html>

The screenshot shows the Apache NetBeans download page. At the top, there's a navigation bar with links for Community, Participate, Blog, Get Help, Plugins, and Download. Below the header, a large blue banner for 'Apache NetBeans 12.1' is displayed, with a 'Find out more' button. Underneath, a section titled 'Apache NetBeans Releases' is shown. It highlights the 'Latest release' (Apache NetBeans 12.1) and includes a note about the annual May/June release being a long-term support (LTS) release. Below this, two sections are listed: 'Apache NetBeans 12 feature update 1 (NB 12.1)' and 'Apache NetBeans 12 LTS (NB 12.0)'. Each section has a 'Features' and a 'Download' button.

Seleccionamos la opción LTS (Long Term Support) dado que es la versión estable. Seleccionamos la versión apropiada para nuestro S.O. dentro de Installers. Para **Windows de 64 bits**:

- Installers:
- [Apache-NetBeans-12.0-bin-windows-x64.exe \(SHA-512, PGP ASC\)](#)

Una vez descargado lo instalamos sin más.

1.3. Estructura de un programa Java

Estructura básica de un programa

Para crear un programa en Java, **debemos definir las clases** que van a formar parte de la aplicación. Estas clases pueden **definirse en uno o varios ficheros de texto que tienen como extensión .java** (en un mismo fichero pueden definirse varias clases diferentes). En cada fichero java pueden aparecer, entre otros, los siguientes elementos :

- **declaración del paquete** en el que queremos incluir la clase.
- sentencias `import` para **importar paquetes y/o clases que queremos emplear** en nuestro programa
- **declaración de interfaces**
- **declaración de clases**

Las declaraciones de **clases e interfaces** a su vez pueden contener:

- **declaración de variables/propiedades/campos** de instancia y de clase
- **declaración e implementación de los métodos** de las clases e interfaces

por último, en cada **implementación de los métodos** pueden aparecer :

- **declaración de variables locales**
- **instrucciones válidas en Java** que realicen las tareas asociadas al método

La compilación de los ficheros java (por ejemplo mediante el compilador `javac`) generará el **bytecode** asociado a **cada una de las clases e interfaces** definidas en el fichero y que tendrán extensión `.class`. Dado que en Java **todo son objetos, ejecutar** un programa implica **crear una instancia de una clase y ejecutar un método específico** de la misma. Para aplicaciones independientes, la clase deberá tener definido un método `main` para ser ejecutable. El **intérprete (el programa java)** **instanciará la clase y ejecutará el código** incluido en el método `main` de la misma. Dentro de ese código es donde se implementará la **funcionalidad del programa**. Por ejemplo, para una clase que va a ser ejecutada como una aplicación tendríamos el siguiente esquema :

```
// Declaración de paquete (opcional)
package com.ejemplo.miproyecto;

// Importaciones (opcionales)
import java.util.ArrayList;
import java.util.List;

// Declaración de la clase
public class MiClase {

    // Atributos (variables de instancia)
    private String nombre;
    private int edad;
    // Atributo estático y constante
    public static final String CONSTANTE = "Valor constante";

    // Constructor
    public MiClase(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

```

// Métodos
public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public int getEdad() {
    return edad;
}

public void setEdad(int edad) {
    this.edad = edad;
}

public void imprimirInformacion() {
    System.out.println("Nombre: " + nombre + ", Edad: " + edad);
}

// Método estático
public static void metodoEstatico() {
    System.out.println("Este es un método estático");
}

// Clase interna (opcional)
private class ClaseInterna {
    // Contenido de la clase interna
}

// Método principal (opcional)
public static void main(String[] args) {
    System.out.println("Si la clase es ejecutable, aquí pondremos el código");
}
}

```

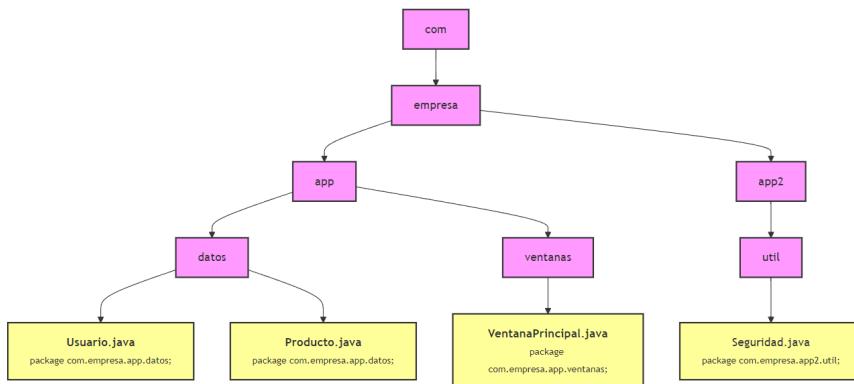
Como se ve el método main debe ser estático (java llamará a MiClase.main). Además, el método tiene un parámetro de tipo String[] (el nombre args puede ser cualquiera) que representa una matriz de cadenas (veremos las matrices en Java más adelante pero baste decir por ahora que es una colección ordenada de elementos). Esta matriz almacenará los parámetros pasados al programa cuando se ejecuta: java MiClase param1 param2... En nuestro caso, de momento, no vamos a emplear dicha información.

Paquetes en Java

Los paquetes en Java son **una forma de organizar y agrupar clases y interfaces relacionadas**. Funcionan de manera similar a las carpetas en un sistema de archivos, ayudando a estructurar y modularizar el código (de hecho, en el sistema de archivos los paquetes están asociados a directorios). Los paquetes ofrecen varios **beneficios** importantes:

- **Organización del código:** Permiten agrupar clases relacionadas, mejorando la estructura y mantenibilidad del proyecto.
- **Evitar conflictos de nombres:** Clases con el mismo nombre pueden coexistir en diferentes paquetes sin conflicto.
- **Control de acceso:** Ofrecen un nivel adicional de encapsulación y control de acceso a clases y miembros.
- **Facilitan la reutilización:** Los paquetes bien diseñados pueden ser fácilmente reutilizados en diferentes proyectos.

En el siguiente esquema se ve la estructura de directorios de un proyecto y cómo afecta a los nombres de los paquetes:



Si desde el código de `Usuario.java` queremos acceder a la clase `Producto` podríamos indicar simplemente `Producto` (dado que **están en el mismo paquete/carpeta**), pero si queremos acceder a la clase `VentanaPrincipal` deberíamos **indicar toda la ruta** (`com.empresa.app.ventanas.VentanaPrincipal`). Por tanto, para acceder a `Seguridad` utilizaríamos `com.empresa.app2.util.Seguridad`.

Esto implica que, **cada vez que, desde una clase, referenciamos otra que no está en el mismo paquete, debemos especificar el camino completo**. Para evitar tener que hacerlo cada vez podemos informar a Java de dónde se encuentra esa clase añadiendo una sentencia `import`. Por ejemplo, en la clase `Usuario`:

```
import com.empresa.app.ventanas.VentanaPrincipal; import com.empresa.app2.util.Seguridad;
```

A partir de este momento, en la clase podemos referenciar las otras clases simplemente por su nombre sin incluir toda la ruta.

Es posible emplear el * para incluir **todas las clases de un paquete**. Por ejemplo, si ponemos en la clase VentanaPrincipal:

```
import com.empresa.app.datos.*  
  
podríamos acceder a Usuario y Producto directamente por sus nombres.
```

Ejemplo

Para poder crear una aplicación con Java, **tan sólo necesitamos el JDK y un editor de texto** sin formato (por ejemplo el Notepad). Como hemos visto necesitamos que se puedan ejecutar los comandos `javac` y `java` que se encuentran en el **directorio de instalación de Java** que, por defecto es `C:\Program Files\Java\versiónJDK`. Vamos a dar los siguientes pasos:

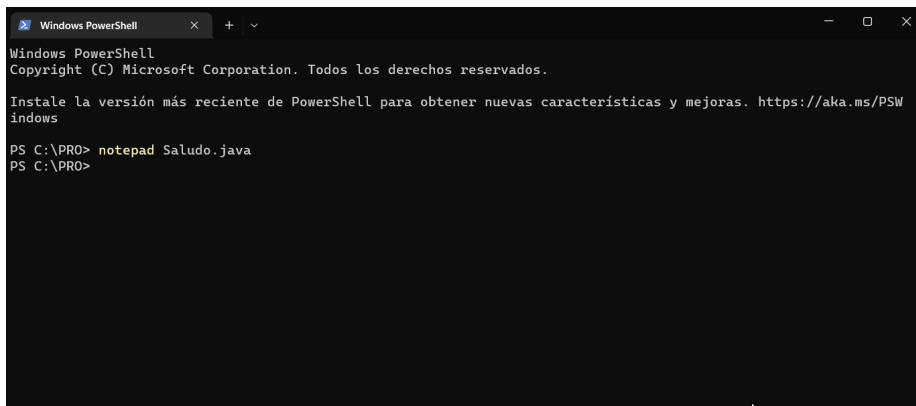
- **Crear un directorio de trabajo:** Es conveniente tener un directorio donde vamos a dejar nuestros proyectos. En este caso vamos a dejarlo en `C:\PRO`
- **Abrir un terminal:** Para ello nos situamos en la carpeta recién creada, pulsamos **botón derecho** y ejecutamos `Abrir en Terminal` (u otra opción similar).
- **Abrir el bloc de notas** y crear el fichero . Desde el terminal ejecutamos:

```
notepad Saludo.java
```

- **Escribimos** el programa de saludo (recordad que la clase se llama `Saludo`). Es importante escribirlo tal y como se muestra en la imagen

```
public class Saludo {  
    public static void main(String[] args){  
        System.out.println("Hola desde JAVA!")  
    }  
}
```

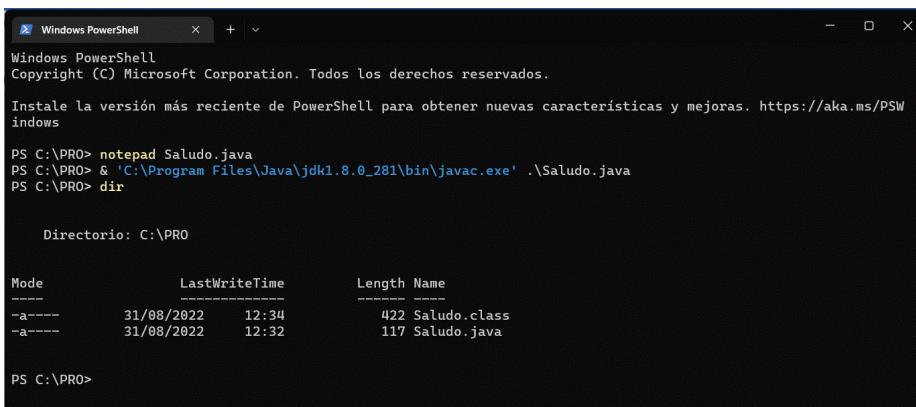
- **Compilamos**: Una vez guardado debemos compilarlo. La aplicación `javac` se encuentra en la carpeta bin dentro del directorio del JDK



```
Windows PowerShell  
Copyright (C) Microsoft Corporation. Todos los derechos reservados.  
Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras. https://aka.ms/PSW  
indows  
PS C:\PRO> notepad Saludo.java  
PS C:\PRO>
```

- El comando **no debe retornar nada**. Si muestra información es que existen errores de sintaxis en el código escrito (errores de **compilación**). En una aplicación podemos distinguir **tres tipos de errores**:
- **errores de compilación**: Son debidos a que no se cumplen con las especificaciones gramaticales del lenguaje de programación (las instrucciones no acaban en ;, paréntesis o llaves mal cerrados, poner instrucciones en lugares no permitidos...). Se **detectan cuando se compila el programa** (en un IDE se detectan al momento) e impiden que se genere un programa ejecutable. Son los más fáciles de detectar y corregir.
- **errores en tiempo de ejecución (excepciones)**: El programa **se compila correctamente pero cuando se ejecuta y en determinadas circunstancias se pueden provocar errores** que impidan la ejecución del mismo. **Ejemplos**: operaciones no permitidas (divisiones por cero), errores en el acceso a recursos externos... Son **más difíciles de detectar** (en ocasiones sólo se producen en determinadas circunstancias) aunque disponemos de herramientas que nos permiten controlarlos desde el propio código.
- **errores de la lógica del programa**: El programa **compila y se ejecuta correctamente pero los resultados no son los deseados**. Son los **más complicados de solucionar**. Para detectarlos es necesario definir una batería de tests que comprueben la correcta ejecución de la aplicación.

- Con el comando `dir` podemos comprobar que se ha creado el fichero `Saludo.class` (que es donde se encuentra el **bytecode**):



```
Windows PowerShell  
Copyright (C) Microsoft Corporation. Todos los derechos reservados.  
Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras. https://aka.ms/PSW  
indows  
PS C:\PRO> notepad Saludo.java  
PS C:\PRO> & 'C:\Program Files\Java\jdk1.8.0_281\bin\javac.exe' .\Saludo.java  
PS C:\PRO> dir  
  
Directorio: C:\PRO  
  
Mode LastWriteTime Length Name  
---- ----- ---- --  
-a--- 31/08/2022 12:34 422 Saludo.class  
-a--- 31/08/2022 12:32 117 Saludo.java  
  
PS C:\PRO>
```

- Ejecutamos la aplicación con el comando `java`:

```

Windows PowerShell
PS C:\PRO> notepad Saludo.java
PS C:\PRO> & 'C:\Program Files\Java\jdk1.8.0_281\bin\javac.exe' .\Saludo.java
PS C:\PRO> dir

Directorio: C:\PRO

Mode LastWriteTime Length Name
---- ----- ---- -
-a--- 31/08/2022 12:43 422 Saludo.class
-a--- 31/08/2022 12:32 117 Saludo.java

PS C:\PRO>

```

Es importante ver que la aplicación `java.exe` espera el nombre de la clase que tiene el método `main` por lo que no hay que poner extensión. Como se puede ver, la salida se muestra directamente por consola.

- En el caso de tener **errores de sintaxis**, el compilador nos mostrará **información sobre dónde se ha detectado el error y de qué error se trata** para poder corregirlo. Por ejemplo, si **eliminamos** el ; detrás de la instrucción y compilamos de nuevo:

```

Windows PowerShell
PS C:\PRO> & 'C:\Program Files\Java\jdk1.8.0_281\bin\javac.exe' Saludo
Hola desde JAVA!
PS C:\PRO> dir

Directorio: C:\PRO

Mode LastWriteTime Length Name
---- ----- ---- -
-a--- 31/08/2022 12:43 422 Saludo.class
-a--- 31/08/2022 12:32 117 Saludo.java

PS C:\PRO>

```

Como ya veremos, aparte de los errores de compilación también se pueden producir errores cuando se ejecuta el programa (**errores de ejecución**). En este caso, **por defecto, la aplicación se parará y se mostrará por consola información sobre el error**.

En el ejemplo **no hemos metido la aplicación en un paquete** (que es lo que se recomienda desde Java). Si por ejemplo quisieramos que la **aplicación estuviera** en el paquete `zabalburu` los pasos a dar serían los siguientes:

- **Crear una carpeta** `zabalburu` y poner en ella el fichero `java`. Podemos hacerlo desde la línea de comandos:

```

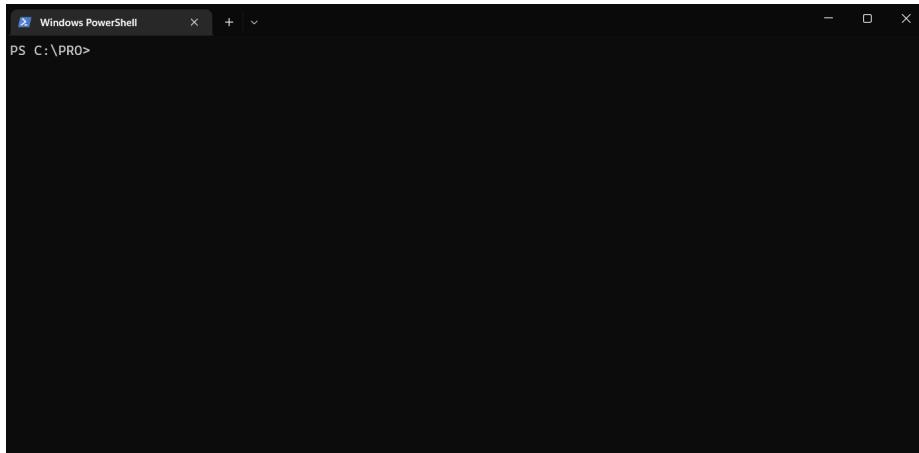
Windows PowerShell
PS C:\PRO> mkdir zabalburu

Directorio: C:\PRO

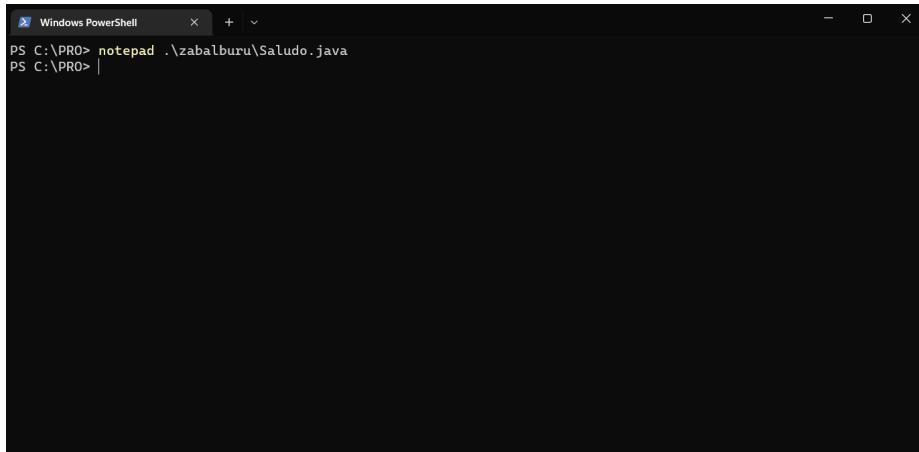
Mode LastWriteTime Length Name
---- ----- ---- -
d----

```

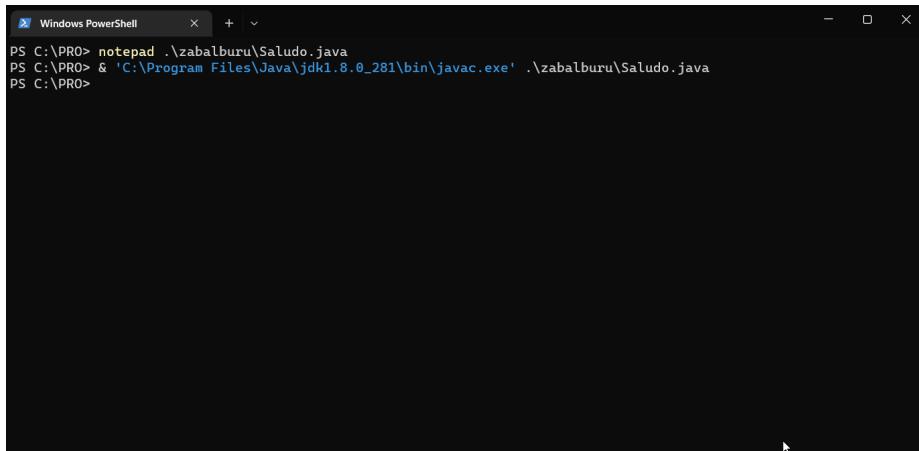
- **Añadir la instrucción que define el paquete en el fichero** (y corregir el error anterior):



- **Compilar el programa**: Hay que hacerlo **desde el directorio actual**:



- **Y ejecutarlo**. Ahora hay que recordar que, dado que la clase está en un paquete, su nombre ahora es `paquete.Clase`:

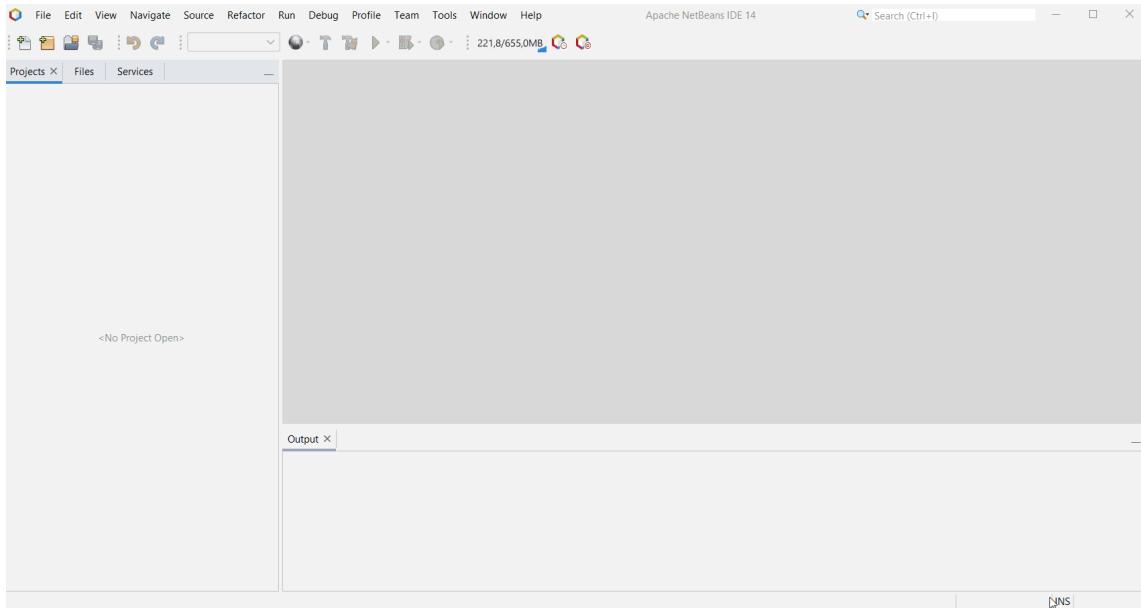


Ejemplo con un IDE (NetBeans)

Una vez iniciado el IDE deberemos seleccionar la **opción de crear un nuevo Proyecto Java** seleccionando la opción `File → New Project`. Podemos crear un proyecto Java con tres herramientas distintas que nos van a facilitar la tarea de compilar la aplicación:

- **Maven**: Como veremos más adelante, Maven nos facilita la utilización de librerías de terceras partes dentro de nuestras aplicaciones. Es lo que se denomina un **gestor de dependencias**. Veremos maven más en detalle este curso y el que viene.
- **Gradle**: Otro gestor de dependencias **similar a Maven**.
- **Ant**: Al no ser un gestor de dependencias **deberemos incluir manualmente las librerías que necesitemos** en nuestros proyectos.

Si queremos crear un proyecto Maven:

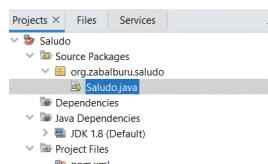


Como se puede ver, seleccionamos la opción de **Java with Maven** y, dentro de ella, queremos crear una aplicación Java (**Java Application**). En la siguiente pantalla deberemos indicar las **opciones** de nuestro proyecto:

- **Project Name**: El **nombre del proyecto** (NetBeans asignará este nombre automáticamente como nombre del archivo compilado y empaquetado -**Artifact Id**-)
- **Project Location**: La **ubicación** del mismo (dentro de él se creará una carpeta con el nombre del proyecto -**Project Folder**- donde se almacenan todos los ficheros necesarios)
- **Group Id**: Identifica el **grupo al que pertenece la aplicación**. Como ya se ha comentado se **recomienda que sea el nombre de dominio de la empresa al revés** (`org.zabaluru`).
- **Versión**: La **versión de la aplicación**. En principio `1.0-SNAPSHOT`. El `1.0` indica la versión de la aplicación. El **primer número indica la versión** y sólo debería incrementarse cuando hay **cambios sustanciales** en la aplicación. El **segundo es la revisión** (se incrementa cuando hay **pequeños cambios**). La palabra `SNAPSHOT` indica que la **aplicación está en desarrollo** (se puede emplear la palabra `RELEASE` para indicar que es una **versión definitiva**). En cualquier caso este sistema de versionado no es obligatorio y podemos poner en este campo lo que queramos.
- **Package**: El **paquete de la aplicación**. En principio la concatenación entre el **Group Id** y el **Project Name**. Recordemos que este paquete se convierte en una **estructura de directorios donde estarán las clases**

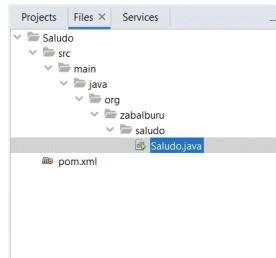
Como vemos una vez creado el proyecto, NetBeans nos crea automáticamente la clase `Saludo` con código de ejemplo. Disponemos de **tres vistas diferentes del mismo proyecto**:

1. **Vista Projects**: Nos muestra una visión lógica de la aplicación:



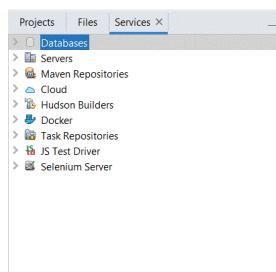
En esta vista vemos los siguientes elementos:

- **El proyecto** (`Saludo`)
- **El código fuente de nuestro proyecto** (las clases que escribimos). Se muestran dentro de los **paquetes** (carpetas) a los que corresponden y están dentro de la opción `Source Packages`
- **Las dependencias**: Aquí aparecerán las **librerías adicionales** que emplearemos en nuestra aplicación (por ejemplo para acceder a bases de datos)
- **Java Dependencies**: Aquí vemos el `JDK` que estamos empleando y **podemos ver todos los paquetes y clases del mismo** (incluso podemos ver el código de las mismas)
- **Project Files**: **Ficheros adicionales**. Entre ellos siempre aparecerá `pom.xml` que lo veremos en detalle cuando trabajemos con dependencias. Este fichero aparece **siempre en los proyectos maven**.
- **Vista Files**: Esta vista es similar a la anterior pero en ella vemos el **contenido real de la carpeta del proyecto**:



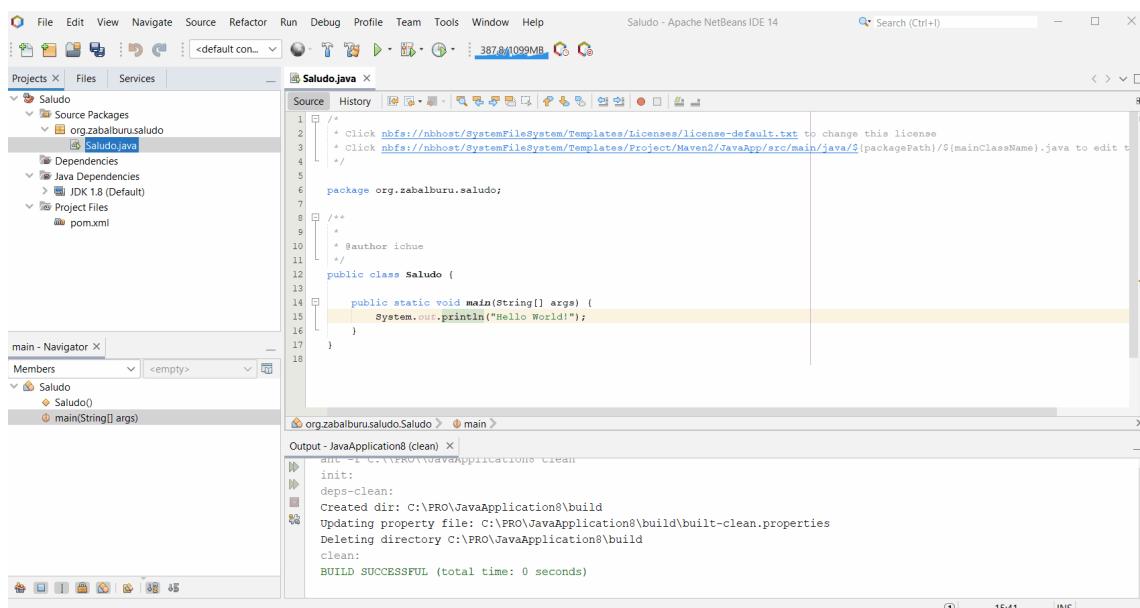
Como se puede ver, aquí aparecen las carpetas de nuestra aplicación. El **código fuente** se ubica en la carpeta `src/main/java` dentro de la carpeta del proyecto mientras que el fichero `pom.xml` se ubica en la propia carpeta del mismo. Cuando **compilemos** aparecerán **nuevas carpetas y ficheros** que sólo veremos desde aquí.

1. **Vista Services**: Esta vista nos permite **acceder a servicios externos** que nuestra aplicación pueda necesitar. Es independiente del proyecto actual:



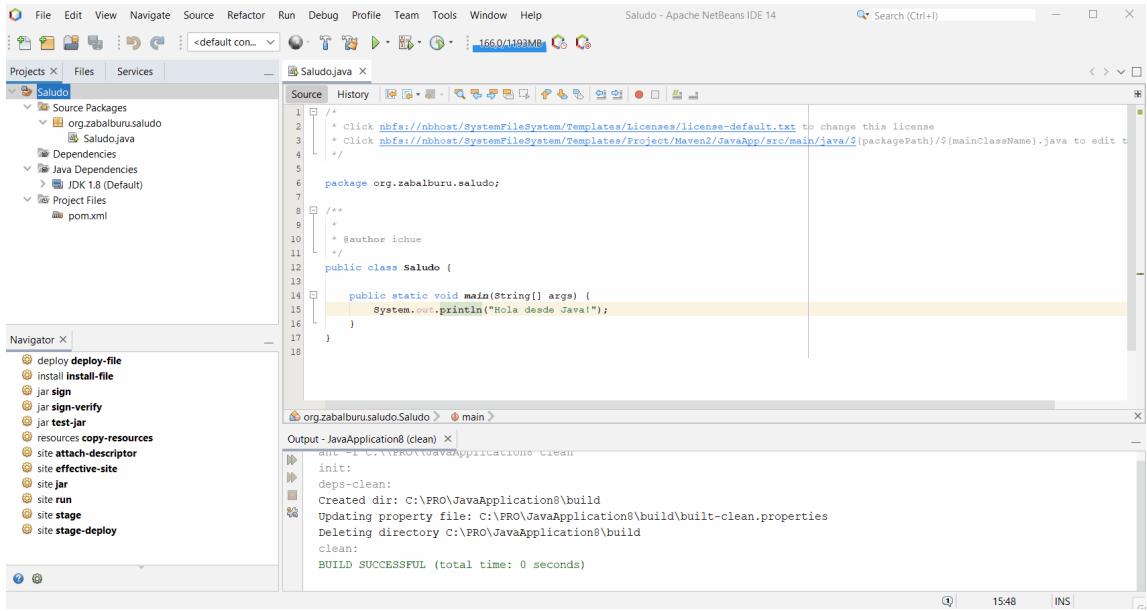
En este curso veremos las opciones **Databases** y **Maven Repositories**

Una ventaja que tienen los IDEs es que **detectan los errores según vamos escribiendo el código**:

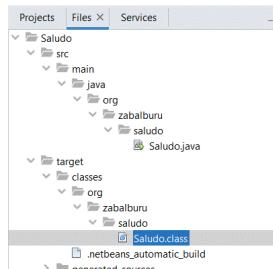


Como se puede ver, si una clase tiene errores de compilación se marca **inmediatamente con una línea roja**. Situándonos en ella nos aparece el **mensaje de error para que podamos corregirlo** (veremos más adelante que, en determinadas ocasiones, NetBeans nos sugiere cómo podemos corregir el error). Si un fichero tiene errores, al guardarlo se mostrará un ícono de error junto al mismo (y en toda la ruta para llegar a él). Para **ejecutar un proyecto** (primero lo compila y después lo ejecuta) disponemos de varias opciones:

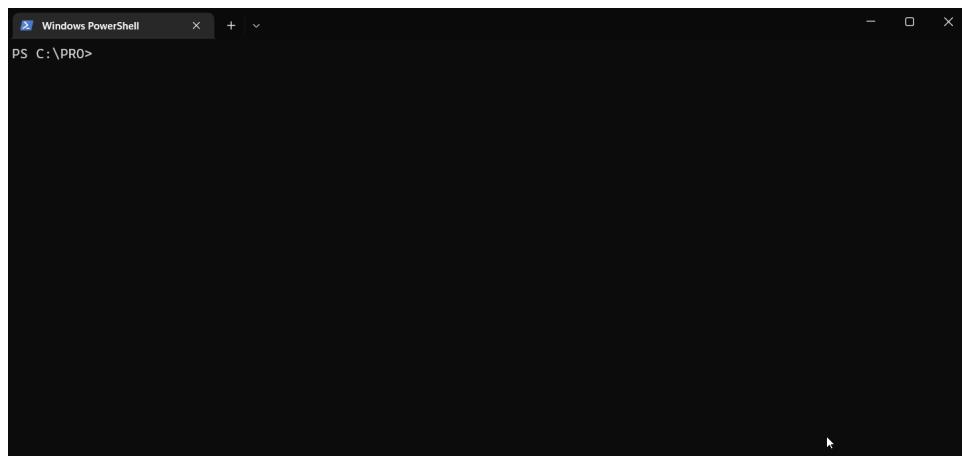
- Pulsar sobre el botón **Run Project** (F6). Esto ejecutará la **clase principal del proyecto** (que deberá ser una clase ejecutable, es decir que tenga el método `main`). Es importante tener en cuenta que **un proyecto puede tener múltiples clases ejecutables pero sólo una de ellas será la clase principal del mismo** (la que se ejecute al seleccionar esta opción). Es posible modificar cuál es la clase principal desde la categoría **Run** de las propiedades del proyecto que aparecen al seleccionar **Properties** en el **menú contextual** del mismo:



Como se puede ver en la pestaña Output, Maven compila el proyecto y luego lo ejecuta mostrando la salida por consola en dicha pestaña. Si vamos ahora a la vista de ficheros podremos ver cómo se ha creado la clase en otra carpeta dentro del proyecto:



De hecho podríamos ejecutar la aplicación desde dicha carpeta (target/classes):



- Si la clase ejecutable no es la clase principal podemos seleccionar la opción Run File (Shift + F6) desde el menú contextual de dicha clase o desde el menú Run (si la clase es la activa):

```

1 /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Project/Maven2/JavaApp/src/main/java/S{packageName}.java to edit t
4  */
5
6 package org.zabalburu.saludo;
7
8 /**
9  * 
10 * @author iuchue
11 */
12 public class Saludo {
13
14     public static void main(String[] args) {
15         System.out.println("Hola desde Java!");
16     }
17 }

```

Como se puede ver, el resultado es el mismo.

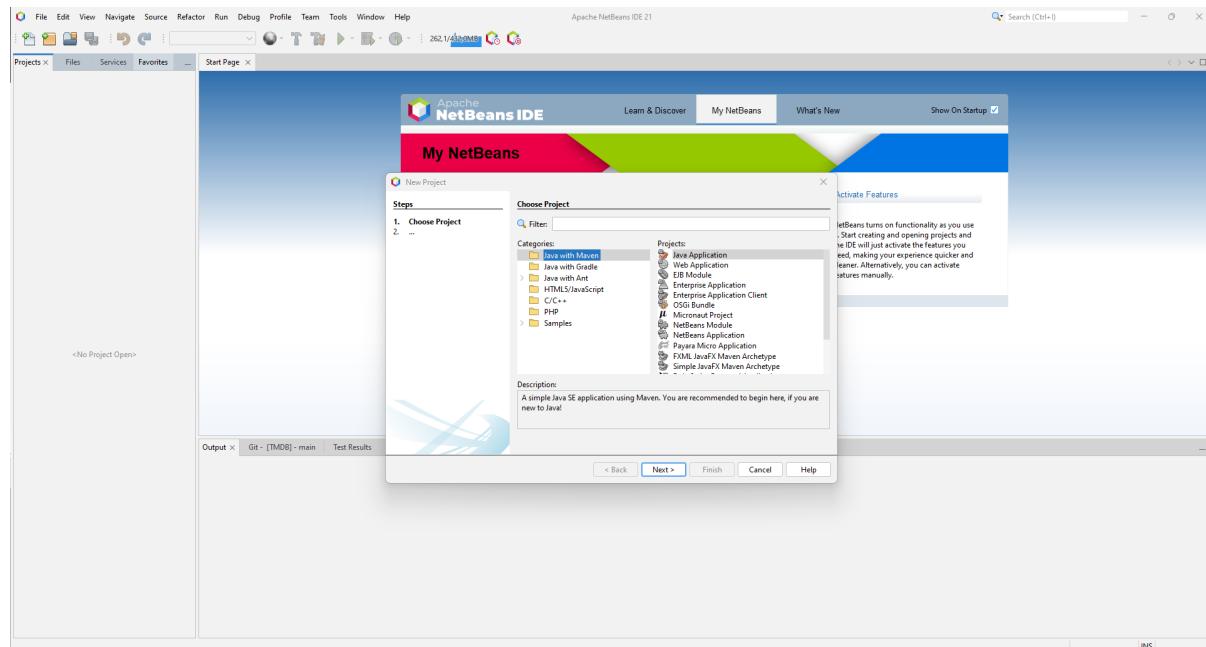
Comentarios

Toda aplicación que se desarrolle en cualquier lenguaje de programación debe ser documentada mediante comentarios. En Java hay tres formas de indicar comentarios :

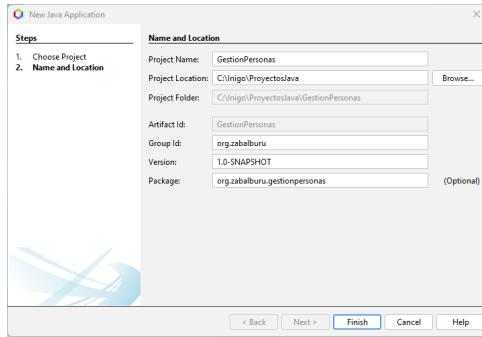
- *//* : El resto de la línea se considera un comentario.
- */ * /* : Todo lo incluido entre estos símbolos se considera un comentario.
- */ * /* : *Todo es un comentario que además puede usarse a la hora de generar la documentación de la aplicación de forma automática** (mediante alguna utilidad como la aplicación `javadoc` del JDK).

1.4. Ejemplo Creación de un Programa Básico

Vamos a **crear un programa básico en Java** que nos permita empezar a ver cómo trabajar con clases y nos hagamos una idea de cómo vamos a trabajar a lo largo del curso. Una explicación más detallada sobre las clases se verá más adelante en el curso. **Vamos a crear una aplicación que nos permita guardar y consultar la información de tres personas.** De cada persona vamos a almacenar su nombre, apellidos y edad. Como podemos ver, **todas las personas tienen que poder almacenar la misma información aunque cada cual tendrá sus propios valores** (su propio nombre, apellido y edad). En **Programación Orientada a Objetos**, definiríamos la **clase Persona con las propiedades (o campos)** nombre, apellidos y edad. Veamos cómo se hace en **NetBeans**. **El primer paso es crear un nuevo proyecto.** Abrimos Netbeans y seleccionamos File --> New Project --> Java with Maven --> Java Application:



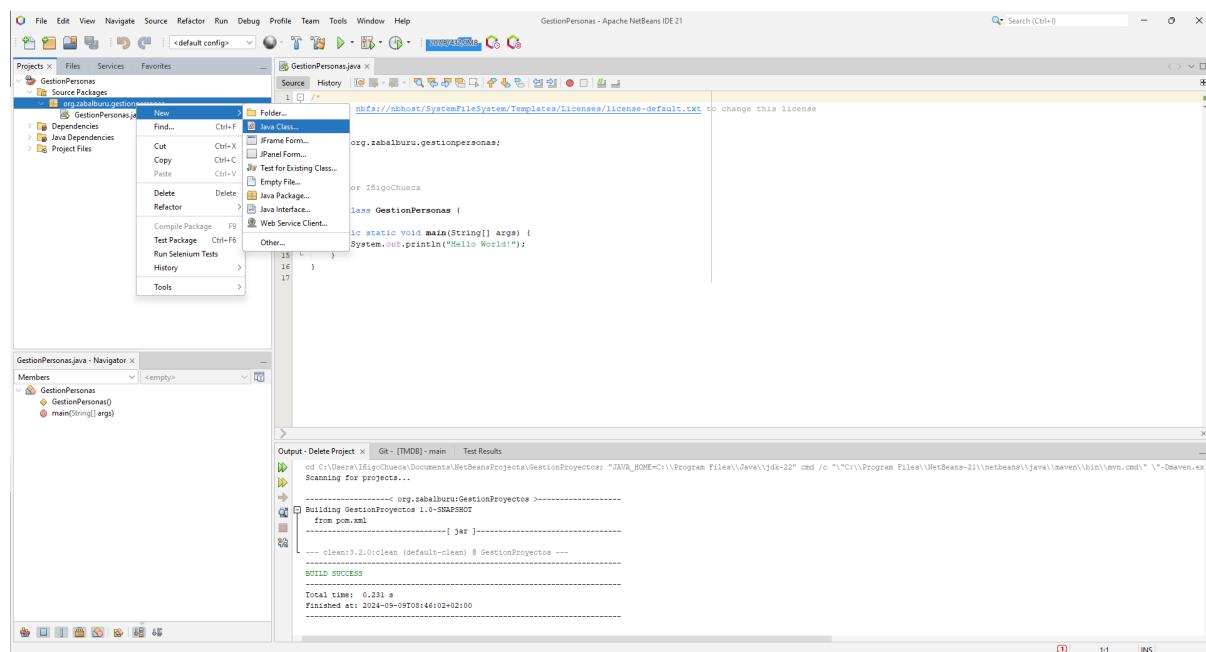
Luego especificamos los datos de nuestro proyecto:



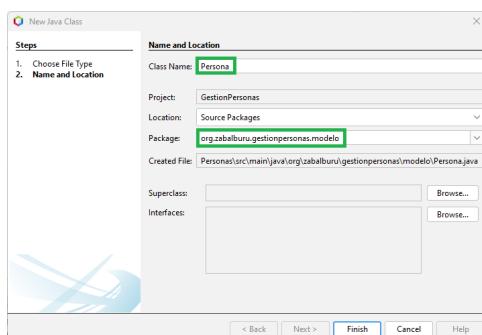
- Project Name**: El nombre del proyecto. Será también el nombre de la carpeta donde se almacenará la aplicación
- Project Location**: La carpeta donde almacenamos los proyectos
- Project Folder**: La carpeta específica del proyecto
- Artifact Id**: El nombre de la aplicación (coincide con el del proyecto)
- Group Id**: El nombre que identifica a la empresa creadora del proyecto
- Version**: La versión del proyecto
- Package**: La ubicación de las clases del proyecto (el **Group Id** junto con el **Artifact Id** en minúsculas). Recordemos que se corresponde con la estructura de carpetas en la que se crearán las clases.

Creación de la clase Persona

Nuestro proyecto ya tiene una clase ejecutable (`GestionPersonas`) en el paquete (carpeta) `org.zabalburu.gestionpersonas`. Ahora deberemos definir la clase `Persona` con sus campos. Normalmente las clases en las que almacenamos información se suelen poner en un paquete distinto. Es habitual llamar a ese paquete `modelo` (**el modelo representa los tipos de datos con los que trabaja la aplicación**). Creamos la clase `Persona` dentro del paquete `org.zabalburu.gestionpersonas.modelo`. Para ello elegimos `New --> Java Class` desde el menú contextual del paquete de la aplicación:



Y ponemos los datos:



Ahora podemos escribir el código:

```
package org.zabalburu.gestionpersonas.modelo;

/**
 *
 * @author IñigoChueca
 */
public class Persona {
    // Aquí definimos los campos o propiedades de cada Persona
    public String nombre; // Aquí guardaremos el nombre de cada persona como una cadena de caracteres (String)
    public String apellidos; // Aquí guardamos los apellidos
    public int edad; // La edad es un valor entero

    // Aquí vamos a definir con código una funcionalidad (método) para mostrar los datos de la Persona por la consola
    // El método se llama saludo
    // Como no retorna nada, se indica void
    public void saludo() {
        // Para mostrar información por pantalla empleamos System.out.println(lo que queremos mostrar);
        // Hay que tener cuidado con las mayúsculas / minúsculas
        System.out.println("Me llamo " + nombre); // El operador + (concatenación) concatena el texto Me llamo con el CONTENIDO del campo nombre
        System.out.println("Me apellido " + apellidos);
        System.out.println("Y tengo " + edad + " años.");
    }
}
```

Básicamente estamos diciendo lo siguiente:

- **Toda persona tiene un nombre y unos apellidos de tipo texto**
- Para especificar una **cadena** en el código de Java la ponemos **entre comillas dobles**.
- **Toda persona tiene una edad que es un número entero**
- Para especificar un **entero** en el código de Java simplemente se escribe un **número sin decimales**.
- Si le **mandamos el mensaje** `saludo()` a una persona, **mostrará por pantalla** cómo se llama y su edad
- `System.out.println(expresión)` es el mecanismo que nos proporciona Java para **escribir el resultado de evaluar una expresión, mostrarlo por pantalla y situarse en la línea siguiente**.
- + es un **operador** que **concatena una cadena con otra cosa y retorna otra cadena**. Si en `nombre` hemos guardado "Juan", "Me llamo " + nombre devolverá "Me llamo Juan" que es lo que se mostrará por pantalla con la instrucción anterior (evidentemente sin las comillas).

Podemos ver que el **mensaje lleva paréntesis** lo que indica que **representa una acción que se le pide hacer a la clase**. Las acciones son código Java. Lo mismo pasa con `println()` que indica que es código que Java va a ejecutar.

Creación del Programa

Ahora vamos a ver cómo podemos **modificar nuestro programa para almacenar la información de una persona y luego mostrarla por pantalla**. Para **crear un objeto a partir de una clase** se emplea un **método especial de Java** llamado **constructor** y debe existir obligatoriamente (si no existe, Java lo creará por nosotros). Nosotros no lo hemos puesto (ya veremos cómo se hace) por lo que Java lo creará por su cuenta. El **constructor** se llama **igual que la clase y para ejecutarlo hay que emplear el operador new**. Dado que es un **método**, el constructor **requiere paréntesis**:

```
new Persona()
```

¿Qué es lo que hace? **Crea un objeto** de tipo `Persona`, **lo guarda en memoria** y **devuelve dónde está** para que podamos acceder a él. Una vez creado **necesitamos acceder a él, así que lo guardaremos** en una zona de memoria a la que asignamos un nombre para poder acceder a el objeto (lo que se conoce como una variable). Para ello emplearemos la siguiente instrucción:

```
Persona personal = new Persona();
```

Se crea una persona nueva y se guarda en `personal`. Para **acceder a las propiedades** de la `personal` pondremos `personal.propiedad (personal.nombre)` y para **pedirle que ejecute código** pondremos `personal.método () (personal.saludo ())`. Veamos el código:

```
package org.zabalburu.gestionpersonas;

import org.zabalburu.gestionpersonas.modelo.Persona;

public class GestionPersonas {

    public static void main(String[] args) {
        // Creamos las personas
        Persona personal = new Persona();
        // Asignamos valores a las propiedades de la primera persona
        personal.nombre = "Luis";
        personal.apellidos = "Marcial Sanz";
        personal.edad = 35;
        // Le pedimos que salude
        personal.saludo();
    }
}
```

Como se puede ver, **creamos una persona, le pasamos los datos y le pedimos que salude**. La salida:

Me llamo Luis

Me apellido Marcial Sanz
Y tengo 35 años.

Vamos a **modificarlo para añadir datos de tres personas, mostrar su datos y su edad media.**

```
package org.zabalburu.gestionpersonas;

import org.zabalburu.gestionpersonas.modelo.Persona;

/**
 *
 * @author IñigoChueca
 */
public class GestionPersonas {

    public static void main(String[] args) {
        // Creamos las personas
        Persona personal = new Persona();
        Persona persona2 = new Persona();
        Persona persona3 = new Persona();
        // Asignamos valores a las propiedades de la primera persona
        personal.nombre = "Luis";
        personal.apellidos = "Marcial Sanz";
        personal.edad = 35;
        // Le pedimos que salude
        personal.saludo();

        persona2.nombre = "Ana";
        persona2.apellidos = "Simón Pérez";
        persona2.edad = 25;

        persona3.nombre = "Luisa";
        persona3.apellidos = "De Andrés López";
        persona3.edad = 28;

        // Mostramos los datos
        System.out.println(""); // Una linea en blanco
        persona2.saludo();
        System.out.println(""); // Una linea en blanco
        persona3.saludo();

        // Calculamos la edad media:
        int edadMedia = (personal.edad + persona2.edad + persona3.edad) / 3; // + es el operador suma y / el operador división
        System.out.println(""); // Una linea en blanco
        System.out.println("La edad media es de " + edadMedia + " años.");
    }
}
```

La salida:

Me llamo Luis
Me apellido Marcial Sanz
Y tengo 35 años.

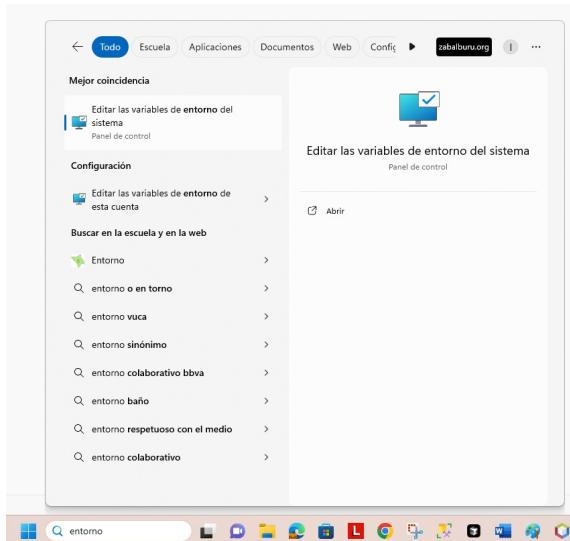
Me llamo Ana
Me apellido Simón Pérez
Y tengo 25 años.

Me llamo Luisa
Me apellido De Andrés López
Y tengo 28 años.

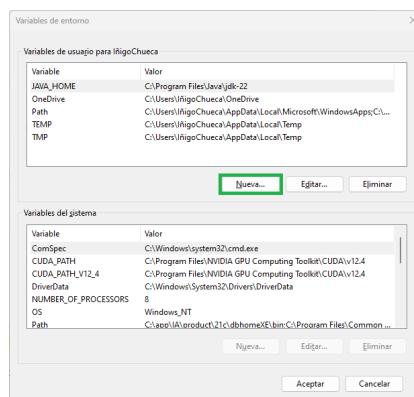
La edad media es de 29 años.

Cuando ejecutemos **veremos que la consola no muestra los datos correctamente**. Esto es debido a que emplea una **codificación diferente a la de nuestro ordenador** (`UTF-8`). Para solucionarlo hay que dar los siguientes pasos:

- Vamos a la **búsqueda de windows** y escribimos entorno:



- Abrimos la aplicación **Editar las variables de entorno del sistema**. Seleccionamos la opción **Nueva**:



Y copiamos lo siguiente:

- **Nombre de la variable :** JAVA_TOOL_OPTIONS
- **Valor de la variable:** -Dfile.encoding=UTF-8 -Dsun.jnu.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=UTF-8 -Dconsole.encoding=UTF-8



Guardamos todo y reiniciamos Netbeans.

Añadir nuevos métodos con parámetros y que retornan información

Como hemos visto, la clase `Persona` tiene una serie de propiedades (`nombre`, `apellidos` y `edad`) y un método (`saludo()`) que simplemente muestra los datos de la persona por pantalla. **Supongamos que queremos almacenar información sobre el sueldo de cada persona.** Para ello, lo primero que tendríamos que hacer es definir una propiedad en la clase para almacenarlo:

```
public int sueldo;
```

Asumiendo que ese es el **sueldo bruto** de una persona queremos ahora **calcular el sueldo neto** (lo que cobra realmente). Simplificando mucho, el **sueldo neto es el sueldo bruto pero descontando lo que se paga a hacienda (IRPF : Impuesto de la Renta sobre las Personas Físicas)**. El **IRPF** es un **porcentaje que depende de los ingresos brutos anuales y del número de hijos**. Como vemos, para poder calcular el **sueldo neto** necesitamos el **sueldo** y el **porcentaje de IRPF**. Dado que vamos a necesitar ejecutar código tenemos que **definir un método para que haga los cálculos adecuados** (restar del **sueldo** lo correspondiente al **IRPF**) y **retorne el resultado** para que quien use la clase (nuestro programa principal) pueda obtener dicha información. Nuestra clase `Persona` sólo sabe cuál es el **sueldo** de la persona (está en la propiedad `sueldo`) pero no sabe cuál es el **porcentaje de IRPF**. ¿Quién tiene esa información? Quien quiera calcular el sueldo neto debería saber el porcentaje de IRPF. La pregunta es: **¿cómo pasamos esa información al método?**. La respuesta es: **mediante parámetros**. Un **parámetro no es más que un dato que se le pasa a un método para que pueda realizar su tarea** (en nuestro caso, el **porcentaje de IRPF**). Los **parámetros se especifican cuando se crea el método** indicando **su tipo y su nombre entre los paréntesis del método** (en la práctica es una variable). Cuando el usuario **ejecuta (invoca)** el método, **debe proporcionar un valor adecuado (del tipo indicado)** para cada **parámetro** en su llamada (indicándolo entre paréntesis). Un **método puede tener 0, 1 o múltiples parámetros y cada uno de ellos puede ser de un tipo distinto**. Todos los **parámetros deben ir entre paréntesis y separados por comas**. Cuando se llama un método, se le deben **pasar valores para cada uno de sus parámetros y se asignan los valores a los parámetros en base a su posición**. Por tanto a la hora de **definir el método** pondríamos `getSueldo(int porcIRPF)` asumiendo que el `porcIRPF` es un **entero**.

El nombre del método `getSueldo` sigue las **recomendaciones de Java para métodos** que retornan un valor como veremos más adelante.

Perfecto, pero hemos dicho que **nuestro método debe retornar el sueldo neto**. ¿Cómo lo indicamos en el método y cómo lo retornamos?. Un **método tiene el siguiente formato** (firma):

```
public tipoRetorno nombreMetodo([parámetros]) {  
    código;  
    [return resultado];  
}
```

donde **los corchetes indican elementos opcionales**.

- **tipoRetorno** : Es el **tipo de dato que va a retornar el método** (`int, String, Date...`). Si el **método no retorna nada** (como en el método `saludo()`) se utiliza la palabra `void`. En nuestro caso, retornaremos un entero
- **parámetros** : Como ya hemos indicado es una **lista separada con comas de los datos que se tienen que proporcionar al método para que pueda realizar su trabajo**. Cada parámetro tendrá el formato `tipo nombre` y, si no hay ningún parámetro, se indican simplemente los paréntesis* (`(saludo())`)
- **return** : **Especifica el fin de la función y, opcionalmente, retorna un valor**. Si el método no devuelve nada (`void`) puede omitirse o poner simplemente `return;`. Si el método no es `void` debe especificarse el valor que se deve devolver tras la ejecución del código (evidentemente, el tipo de dicho valor debe coincidir con el `tipoRetorno` especificado en el método).

Por tanto, nuestra clase quedaría así:

```
package org.zabalburu.gestionpersonas.modelo;  
  
/**  
 *  
 * @author IñigoChueca  
 */  
public class Persona {  
    // Aquí definimos los campos o propiedades de cada Persona  
    public String nombre; // Aquí guardaremos el nombre de cada persona como una cadena de caracteres (String)  
    public String apellidos; // Aquí guardamos los apellidos  
    public int edad; // La edad es un valor entero  
    public int sueldo; // El sueldo es un entero  
  
    // Aquí vamos a definir con código una funcionalidad (método) para mostrar los datos de la Persona por la consola  
    // El método se llama saludo  
    // Como no retorna nada, se indica void  
    public void saludo() {  
        // Para mostrar información por pantalla empleamos System.out.println(lo que queremos mostrar);  
        // Hay que tener cuidado con las mayúsculas / minúsculas  
        System.out.println("Me llamo " + nombre); // El operador + (concatenación) concatena el texto Me llamo con el CONTENIDO del campo nombre  
        System.out.println("Me apellido " + apellidos);  
        System.out.println("Y tengo " + edad + " años.");  
    }  
  
    // Asumimos que si el porcIRPF es 21 esto significa un 21%  
    // Cuando el usuario llame a getSueldoNeto(21) el valor 21 se guardará en porcIRPF  
    public int getSueldoNeto(int porcIRPF){  
        int irpf = sueldo * porcIRPF / 100;  
        int sueldoNeto = sueldo - irpf;  
        return sueldoNeto;  
        // return sueldo - sueldo * porcIRPF / 100  
    }  
}
```

Modificamos la aplicación para hacer uso de la nueva propiedad y método: La salida:

```
Me llamo Luis  
Me apellido Marcial Sanz  
Y tengo 35 años.  
Sueldo Bruto : 1450€  
Sueldo Neto : 1218€
```

```
Me llamo Ana  
Me apellido Simón Pérez  
Y tengo 25 años.  
Sueldo Bruto : 2100€  
Sueldo Neto : 1659€
```

```
Me llamo Luisa  
Me apellido De Andrés López  
Y tengo 28 años.  
Sueldo Bruto : 2100€  
Sueldo Neto : 1617€
```

La edad media es de 29 años.

1.5. Actividad 1 - Creación de una aplicación básica

Instrucciones

Crear con el NetBeans un nuevo **proyecto Maven** denominado `Actividad01`. Como opciones deberéis poner las **siguientes** (el resto dejarlas como están):

- **Project Name** : `Actividad01`
- **Group Id** : `org.zabalburu.daw1`

En el proyecto deberéis **crear una clase** llamada `Producto` que conste de los **siguientes campos/propiedades**:

- `nombre`: Un objeto `String` que almacenará el **nombre del producto**
- `precio`: Un valor entero (`int`) donde se **guardará el precio del producto** (asumimos que no tiene decimales)

Además la clase dispondrá de los siguientes **métodos**:

- `public void mostrar()`: Este método mostrará por pantalla el nombre y el precio del producto con el siguiente formato:

`Producto : nombre Precio: precio€`

- `public int getImporte(int unidades)`: Este método retornará el resultado de multiplicar las unidades vendidas multiplicadas por el precio del producto

En el método `main` de la clase principal (`Actividad01` si habéis seguido los pasos) se deberán ejecutar las siguientes instrucciones:

- **Crear dos productos y almacenarlos en dos variables** (`producto1` y `producto2`)
- **Asignar un nombre y un precio a cada uno de los productos** (los que queráis)
- **Mostrar la información de cada producto por pantalla** (llamando al método `mostrar()` sobre cada producto)
- **Definir una variable** `unidades` de tipo `int` y **almacenar** en ella `5` (suponemos que tenemos 5 unidades de cada producto)
- **Definir dos variables** `importe1` e `importe2` y **almacenar en ellas** el resultado de **llamar** al método `getImporte` de cada producto
- **Mostrar el importe de vender esas unidades de cada uno de los productos**. Deberán aparecer mensajes similares al siguiente:

`Vendidas unidades unidades de nombre (precio : precio). Importe : importe€`

Por ejemplo debería mostrarse algo similar a lo siguiente:

`Vendidas 5 unidades de zapatillas (precio: 87). Importe : 435€`

- **Cambiar el valor de** `unidades` en el programa y volverlo a ejecutar. **Comprobar cómo se modifican los importes**

ADICIONAL:

- **Modificar la clase** `Producto` para que **retorne con otro método el importe con IVA (21%)**. Modificar el programa para que muestre también dicho importe
- **Incluir la posibilidad de que haya un descuento**. Modificar la clase y el programa como creáis oportuno.

1.6. Ejemplos Prácticos con Solución

Lecturas

Enunciado

Crear con NetBeans un nuevo proyecto Maven denominado `GestionLibreria`. Como opciones, deberás configurar lo siguiente (el resto déjalo como está):

- **Project Name:** `GestionLibreria`
- **Group Id:** `org.zabalburu.dawi`

En el proyecto, deberás crear una clase llamada `Libro` que conste de los siguientes **campos/propiedades**:

- `título`: Un objeto `String` que almacenará el **título del libro**
- `páginas`: Un valor entero (`int`) donde se guardará el **número de páginas del libro**

Además, la clase dispondrá de los siguientes **métodos**:

- `public void mostrarInfo()`: Este método **mostrará por pantalla** el `título` y el número de `páginas` del libro con el siguiente formato: `Libro: [título] - Páginas: [número de páginas]`
- `public int calcularTiempoLectura(int páginasPorHora)`: Este método retornará el resultado de **dividir** el número de `páginas` del libro entre las **páginas por hora** que se pasan como **parámetro**. La idea es calcular el número de horas necesario para leer el libro en base al número de páginas que leemos en una hora

En el método `main` de la clase principal (`GestionLibreria` si has seguido los pasos) se deberán ejecutar las siguientes **instrucciones**:

1. **Crear dos libros** y almacenarlos en dos **variables** (`libro1` y `libro2`)
2. Asignar un `título` y un `número de páginas` a cada uno de los libros
3. Mostrar la información de cada libro por pantalla (método `mostrarInfo()`)
4. **Definir una variable** `páginasPorHora` de tipo `int` y **almacenar** en ella `30`
5. **Definir dos variables** `tiempoLectura1` y `tiempoLectura2` y **almacenar** en ellas el resultado de llamar al método `calcularTiempoLectura` de cada libro
6. **Mostrar** el `tiempo estimado de lectura` para cada libro. Deberán aparecer mensajes similares al siguiente: `Tiempo de lectura para '[título]': [tiempo] horas`
7. Cambiar el valor de `páginasPorHora` en el programa a `40` y volver a **calcular y mostrar** los tiempos de lectura

Solución

Implementación de la clase `Libro`:

```
public class Libro {
    public String título;
    public int páginas;
```

```

public void mostrarInfo() {
    System.out.println("Libro: " + titulo + " - Páginas: " + paginas);
}
public int calcularTiempoLectura(int paginasPorHora) {
    return paginas / paginasPorHora;
}
}

```

Implementación de la clase principal GestionLibreria:

```

public class GestionLibreria {
    public static void main(String[] args) {
        Libro libro1 = new Libro();
        Libro libro2 = new Libro();

        libro1.titulo = "El Quijote";
        libro1.paginas = 863;

        libro2.titulo = "Cien años de soledad";
        libro2.paginas = 471;

        libro1.mostrarInfo();
        libro2.mostrarInfo();

        int paginasPorHora = 30;

        int tiempoLectura1 = libro1.calcularTiempoLectura(paginasPorHora);
        int tiempoLectura2 = libro2.calcularTiempoLectura(paginasPorHora);

        System.out.println("Tiempo de lectura para '" + libro1.titulo + "' : " + tiempoLectura1 + " horas");
        System.out.println("Tiempo de lectura para '" + libro2.titulo + "' : " + tiempoLectura2 + " horas");

        paginasPorHora = 40;
        tiempoLectura1 = libro1.calcularTiempoLectura(paginasPorHora);
        tiempoLectura2 = libro2.calcularTiempoLectura(paginasPorHora);

        System.out.println("\nCon nueva velocidad de lectura:");
        System.out.println("Tiempo de lectura para '" + libro1.titulo + "' : " + tiempoLectura1 + " horas");
        System.out.println("Tiempo de lectura para '" + libro2.titulo + "' : " + tiempoLectura2 + " horas");
    }
}

```

Salida

```

Libro: El Quijote - Páginas: 863
Libro: Cien años de soledad - Páginas: 471
Tiempo de lectura para 'El Quijote': 28 horas
Tiempo de lectura para 'Cien años de soledad': 15 horas

```

```

Con nueva velocidad de lectura:
Tiempo de lectura para 'El Quijote': 21 horas
Tiempo de lectura para 'Cien años de soledad': 11 horas

```

Flota de Vehículos

Enunciado

Crear con NetBeans un nuevo proyecto Maven denominado `GestionFlotaVehiculos`. Como opciones, deberás configurar lo siguiente (el resto déjalo como está):

- **Project Name:** `GestionFlotaVehiculos`
- **Group Id:** `org.zabalburu.daw1`

En el proyecto, deberás crear una clase llamada `Vehiculo` que conste de los siguientes **campos/propiedades**:

- `modelo`: Un objeto `String` que almacenará el modelo del vehículo
- `velocidadMaxima`: Un valor entero (`int`) donde se guardará la velocidad máxima del vehículo en km/h
- `consumo`: Un valor entero (`int`) que representa el consumo de combustible en litros por 100 km
- `capacidadTanque`: Un valor entero (`int`) que representa la capacidad del tanque de combustible en litros

Además, la clase dispondrá de los siguientes **métodos**:

- `public void mostrarInfo()`: Este método mostrará por pantalla la información completa del vehículo con el siguiente formato: `Vehiculo: [modelo] - Velocidad Máxima: [velocidadMaxima] km/h - Consumo: [consumo] L/100km - Capacidad del tanque: [capacidadTanque]`
- `public int calcularTiempoViaje(int distancia)`: Este método retornará el resultado de dividir la `distancia` (en km) entre la `velocidad máxima` del vehículo (asumiendo que viaja a velocidad constante)
- `public int calcularConsumoViaje(int distancia)`: Este método retornará el consumo de combustible para la distancia dada
- `public int calcularAutonomia()`: Este método retornará la distancia máxima que puede recorrer el vehículo con un tanque lleno

En el método `main` de la clase principal (`GestionFlotaVehiculos` si has seguido los pasos) se deberán ejecutar las siguientes **instrucciones**:

1. **Crear dos vehículos** (`vehiculo1, vehiculo2`)

2. Asignar un modelo, velocidad máxima, consumo y capacidad del tanque a cada uno de los vehículos
3. Mostrar la información de cada vehículo por pantalla (método mostrarInfo())
4. Definir una variable distancia de tipo int y almacenar en ella 500 (km)
5. Para cada vehículo, calcular y mostrar:
 6. Tiempo de viaje para la distancia dada
 7. Consumo de combustible para la distancia dada
 8. Autonomía del vehículo

Solución

Implementación de la clase Vehículo:

```
public class Vehiculo {
    public String modelo;
    public int velocidadMaxima;
    public int consumo;
    public int capacidadTanque;
    public void mostrarInfo() {
        System.out.println("Vehículo: " + modelo + " - Velocidad Máxima: " + velocidadMaxima +
                           " km/h - Consumo: " + consumo + " L/100km - Capacidad del tanque: " +
                           capacidadTanque + " L");
    }
    public int calcularTiempoViaje(int distancia) {
        return distancia / velocidadMaxima;
    }
    public int calcularConsumoViaje(int distancia) {
        return (consumo * distancia) / 100;
    }
    public int calcularAutonomia() {
        return (capacidadTanque * 100) / consumo;
    }
}
```

Implementación de la clase principal GestionFlotaVehiculos:

```
public class GestionFlotaVehiculos {
    public static void main(String[] args) {
        Vehiculo vehiculo1 = new Vehiculo();
        vehiculo1.modelo = "Sedan Eléctrico";
        vehiculo1.velocidadMaxima = 180;
        vehiculo1.consumo = 15; // kWh/100km en este caso
        vehiculo1.capacidadTanque = 60; // kWh en este caso

        Vehiculo vehiculo2 = new Vehiculo();
        vehiculo2.modelo = "SUV Hibrido";
        vehiculo2.velocidadMaxima = 200;
        vehiculo2.consumo = 5;
        vehiculo2.capacidadTanque = 45;

        vehiculo1.mostrarInfo();
        vehiculo2.mostrarInfo();

        int distancia = 500;

        System.out.println("\nCálculos para una distancia de " + distancia + " km:");

        int tiempoViaje = vehiculo1.calcularTiempoViaje(distancia);
        int consumoViaje = vehiculo1.calcularConsumoViaje(distancia);
        int autonomia = vehiculo1.calcularAutonomia();

        System.out.println(vehiculo1.modelo + ":");

        System.out.println(" Tiempo de viaje: " + tiempoViaje + " horas");
        System.out.println(" Consumo del viaje: " + consumoViaje + " L");
        System.out.println(" Autonomía: " + autonomia + " km");

        System.out.println(); // Dejamos una línea en blanco

        tiempoViaje = vehiculo2.calcularTiempoViaje(distancia);
        consumoViaje = vehiculo2.calcularConsumoViaje(distancia);
        autonomia = vehiculo2.calcularAutonomia();

        System.out.println(vehiculo2.modelo + ":");

        System.out.println(" Tiempo de viaje: " + tiempoViaje + " horas");
        System.out.println(" Consumo del viaje: " + consumoViaje + " L");
        System.out.println(" Autonomía: " + autonomia + " km");
    }
}
```

Salida

Vehículo: Sedan Eléctrico - Velocidad Máxima: 180 km/h - Consumo: 15 L/100km - Capacidad del tanque: 60 L
 Vehículo: SUV Hibrido - Velocidad Máxima: 200 km/h - Consumo: 5 L/100km - Capacidad del tanque: 45 L

Cálculos para una distancia de 500 km:

Sedan Eléctrico:
 Tiempo de viaje: 2 horas
 Consumo del viaje: 75 L
 Autonomía: 400 km

SUV Hibrido:
 Tiempo de viaje: 2 horas
 Consumo del viaje: 25 L
 Autonomía: 900 km

1.7. Tipos de Datos variables y expresiones

Tipos de datos

Un ordenador básicamente se encarga de procesar información. Un programa normalmente recibe un conjunto de datos de entrada, los procesa y entrega una serie de datos de salida. Podemos definir un **dato como cada elemento de información que puede ser utilizado por un programa**. Los lenguajes de programación pueden trabajar con **diferentes tipos de datos**. En JAVA para almacenar información, disponemos de **tipos primitivos y tipos de referencia**.

Tipos Primitivos

Son aquellos que **almacenan un valor y no derivan de otros tipos de datos**. Tenemos los siguientes tipos primitivos en Java:

Tipo	Bits	Defecto	Rango valores
boolean	8	false	true, false
byte	8	0	-128 a +127
short	16	0	-32.768 a +32767
int	32	0	-2.147.483.648 a +2.147.483.647
long	64	0	-9.223.372.036.854.775.808 a +9.223.372.036.854.775.807
float	32	0.0F	aprox. -3.4 e38 a +3.4 e38 con 7 decimales
double	64	0.0D	aprox. -1.8 e308 a +1.8 e308 con 16 decimales
char	16	x0	cualquier carácter Unicode. Los primeros 127 caracteres coinciden con el código ASCII

Constantes Literales

Se denomina así a los **datos cuyo valor no cambia en la ejecución de la aplicación**. Se pueden emplear directamente en el programa y JAVA les asignará el tipo adecuado en función de cómo estén escritas. Si escribimos un **número sin decimales (y sin separador de miles)**, JAVA entenderá que es un **int**. Si a dicho número le **añadimos una l mayúscula o minúscula**, asumirá que es un **long**. Si el **número lleva separador decimal (recordemos que es el punto)**, JAVA asumirá que es un **double**. Y si le añadimos una **f** al final (tenga o no decimales) asumirá que es un **float**. Cualquier carácter **encerrado entre comillas simples ('')** se considerará un **char**. En Java se emplean los **caracteres Unicode** que permite especificar además de los caracteres normales caracteres de otros idiomas (como árabe, chino, indio o japonés). Los caracteres Unicode se expresan con **\uhhhh** donde h es un dígito hexadecimal. Por ejemplo, el símbolo π es **\u03c0**

La mayor parte de los caracteres Unicode no se pueden mostrar en la consola)

Adicionalmente se admiten los siguientes caracteres de control :

Carácter	Significado
\n	nueva línea
\t	salto de tabulación
\b	retroceso
\r	retorno de carro
\f	salto de página
\\\	el carácter \
\'	el carácter '
\"	el carácter "
\ddd	el carácter especificado en octal
\xdd	el carácter especificado en hexadecimal
\uhhhh	el carácter en código Unicode.

Los literales **true** y **false** escritos tal cual (en minúsculas y sin delimitadores) se consideran valores **boolean**. Cualquier **combinación de caracteres encerrados entre comillas dobles ("")** se consideran **objetos** de la clase **String**. La cadena **null** escrita tal cual representa un **objeto vacío**. Para clarificar el código podemos asignar un **identificador a un valor constante**. Para ello hay que precederlo de la declaración **final**, del **tipo de datos** y, de un **igual** seguido del **valor** a asignarle.

Se recomienda que los identificadores de las constantes vayan escritas en mayúsculas

Ejemplo

- 14, 3, -1 o 12 son datos de tipo int
- 2L, -5L son datos de tipo long
- 4.0, -2.234 son datos de tipo double
- 4.0F, -2.234F, 8F son datos de tipo float
- 'a', '9', '\xFF' son datos de tipo char
- "Hola" es un objeto de tipo String Para declarar constantes
- final int IVA = 21;
- final boolean OK = true;

Tipos de Referencia

Los tipos de datos de referencia son más complejos y se utilizan para referirse a objetos. En la práctica almacenan la ubicación de objetos o estructuras en memoria e incluyen clases, interfaces, arrays, enumeraciones y registros (record) a partir de Java 16.

Clases

Ya hemos visto que una clase se define con la palabra reservada class.

Interfaces

Un interface es similar a una clase pero, en principio, sólo especifica una serie de métodos que deben tener los objetos que lo implementan. Las veremos en detalle más adelante. Se especifican con la palabra reservada interface. Luego las clases pueden definir esos métodos implementando el interfaz:

```
public interface Animal {  
    public String getTipo();  
    public String getGrito();  
}  
  
public class Perro implements Animal {  
    @Override  
    public String getTipo(){  
        return "Perro";  
    }  
    @Override  
    public String getGrito(){  
        return "Ladrar";  
    }  
}  
  
public class Gato implements Animal {  
    @Override  
    public String getTipo(){  
        return "Gato";  
    }  
    @Override  
    public String getGrito(){  
        return "Maullar";  
    }  
}  
  
...  
Animal elvis = new Perro();  
// Soy un Perro y sé Ladrar  
System.out.println("Soy un " + elvis.getTipo() + " y sé " + elvis.getGrito())  
Animal bowie = new Gato();  
// Soy un Gato y sé Maullar  
System.out.println("Soy un " + bowie.getTipo() + " y sé " + bowie.getGrito())  
...
```

Arrays (Matrices)

Las matrices son estructuras de datos que almacenan múltiples valores del mismo tipo a los que se puede acceder a través de un índice:

```
String[] nombres = {"Juan", "Ana", "Luis", "María"};  
System.out.println(nombres[0]); // "Juan"
```

Enumeraciones

Las enumeraciones son tipos de datos que consisten en un conjunto de constantes. Permiten clarificar el código.

```
public enum DiaSemana {  
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO  
}  
  
...  
DiaSemana hoy = DiaSemana.MIERCOLES;  
System.out.println("Hoy es: " + hoy); // Hoy es MIERCOLES  
System.out.println("Número: " + hoy.ordinal()); // Número: 2
```

...

Record (Registros)

Un record es una clase que se utiliza principalmente para almacenar datos. Proporciona una forma compacta de declarar una clase que es principalmente un contenedor de datos. Los record automáticamente generan métodos como equals(), hashCode(), y toString(), así como un constructor, basados en los campos que definas.

```
// A partir de Java 14
public record Punto(int x, int y) {}

...
Punto punto = new Punto(5, 10);
System.out.println("Punto: " + punto);
System.out.println("X: " + punto.x());
System.out.println("Y: " + punto.y());
```

Variables

Se denomina así a los **elementos que permiten almacenar datos utilizados por el programa**. Una variable **se identifica por un nombre (identificador)**. Al nombrar el **identificador en el programa nos estamos refiriendo al valor que contiene**. El **dato** contenido en una variable (valor de la variable) **puede cambiar a lo largo de la ejecución del programa** (a diferencia de las constantes). Para **definir una variable** basta con especificar su **tipo, seguido de su identificador**. Los identificadores en Java :

- son **sensibles a mayúsculas / minúsculas**. No es lo mismo nota que Nota o NOTA.
- deben **comenzar por una letra o símbolo del subrayado** y se recomienda que sólo lleven letras (aunque pueden llevar dígitos y el carácter de subrayado)
- **no pueden llevar espacios en medio**
- se **recomienda que se escriban con la primera letra en minúsculas**. Si está formado por **varias palabras, la inicial** de cada una de ellas irá en **mayúscula (camelCase)**

Ejemplo

```
int nota;
float media;
boolean aprobado;
String nombreAlumno;
```

Expresiones

Una **expresión** es una **combinación de constantes, variables, métodos unidos por operadores** y que, al ser evaluadas, **devuelven un dato (resultado)**. Todos los lenguajes de programación disponen de los mismos operadores básicos aunque no siempre se escriben de la misma forma. En nuestro caso, como ejemplo, estudiaremos los operadores de Java.

Expresiones aritméticas

Trabajan sobre **datos numéricos** y el resultado es también numérico. Podemos distinguir entre **operadores unarios** (trabajan con un único operando) y **operadores binarios** (trabajan sobre dos operandos).

Operadores Unarios

Operador	Comentario
+	Si el operando es un byte, short o char lo convierte a int
-	Negación . Invierte el signo del operador.
++	Incremento
--	Decremento

. Los operadores ++ y -- actúan de distinta forma si se ponen **antes o después del valor**. Por ejemplo, suponiendo que numEntero valga 10, tras la siguiente expresión:

```
numNuevo = ++numEntero + 10
```

numNuevo contendría 21 y numEntero 11 (**primero se incrementa numEntero y luego se suma -preincremento-**).

Por el contrario, en el mismo supuesto, tras:

```
numNuevo = numEntero++ + 10
```

numNuevo contendría 20 y numEntero 11, puesto que **no se incrementa numEntero hasta no realizar la suma (postincremento)**.

Operadores binarios :

Operador	Comentario
+	Suma
-	Resta
*	Multiplicación

Operador	Comentario
/	División ($7/2 \rightarrow 3$, $7.0/2 \rightarrow 3.5$)
%	Resto de la división ($7\%2 \rightarrow 1$, $7.5 \% 2 \rightarrow 1.5$)

En Java, la **división entre dos enteros da un resultado entero** (truncando el cociente). Así si ponemos la expresión `14 / 4` el resultado sería `3` y no `3.5`. Para obtener un **valor con decimales es necesario que alguno de los datos lleve decimales**. Las expresiones `14.0 / 4`, `14 / 4.0`, `14.0 / 4.00` o `14F / 4` sí retornarían `3.5`.

Posteriormente volveremos sobre este tema al estudiar la conversión de datos en Java.

Ejemplo

```
package org.zabalburu.operadores;

/**
 *
 * @author IñigoChueca
 */
public class OperadoresAritmeticos {

    public static void main(String[] args) {
        final double IVA = 0.21;
        double precioEntrada = 30.5;
        int numEntradas = 2;
        double importe = numEntradas * precioEntrada;
        double importeConIVA = importe * (1 + IVA);
        System.out.println("El importe de las entradas es " + importe + "€.");
        System.out.println("Importe (IVA incluido) : " + importeConIVA + "€.");
        System.out.println("=====");
        int plazasAutobus = 35;
        int personas = 110;
        int autobuses = personas / plazasAutobus;
        System.out.println("Hacen falta " + autobuses + " autobuses para llevar a " + personas + " personas");
        int personasSinAutobus = personas % plazasAutobus;
        System.out.println("Quedarian " + personasSinAutobus + " personas sin plaza");
        System.out.println("=====");
        int num1 = 10;
        int num2 = 20;
        int rdo1 = ++num1 + num2;
        System.out.println("++num1 + num2 : " + rdo1);
        System.out.println("num1 : " + num1 + "\tnum2 : " + num2);
        num1 = 10;
        num2 = 20;
        int rdo2 = num1++ + num2;
        System.out.println("num1++ + num2 : " + rdo2);
        System.out.println("num1 : " + num1 + "\tnum2 : " + num2);
    }
}
```

La salida:

```
El importe de las entradas es 61.0€.
Importe (IVA incluido) : 73.81€.
=====
Hacen falta 3 autobuses para llevar a 110 personas
Quedarian 5 personas sin plaza
=====
++num1 + num2 : 31
num1 : 11      num2 : 20
num1++ + num2 : 30
num1 : 11      num2 : 20
```

Expresiones de cadena

Operador	Comentario
+	Concatenación

El operador concatenación sirve para **retornar una nueva cadena con la concatenación de las dos cadenas** a las que se aplica el operador.

Expresiones de comparación

Los operadores de comparación comparan dos valores del mismo tipo y retornan un valor booleano.

Operador	Comentario
>	Mayor que
<	Menor que
>=	Mayor o igual
<=	Menor o igual
==	Igual

Operador	Comentario
<code>!=</code>	Distinto

Dado que las cadenas en Java son objetos NO SE PUEDEN COMPARAR con los operadores de comparación, es decir, "`A <= B`" dará un error de compilación. Sólo se pueden emplear los operadores `==` y `!=` pero funcionan de una manera especial que explicaremos posteriormente al estudiar los objetos más detenidamente.

Ejemplo

```
package org.zabalburu.operadores;

/**
 *
 * @author IñigoChueca
 */
public class OperadoresComparacion {

    public static void main(String[] args) {
        System.out.println("12 >= 10 : " + (12 >= 10));
        System.out.println("12 < 10 : " + (12 < 10));
        System.out.println("12 == 24 / 2 : " + (12 == 24 / 2));
        System.out.println("a' > 'b' : " + ('a' > 'b'));
        System.out.println("a' > 'B' : " + ('a' > 'B'));
        System.out.println("a' == 'a' : " + ('a' == 'a'));
        System.out.println("a' == 'A' : " + ('a' == 'A'));
    }
}
```

La salida:

```
12 >= 10 : true
12 < 10 : false
12 == 24 / 2 : true
'a' > 'b' : false
'a' > 'B' : true
'a' == 'a' : true
'a' == 'A' : false
```

Expresiones lógicas

Los operadores lógicos trabajan con operandos lógicos y el resultado es un valor lógico.

Operador	Comentario
<code>&&</code>	AND true si ambos operandos son true
<code>'</code>	
<code>^</code>	XOR true si sólo uno de los dos es true
<code>!</code>	NOT Invierte el valor lógico
<code>&</code>	AND true si ambos operandos son true
<code> </code>	
<code>?:</code>	Op?I1:I2 Si Op es true se retorna I1, si no se retorna I2

Tablas Lógicas

Las tablas lógicas o tablas de la verdad permiten ver cómo se comportan los operadores lógicos en todas las circunstancias posibles:

```
package org.zabalburu.operadores;

/**
 *
 * @author IñigoChueca
 */
public class OperadoresLogicos {

    public static void main(String[] args) {
        System.out.println("Op1\t\tOp1\t\tOp1 && Op2");
        System.out.println("----\t----\t----");
        System.out.println("true\ttrue\t" + (true && true));
        System.out.println("true\tfalse\t" + (true && false));
        System.out.println("false\ttrue\t" + (false && true));
        System.out.println("false\tfalse\t" + (false && false));
        System.out.println("=====");
        System.out.println("Op1\t\tOp1\t\tOp1 || Op2");
        System.out.println("----\t----\t----");
        System.out.println("true\ttrue\t" + (true || true));
        System.out.println("true\tfalse\t" + (true || false));
        System.out.println("false\ttrue\t" + (false || true));
        System.out.println("false\tfalse\t" + (false || false));
        System.out.println("=====");
        System.out.println("Op1\t\tOp1\t\tOp1 ^ Op2");
        System.out.println("----\t----\t----");
        System.out.println("true\ttrue\t" + (true ^ true));
```

```

        System.out.println("true\ntfalse\t" + (true ^ false));
        System.out.println("false\true\t" + (false ^ true));
        System.out.println("false\tnfalse\t" + (false ^ false));
        System.out.println("=====");
        System.out.println("Op1\tt!Op1");
        System.out.println("---\t\t----");
        System.out.println("true\t" + !true);
        System.out.println("false\tt" + !false);
        System.out.println("=====");
        System.out.println("Op1\ttOp1?1:2");
        System.out.println("---\t\t----");
        System.out.println("true\t" + (true ? 1 : 2));
        System.out.println("false\tt" + (false ? 1 : 2));
    }
}

```

La salida:

```

Op1      Op1      Op1 && Op2
---      ---      -----
true     true     true
true     false    false
false    true     false
false    false    false
=====
Op1      Op1      Op1 || Op2
---      ---      -----
true     true     true
true     false    true
false    true     true
false    false    false
=====
Op1      Op1      Op1 ^ Op2
---      ---      -----
true     true     false
true     false    true
false    true     true
false    false    false
=====
Op1      !Op1
---      ---
true     false
false    true
=====
Op1      Op1?1:2
---      -----
true     1
false   2

```

La diferencia entre los operadores `&&` y `||` respecto a `&` y `|` es que los primeros proporcionan lo que se denomina **cortocircuitado de expresiones**. Si a la hora de **comparar dos valores** con `&&` el **primer** es `false` la expresión **siempre será** `false` por lo que **Java no evalúa la segunda expresión** (si empleamos el operador `&` **sí se evaluaría** aunque el resultado de la expresión ya se sepa). Lo mismo ocurre con los operadores `||` y si el **primer operando** es `true` (con `||` **no se evaluaría el segundo operador** dado que la expresión ya va a ser cierta y con `|` sí se evaluaría).

Ejemplo Cortocircuitado Expresiones

```

package org.zabalburu.operadores;

/**
 *
 * @author IñigoChueca
 */
public class CortocircuitadoExpresiones {

    public static void main(String[] args) {
        int num = 10;
        System.out.println("true || ++num>10 : " + (true || ++num > 10));
        System.out.println("num : " + num);
        num = 10;
        System.out.println("true || ++num>10 : " + (true | ++num > 10));
        System.out.println("num : " + num);
        System.out.println("-----");
        System.out.println("false && ++num>10 : " + (false && ++num > 10));
        System.out.println("num : " + num);
        num = 10;
        System.out.println("false & ++num>10 : " + (false & ++num > 10));
        System.out.println("num : " + num);
    }
}

```

La salida:

```

true || ++num>10 : true
num : 10
true || ++num>10 : true
num : 11
-----
false && ++num>10 : false
num : 11

```

```
false & ++num>10 : false  
num : 11
```

Operadores binarios

Trabajan con los **operando**s en modo binario.

Operador	Comentario
&	AND de bits 1 & 1 → 1, resto 0
^	XOR. $1 \wedge 1$ o $0 \wedge 0 \rightarrow 0$, resto 1
~	Complemento a 1. Pasa los unos a ceros y al revés
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha
>>>	Desplazamiento a la derecha con relleno de ceros

Hay que tener en cuenta a la hora de trabajar con datos binarios que en Java se almacenan los valores negativos en formato de complemento a 2 (se invierten los unos y los ceros y, tras ello, se suma 1). Por tanto los valores negativos tienen siempre el bit de más peso a 1. Por ejemplo, el valor -100 se representaría de la siguiente forma como un entero (4 octetos → 32 bits): 00000000 00000000 00000000 01100100 → 100 en binario 11111111 11111111 11111111 10011011 → Complemento a uno de 100 11111111 11111111 11111111 10011100 → Complemento a 2 (-100) La diferencia entre >> (desplazamiento aritmético) y >>> (desplazamiento lógico) es que, en los números negativos, el desplazamiento aritmético muerde manteniendo el signo mientras que el desplazamiento lógico no.

Ejemplo

La salida:

Otros operadores

Existe otro tipo de operadores que no encaja en ninguno de los grupos vistos hasta ahora y que son:

Operador	Comentario
[]	Declarar matrices, inicializarlas y acceder a sus elementos.
.	Definir nombres cualificados
()	Especificar una lista de parámetros o para convertir un tipo de dato a otro (casting).
instanceof	Comprobar si el primer operando es una instancia de la clase definida como segundo operando. var instanceof T devuelve true si var es una instancia de la clase T o de alguna clase ascendente de T.

Prioridad de operadores

Cuando en una expresión se **combinan diferentes tipos de operadores** (numéricos, lógicos, relacionales ...) es **preciso tener en cuenta la prioridad** que se dará a dichos operadores para evaluar la expresión. Si no se indica lo contrario, la prioridad mayor la tienen los operadores aritméticos, después los relacionales y, por último, los lógicos. De mayor a menor prioridad:

Prioridad
[] ()
++ (preincremento) -- (predecremento) ! ~ instanceof
* / %
+ -
++ (postincremento) -- (postdecremento)
<< >> >>>
'<'>
'=='
&
^
'`
&&, &
'`
? :
= += -= *= /= %= ^= &= != <=>= >>>=

Para modificar la prioridad se pueden utilizar paréntesis.

Ejemplo

```
public static void main(String[] args) {
    double d = 4 + 9.0 / 2 % 3;
    /*
     * d = 4 + 4.5 % 3
     * d = 4 + 1.5
     * d = 5.5
     */
    System.out.println("4 + 9.0 / 2 % 3 : " + d);
    d = (4 + 9.0) / (2 % 3);
    /*
     * d = 13.0 / 2
     * d = 6.5
     */
    System.out.println("(4 + 9.0) / (2 % 3) : " + d);
    char curso = 'A';
    int edad = 25;
    boolean b = curso == 'A' || curso == 'B' && edad < 18;
    System.out.println("curso == 'A' || curso == 'B' && edad < 18 : " + b);
    b = (curso == 'A' || curso == 'B') && edad < 18;
    System.out.println("(curso == 'A' || curso == 'B') && edad < 18 : " + b);
}
```

La salida:

```
4 + 9.0 / 2 % 3 : 5.5
(4 + 9.0) / (2 % 3) : 6.5
curso == 'A' || curso == 'B' && edad < 18 : true
(curso == 'A' || curso == 'B') && edad < 18 : false
```

Cadenas (String)

Las cadenas en Java son **instancias (objetos)** de la clase `String`. La diferencia entre un tipo de dato básico y un objeto es que el primero sólo puede almacenar **un valor** que posteriormente podremos recuperar, mientras que los segundos pueden almacenar **múltiples valores** y código que realiza **diversas tareas con dichos valores**. Por ejemplo, en una **cadena** se almacenan **todos los caracteres** de la misma y podemos **realizar diversas operaciones** sobre la misma (retorñala en mayúsculas / minúsculas, extraer parte de la misma...). Estas operaciones se **implementan escribiendo el código** que realiza dichas tareas dentro de la **clase** (los llamados **métodos de la clase**). Para **ejecutar un método** de una clase se emplea el operador `.` (punto). Para **inicializar un objeto** de una clase se emplea el operador `new` aunque, en el caso de la clase `String`, también se puede **crear una cadena simplemente escribiéndola entre comillas dobles**. Podemos ver los métodos de una clase en la **documentación** de la misma ([API](#)). Por ejemplo, el **API** de la clase `String` para la versión 8 del JDK está en <https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>. Como ya hemos comentado antes para **comparar dos objetos no se deben emplear** `==` y `!=` y no se pueden emplear `>`, `<`, `>=` ni `<=`. Las clases que permiten comparar objetos tienen sus propios métodos:

- `objeto.equals(otroObjeto)` : Retorna `true` si `objeto` es igual a `otroObjeto`, `false` en caso contrario. Es el **programador** de la clase el que **decide** qué se tiene que cumplir para que dos objetos de esa clase se consideren iguales
- `objeto.compareTo(otroObjeto)` : Sólo aparece en aquellas clases que **tiene sentido decir que un objeto puede ser mayor que otro** (por ejemplo cadenas, fechas...). Si se define **retorna un valor mayor que 0** si `objeto` es **mayor** que `otroObjeto`, un **valor 0** si **ambos objetos son iguales** y un **valor menor que 0** si `objeto` es **menor que otroObjeto**.

En el caso de la clase `String`, además, se han definido los métodos `equalsIgnoreCase` y `compareToIgnoreCase` que realizan el mismo proceso pero **sin distinguir entre mayúsculas y minúsculas**.

```
package org.zabalburu.operadores;

/**
 *
 * @author IñigoChueca
 */
public class ClaseString {

    public static void main(String[] args) {
        String nombre = "Juan López";
        System.out.println("Nombre : " + nombre);
        System.out.println("Minúsculas : " + nombre.toLowerCase());
        System.out.println("Inicial : " + nombre.charAt(0));
        System.out.println("5 primeros caracteres : " + nombre.substring(0, 5)); // Coge el 0,1,2,3,4
        System.out.println("nombre.substring(0,4).equals(\"juan\") : " + nombre.substring(0, 4).equals("juan"));
        System.out.println("nombre.substring(0,4).equals(\"Juan\") : " + nombre.substring(0, 4).equals("Juan"));
        System.out.println("nombre.substring(0,4).equalsIgnoreCase(\"juan\") : " + nombre.substring(0, 4).equalsIgnoreCase("juan"));
        System.out.println("\"b\".compareTo(\"a\") : " + "b".compareTo("a"));
        System.out.println("\"b\".compareTo(\"b\") : " + "b".compareTo("b"));
        System.out.println("\"b\".compareTo(\"c\") : " + "b".compareTo("c"));
        System.out.println("\\"b\\\".compareTo(\"C\") : " + "b".compareTo("C"));
        System.out.println("\\"b\\\".compareToIgnoreCase(\"c\") : " + "b".compareToIgnoreCase("c"));
    }
}
```

La salida:

```
Nombre : Juan López
Minúsculas : juan lópez
Inicial : J
5 primeros caracteres : Juan
nombre.substring(0,4).equals("juan") : false
nombre.substring(0,4).equals("Juan") : true
nombre.substring(0,4).equalsIgnoreCase("juan") : true
"b".compareTo("a") : 1
"b".compareTo("b") : 0
"b".compareTo("c") : -1
"b".compareTo("C") : 31
"b".compareToIgnoreCase("c") : -1
```

Operación de Asignación

Podríamos **definir esta operación como la asignación a una variable del valor resultante de la evaluación de una expresión**:

```
variable = expresión
```

En Java, además, se puede **realizar una operación a la vez que se asigna el valor**:

```
var op= expr → var = var op expr
```

Operador	Comentario
<code>+=</code>	Suma el valor y después asigna
<code>-=</code>	Resta y asignación
<code>*=</code>	Multiplicación y asignación
<code>/=</code>	División y asignación
<code>%=</code>	Resto y asignación
<code>&=</code>	AND y asignación (binario)
<code>\ =</code>	OR y asignación (binario)
<code>^=</code>	XOR y asignación (binario)
<code><<=</code>	Desplazamiento izquierda y asignación
<code>>>=</code>	Desplazamiento derecha y asignación
<code>>>>=</code>	Desplazamiento con relleno de ceros y asignación

Tabla de conversiones

De \ A	char	byte	short	int	long	float	double	String
char	Directo	Cast implícito	toString()					
byte	(char)	Directo	Cast implícito	toString()				
short	(char)	(byte)	Directo	Cast implícito	Cast implícito	Cast implícito	Cast implícito	toString()
int	(char)	(byte)	(short)	Directo	Cast implícito	Cast implícito	Cast implícito	toString()
long	(char)	(byte)	(short)	(int)	Directo	Cast implícito	Cast implícito	toString()
float	(char)	(byte)	(short)	(int)	(long)	Directo	Cast implícito	toString()
double	(char)	(byte)	(short)	(int)	(long)	(float)	Directo	toString()
String	charAt()	parseByte()	parseShort()	parseInt()	parseLong()	parseFloat()	parseDouble()	Directo
boolean	Boolean.parseBoolean()	N/A	N/A	N/A	N/A	N/A	N/A	String.valueOf()

1.8. Ejemplos de Operadores

Ejemplos de operadores en Java:

1. Operadores Aritméticos:

Ejemplo 1

Calcular el salario bruto de un trabajador, considerando su salario base y el porcentaje de plus por antigüedad.

```
double salarioBase = 1500;
double porcentajePlus = 0.15;
double plusAntiguedad = salarioBase * porcentajePlus; // Plus por antigüedad
double salarioBruto = salarioBase + plusAntiguedad; // Salario bruto

System.out.println("Salario Bruto: " + salarioBruto); // Salida: Salario Bruto: 1725.0
```

Ejemplo 2

Calcular cuántos billetes de denominación (por ejemplo, 100, 50 o 20) se necesitan para entregar un importe en efectivo.

```
int importe = 375;
int denominacion = 50;

int cantidadBilletes = importe / denominacion; // Cantidad de billetes sin considerar el resto
int resto = importe % denominacion; // Cantidad de dinero que sobra

System.out.println("Para entregar " + importe + " se necesitan " + cantidadBilletes + " billetes de " + denominacion);
System.out.println("Queda un resto de " + resto + " para entregar");
```

Ejecución: Si importe es 375 y denominacion es 50, el código mostrará:

```
Para entregar 375 se necesitan 7 billetes de 50
Queda un resto de 25 para entregar
```

Explicación:

- cantidadBilletes calcula la **cantidad de billetes necesarios para cubrir** el importe, ignorando el resto. En este caso, 375 dividido entre 50 da 7.
- resto calcula la **cantidad de dinero que queda después de entregar la mayor cantidad posible de billetes de la denominación especificada**. En este caso, 375 menos ($7 * 50$) da 25.

Ejemplo 3

Calcular el precio final de un producto con un descuento aplicado. Problema: La división entre enteros en Java trunca la parte decimal.

```
int precioOriginal = 92;
int descuento = 10; // Descuento del 10%

int precioFinal = precioOriginal - (precioOriginal * descuento / 100); // Precio final sin decimales (truncado)

System.out.println("Precio final: " + precioFinal); // Salida: Precio final: 82
```

En este caso, el **precio final con el descuento** aplicado sería **82.80** ($92 - (92 * 0.10)$). Sin embargo, al usar enteros, el **resultado se trunca a 82**, perdiendo la parte decimal. Para obtener un resultado decimal se necesitan las siguientes soluciones: Soluciones:

1. Convertir al menos uno de los operandos a double:

```
double precioFinal = precioOriginal - ((double)precioOriginal * descuento / 100); // Castear precioOriginal a double

System.out.println("Precio final: " + precioFinal); // Salida: Precio final: 82.8
```

2. Dividir por un double:

```
double precioFinal = precioOriginal - (precioOriginal * descuento / 100.0); // Dividir por 100.0 (double)

System.out.println("Precio final: " + precioFinal); // Salida: Precio final: 82.8
```

3. Declarar las variables como double:

```
double precioOriginal = 92;
double descuento = 10; // Descuento del 10%
```

```
double precioFinal = precioOriginal - (precioOriginal * descuento / 100); // Calcular precio final con double
System.out.println("Precio final: " + precioFinal); // Salida: Precio final: 82.8
```

2. Operadores de Cadena:

Objetivo: Crear un mensaje personalizado para un cliente, usando su nombre y el nombre del producto que compró.

```
String nombreCliente = "Juan";
String nombreProducto = "Camiseta";

String mensaje = "Hola " + nombreCliente + ", gracias por comprar la " + nombreProducto + "!";
System.out.println(mensaje); // Salida: Hola Juan, gracias por comprar la Camiseta!
```

3. Operadores de Relación:

Ejemplo 1

Comparar dos edades (edad1 y edad2) para determinar si la primera es mayor que la segunda.

```
int edad1 = 25;
int edad2 = 30;

boolean esMayor = edad1 > edad2; // Compara si edad1 es mayor que edad2

System.out.println("Edad 1 es mayor que edad 2: " + esMayor); // Salida: Edad 1 es mayor que edad 2: false
```

Ejemplo 2. Problema con referencias a objetos:

Queremos comparar dos objetos de la clase Persona (personal y persona2) para determinar si son iguales.

Problema: Los operadores == y != para objetos de clase comparan las referencias en memoria, no los atributos (campos o propiedades).

```
class Persona {
    public String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }
}

Persona personal = new Persona("Juan");
Persona persona2 = new Persona("Juan");

boolean sonIguales = personal == persona2; // Compara referencias en memoria

System.out.println("Las personas son iguales: " + sonIguales); // Salida: Las personas son iguales: false

Persona persona3 = personal; // Asignamos la referencia a la primera persona a la variable persona3

sonIguales = personal == persona3; // Compara referencias en memoria

System.out.println("Las personas son iguales: " + sonIguales); // Salida: Las personas son iguales: true
```

En este ejemplo, personal y persona2 son objetos distintos en memoria, aunque tienen el mismo nombre, por lo que == devuelve false. Sin embargo, persona3 es una referencia al mismo objeto que personal, por lo que == devuelve true. **Solución:** Implementar equals() en la clase Persona

```
class Persona {
    public String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public boolean equals(Persona otra) {
        return nombre.equals(otra.nombre); // El método equals de String retorna true si las dos cadenas son iguales
    }
}

Persona personal = new Persona("Juan");
Persona persona2 = new Persona("Juan");
Persona persona3 = new Persona("Pedro");

boolean sonIguales = personal.equals(persona2); // Compara contenido

System.out.println("Las personas son iguales: " + sonIguales); // Salida: Las personas son iguales: true

sonIguales = personal.equals(persona3); // Compara contenido

System.out.println("Las personas son iguales: " + sonIguales); // Salida: Las personas son iguales: false
```

En este caso, equals() compara el atributo nombre de ambos objetos Persona. Si los nombres son iguales, devuelve true, de lo contrario devuelve false. **Conclusión:**

- Para comparar objetos de clase, es esencial implementar el método equals() para comparar los atributos relevantes, no solo las referencias en memoria.
- Implementar equals() garantiza una comparación correcta del contenido de los objetos, independientemente de su ubicación en memoria.

Ejemplo con Cadenas

Comparar dos cadenas de texto (cadena1 y cadena2) para determinar si son iguales.

Los literales, en Java, son **valores constantes** por lo que sólo se crean una vez y **luego se reutilizan**. Es decir si ponemos la cadena "Hola" en un programa, Java crea un objeto `String` con el texto y devuelve su **referencia**. Si, en otro punto del programa, volvemos a escribir el literal "Hola" **exactamente igual**, Java **retorna la referencia al objeto creado anteriormente**.

Si creamos cadenas en Java con el **constructor** y el operador `new` (como cualquier otro objeto) **se crean objetos distintos cada vez**.

```
String cadena1 = "Hola";
String cadena2 = "Hola";

boolean sonIguales = cadena1 == cadena2; // Compara referencias en memoria

System.out.println("Las cadenas son iguales: " + sonIguales); // Salida: Las cadenas son iguales: true

String cadena3 = new String("Hola");

sonIguales = cadena1 == cadena3; // Compara referencias en memoria

System.out.println("Las cadenas son iguales: " + sonIguales); // Salida: Las cadenas son iguales: false
```

En este ejemplo, `cadena1` y `cadena2` apuntan al mismo objeto `String` en memoria (dado que es un literal), por lo que `==` devuelve `true`. Sin embargo, `cadena3` es un **nuevo objeto** `String` en memoria, aunque tenga el mismo contenido, por lo que `==` devuelve `false`. **Solución:** Usar el método `equals()`

El método `equals()` compara el contenido de los objetos `String`, independientemente de las referencias en memoria.

```
String cadena1 = "Hola";
String cadena2 = "Hola";
String cadena3 = new String("Hola");

boolean sonIguales = cadena1.equals(cadena2); // Compara contenido

System.out.println("Las cadenas son iguales: " + sonIguales); // Salida: Las cadenas son iguales: true

sonIguales = cadena1.equals(cadena3); // Compara contenido

System.out.println("Las cadenas son iguales: " + sonIguales); // Salida: Las cadenas son iguales: true
```

En este caso, `equals()` devuelve `true` para ambas comparaciones, ya que el contenido de las cadenas es el mismo, independientemente de que sean objetos distintos en memoria.

Conclusión:

- Para comparar **valores primitivos** (enteros, `double`, etc.) se pueden usar los operadores de relación `==` y `!=`.
- Para **comparar objetos**, se debe usar el método `equals()` para evitar problemas de comparación de referencias.

4. Operadores Lógicos:

Ejemplo 1

Determinar si un cliente cumple con las condiciones para obtener un descuento: debe tener `edad` mayor o igual a 18 años y `puntos` acumulados mayores a 1000.

```
int edad = 25;
int puntosAcumulados = 1200;

boolean cumpleCondicion = edad >= 18 && puntosAcumulados > 1000;

System.out.println("El cliente cumple con las condiciones para el descuento: " + cumpleCondicion); // Salida: El cliente cumple con las condiciones para el
```

Ejemplo 2 - Cortocircuito en Operadores Lógicos:

Evaluando una condición que involucra una `edad` y una condición médica para determinar si una persona puede donar sangre. La persona debe tener al menos 18 años y no tener ninguna condición médica que impida la donación.

```
int edad = 17;
boolean tieneCondicionMedica = true;

boolean puedeDonarSangre = edad >= 18 && !tieneCondicionMedica; // Evaluar condición de donación

System.out.println("Puede donar sangre: " + puedeDonarSangre); // Salida: Puede donar sangre: false
```

En este caso, la condición `edad >= 18` se evalúa primero y resulta en `false`. Como el operador `&&` (y) realiza un **cortocircuito**, la segunda parte de la expresión (`!tieneCondicionMedica`) no se evalúa, ya que el resultado final ya está definido como `false`. **Ejemplo con cortocircuito en el operador || (o)**. Se permite la donación si tiene 18 años o más o está en buenas condiciones

```
int edad = 20;
boolean tieneCondicionMedica = false;

boolean puedeDonarSangre = edad >= 18 || tieneCondicionMedica; // Evaluar condición de donación

System.out.println("Puede donar sangre: " + puedeDonarSangre); // Salida: Puede donar sangre: true
```

En este caso, la condición `edad >= 18` se evalúa primero y resulta en `true`. Como el operador `||` (o) realiza un cortocircuito, la segunda parte de la expresión (`tieneCondicionMedica`) no se evalúa, ya que el resultado final ya está definido como `true`.

Ejemplo3 - Cuidado con el cortocircuitado de expresiones

Supongamos el siguiente código:

```
int contador = 0;
boolean condicion = false;
```

```

boolean resultado = condicion && ++contador == 5; // Cortocircuito en ||
System.out.println("Resultado: " + resultado); // Resultado: false
System.out.println("Contador: " + contador); // Contador: 0

// La segunda parte de la expresión && no se evalúa porque la primera es false,
// por lo que el contador no se incrementa.

```

Al emplear **cortocircuito de expresiones el contador no se incrementa** lo que puede resultar un problema si es lo que pretendíamos hacer. **Solución:** Emplear los operadores sin cortocircuito:

```

int contador = 0;
boolean condicion = false;

boolean resultado = condicion & ++contador == 5; // Cortocircuito en ||
System.out.println("Resultado: " + resultado); // Resultado: false
System.out.println("Contador: " + contador); // Contador: 1

// Ahora se evalúan ambas condiciones

```

5. Operadores de Asignación:

Objetivo: Incrementar el saldo de una cuenta bancaria en un importe determinado.

```

double saldo = 1000;
double importe = 250;

saldo += importe; // Aumenta el saldo en 250

System.out.println("Saldo actual: " + saldo); // Salida: Salario actual: 1250

```

1.9. Actividad 2 - Operadores

Actividad 2 - Operadores

Colección de Videojuegos

Descripción:

Crea una clase llamada `Videojuego` con los siguientes atributos públicos:

- `nombre: String` (nombre del videojuego)
- `plataforma: String` (plataforma del videojuego, por ejemplo: "PS5", "Xbox Series X", "PC")
- `genero: String` (género del videojuego, por ejemplo: "Acción", "Aventura", "RPG")
- `puntuacion: int` (puntuación del videojuego en una escala del 1 al 10)

Método:

Implementa un **método público** llamado `esRecomendable` que devuelve `true` si la **puntuación del videojuego es mayor o igual a 8**, y `false` en caso contrario.

Aplicación Principal

1. Crea un objeto `Videojuego` llamado `juego1` con los siguientes datos:

2. **Nombre:** "Elden Ring"

3. **Plataforma:** "PS5"

4. **Género:** "RPG"

5. **Puntuación:** 9

6. Crea otro objeto `Videojuego` llamado `juego2` con los siguientes datos:

7. **Nombre:** "Horizon Forbidden West"

8. **Plataforma:** "PS5"

9. **Género:** "Acción"

10. **Puntuación:** 7

11. **Compara las puntuaciones** de ambos videojuegos usando **operadores de relación**. Imprime un mensaje indicando qué videojuego tiene una puntuación más alta.

Recuerda el operador ternario (`? :`)

1. **Compara los géneros** de ambos videojuegos usando operadores de relación. Imprime un mensaje indicando si los videojuegos pertenecen al mismo género.

2. **Comprueba** si `juego1` es recomendable usando su método `esRecomendable`. Imprime un mensaje indicando si es recomendable o no.

3. **Comprueba** si `juego2` es recomendable usando su método `esRecomendable`. Imprime un mensaje indicando si es recomendable o no.

4. **Crea un nuevo videojuego** llamado `juego3` con el **nombre** "God of War Ragnarok", **puntuación** de 9 y la **plataforma** "PS4". Asigna el **género** de `juego3` como el mismo que el de `juego1`.

Ejemplo de salida:

Elden Ring tiene una puntuación más alta que Horizon Forbidden West

Elden Ring y Horizon Forbidden West no pertenecen al mismo género
Elden Ring es recomendable
Horizon Forbidden West no es recomendable

Adicional

1. Muestra por pantalla la **puntuación media de los tres juegos con decimales**
2. Define una **variable local** `plataforma`, **asígnale PS5** y muestra por pantalla **cuántos juegos hay de ese género**.
Necesitarás crear una **variable entera** llamada `cuenta` e ir sumándole 1 en función de si cada juego cumple o no la condición. Cambia la plataforma por `PS4` y comprueba que funciona correctamente
1. Mostrar por pantalla el resultado de evaluar las siguientes condiciones :
2. El `genero` del juego1 es `PS5` o `PS4` y su puntuación debe ser mayor o igual a 7
3. Idem pero la puntuación debe ser 10
4. La puntuación del juego3 está entre 7 y 9 (ambos incluidos) y es **recomendable**
5. La puntuación más alta de los tres juegos (emplead `?:` anidados)

Ejemplo de salida

```
La puntuación media de los tres juegos es de 8.3333334
Hay 2 juegos de PS5
Hay 1 juego de PS4
El género del juego1 es PS5 o PS4 y su puntuación debe ser mayor o igual a 7 : true
El género del juego1 es PS5 o PS4 y su puntuación debe ser 10 : false
La puntuación del juego3 está entre 7 y 9 (ambos incluidos) y es recomendable : true
La puntuación más alta de los tres juegos es 9
```

1.10. Actividad de Repaso

Aplicación de Libros

Descripción:

Crea una clase llamada `Libro` con los siguientes **atributos públicos**:

- `titulo: String` (título del libro)
- `autor: String` (nombre del autor del libro)
- `genero: String` (género del libro, por ejemplo: "Novela", "Ensayo", "Poesía")
- `paginas: int` (número de páginas del libro)

Método:

Implementa un **método público** llamado `esLargo` que devuelve `true` si el libro tiene **más de 300 páginas**, y `false` en caso contrario.

Aplicación Principal

Crea un objeto `Libro` llamado `libro1` con los siguientes datos:

- **Título:** "Cien años de soledad"
- **Autor:** "Gabriel García Márquez"
- **Género:** "Novela"
- **Páginas:** 496

Crea otro objeto `Libro` llamado `libro2` con los siguientes datos:

- **Título:** "El principito"
- **Autor:** "Antoine de Saint-Exupéry"
- **Género:** "Novela"
- **Páginas:** 96

Realiza las **siguientes tareas**:

- **Compara el número de páginas de ambos libros usando operadores de relación.** Imprime un mensaje indicando qué libro tiene más páginas.
Recuerda el operador ternario (`?:`)
- **Compara los géneros de ambos libros** usando operadores de relación. Imprime un mensaje indicando si los libros pertenecen al mismo género.
- **Comprueba si libro1 es largo** usando su método `esLargo`. Imprime un mensaje indicando si es largo o no.
- **Comprueba si libro2 es largo** usando su método `esLargo`. Imprime un mensaje indicando si es largo o no.
- Crea un **nuevo libro** llamado `libro3` con el **título** "Rayuela", el **autor** "Julio Cortázar" y 400 **páginas**. Asigna el **género** de `libro3` como el mismo que el de `libro1`.

Adicional

- Muestra por pantalla la **cantidad promedio de páginas** de los tres libros **con decimales**.
- Define una **variable local** `genero`, asignale `Novela` y muestra por pantalla **cuántos libros hay de ese género**.
- Mostrar por pantalla el resultado de evaluar las siguientes **condiciones**:
- El `genero` del libro1 es `Novela` o `Ensayo` y su número de páginas es mayor o igual a 300. (`true`)
- Idem, pero el número de páginas debe ser 500. (`false`)
- El número de páginas de libro3 está entre 350 y 450 (ambos incluidos) y es largo. (`true`)
- El libro con mayor número de páginas entre los tres (emplea `?:` anidados).

Solución paso a paso:

Clase Libro

```
class Libro {  
    public String titulo;  
    public String autor;  
    public String genero;  
    public int paginas;  
  
    public boolean esLargo() {  
        return this.paginas > 300;  
    }  
}
```

Clase ActividadLibros

```
public class ActividadLibros {  
    public static void main(String[] args) {  
        // 1. Crear objeto libro1  
        Libro libro1 = new Libro();  
        libro1.titulo = "Cien años de soledad";  
        libro1.autor = "Gabriel García Márquez";  
        libro1.genero = "Novela";  
        libro1.paginas = 496;  
  
        // 2. Crear objeto libro2  
        Libro libro2 = new Libro();  
        libro2.titulo = "El principito";  
        libro2.autor = "Antoine de Saint-Exupéry";  
        libro2.genero = "Novela";  
        libro2.paginas = 96;  
  
        // 3. Comparar número de páginas  
        String mensajePaginas = (libro1.paginas > libro2.paginas) ?  
            libro1.titulo + " tiene más páginas que " + libro2.titulo :  
            libro2.titulo + " tiene más páginas que " + libro1.titulo;  
        System.out.println(mensajePaginas);  
  
        // 4. Comparar géneros  
        String mensajeGenero = (libro1.genero.equals(libro2.genero)) ?  
            "Los libros pertenecen al mismo género" :  
            "Los libros pertenecen a géneros diferentes";  
        System.out.println(mensajeGenero);  
  
        // 5. Verificar si libro1 es largo  
        String mensajeLargo1 = (libro1.esLargo()) ?  
            libro1.titulo + " es un libro largo" :  
            libro1.titulo + " no es un libro largo";  
        System.out.println(mensajeLargo1);  
  
        // 6. Verificar si libro2 es largo  
        String mensajeLargo2 = (libro2.esLargo()) ?  
            libro2.titulo + " es un libro largo" :  
            libro2.titulo + " no es un libro largo";  
        System.out.println(mensajeLargo2);  
  
        // 7. Crear objeto libro3  
        Libro libro3 = new Libro();  
        libro3.titulo = "Rayuela";  
        libro3.autor = "Julio Cortázar";  
        libro3.paginas = 400;  
        libro3.genero = libro1.genero;  
  
        // Adicional:  
  
        // Promedio de páginas  
        double promedioPaginas = (libro1.paginas + libro2.paginas + libro3.paginas) / 3.0;  
        System.out.println("El promedio de páginas es: " + promedioPaginas);  
  
        // Contar libros del género "Novela"  
        String generoBuscado = "Novela";  
  
        int cuenta = 0;  
  
        if (libro1.genero.equals(generoBuscado)) {  
            cuenta++;  
        }  
        if (libro2.genero.equals(generoBuscado)) {  
            cuenta++;  
        }  
        if (libro3.genero.equals(generoBuscado)) {  
            cuenta++;  
        }  
        System.out.println("Hay " + cuenta + " libros de Novela");  
    }  
}
```

```

        cuenta++;
    }

    if (libro2.genero.equals(generoBuscado)) {
        cuenta++;
    }

    if (libro3.genero.equals(generoBuscado)) {
        cuenta++;
    }
    System.out.println("Hay " + cuenta + " libros del género " + generoBuscado);

    // Evaluar condiciones
    // Condición 1
    boolean condicion1 = (libro1.genero.equals("Novela") || libro1.genero.equals("Ensayo")) && libro1.paginas >= 300;
    System.out.println("Condición 1: " + condicion1);

    // Condición 2
    boolean condicion2 = (libro1.genero.equals("Novela") || libro1.genero.equals("Ensayo")) && libro1.paginas == 500;
    System.out.println("Condición 2: " + condicion2);

    // Condición 3
    boolean condicion3 = libro3.paginas >= 350 && libro3.paginas <= 450 && libro3.esLargo();
    System.out.println("Condición 3: " + condicion3);

    // Libro con mayor número de páginas
    String libroMasLargo = (libro1.paginas > libro2.paginas) ?
        (libro1.paginas > libro3.paginas ? libro1.titulo : libro3.titulo) :
        (libro2.paginas > libro3.paginas ? libro2.titulo : libro3.titulo);
    System.out.println("El libro con más páginas es: " + libroMasLargo);
}
}

```

1.11. Actividad 22 -Tipos de Datos y Operadores

Actividad 2.2 -Tipos de Datos y Operadores

1.

Una empresa de electricidad **cobra los trabajos por hora y queremos diseñar una aplicación para ellos**. Se pide diseñar una clase `Trabajo` que contenga la siguiente información:

- `cliente` (el nombre del cliente)
- `precioHora` (lo que se cobra por cada hora de trabajo)
- `precioKm` (lo que se cobra por cada km de desplazamiento con decimales)
- `horas` (las horas invertidas en el trabajo con decimales)
- `km` (la distancia a la que está el cliente)
- `IVA` (una **constante** de valor 0.21)
- `SALIDA` (una **constante** de valor 50 que representa un coste fijo por el trabajo)

Además dispondrá de los siguientes **métodos**:

- `double importeHoras()` : el resultado de multiplicar las horas del trabajo por el precio hora.
- `double importeKm()` : el importe correspondiente a los km
- `double importeTotal()` : el coste de la salida más los dos importes anteriores
- `double iva()` : El importe total multiplicado por el % de IVA
- `double importeTotalIva()` : El importe total más el iva
- `void factura()` : Mostrará por pantalla la factura con el siguiente formato:

```

Cliente : cliente
Horas Trabajadas : horas      Precio Hora : precio      Importe : importeHoras
Km. Desplazamiento : km          Precio Km. : precio      Importe : importeKm
Total Factura : importeTotal
I.V.A. (IVA%) : iva
Total Factura (IVA incluido) : importeTotalIva

```

En la clase con el método `main`, realizar las siguientes tareas:

- **Crear** una variable de tipo `Trabajo` y **asignarle valores de prueba** a los campos (los que queráis)
- **Mostrar** la factura por pantalla
- **Mostrar** qué **porcentaje del importe total** (sin iva) corresponde a los kilómetros.

ADICIONAL

Modificar la clase `Trabajo` para que los primeros 10Km sean gratis, es decir, si se han hecho 30km sólo se cobrará por 20. NOTA: Para no complicarnos vamos a asumir que, como mínimo se hacen 10 km.

2.

Queremos crear una clase `Alumno` con los siguientes campos:

- nombre (el nombre del alumno)
- curso ("DAW1", "ASIR2", etc),
- sección ('A', 'B',..),
- nota1, nota2 y nota3 (las notas en tres asignaturas, un valor entre 1 y 10)
- repetidor (verdadero o falso)

Además se deberán definir los siguientes **métodos**:

- double media() : Retornará la **nota media** del alumno
- boolean aprobado() : Para aprobar las **tres notas** tienen que ser **mayores o iguales a 5**
- boolean mediaSuperiorA(double media) : true si la **nota media** del alumno es **superior o igual a la que recibimos**
- boolean suspendidoAlgo() : true si ha **suspendido al menos una de las asignaturas**.

En el método `main`:

- Crear un alumno y asignarle los valores que queráis
- Mostrar por pantalla los siguientes datos:
 - La nota media de cada alumno
 - si el alumno está aprobado
 - si el curso es `DAW1` y la sección es `A`
 - lo mismo sin tener en cuenta mayúsculas y minúsculas
 - si es repetidor y tiene suspendida la nota 1
 - si ha suspendido alguna asignatura
 - si su curso es `DAW1` o `ASIR1` y la sección es `A` o `B`
 - si la nota1 es la más alta de todas
- la nota final, asumiendo que la nota1 es un 30% de la nota final, la nota2 un 50% y la nota3 un 20%
- la nota más alta de las tres

ADICIONAL

En la clase `Alumno`, crear un **método mejorMedia** que reciba otro alumno y retorne true si el alumno tiene media más alta que el que se le pasa. Añadir otro alumno con datos al método `main` y mostrar quién tiene mejor media.

1.12. Ejercicios Adicionales

1.

Crear una clase denominada `Rectángulo` que disponga de los siguientes campos:

- int base
- int altura

Además dispondrá de los siguientes **métodos**:

- int área() : Retornará el área del rectángulo
- int perimetro() : Idem con el perímetro

En el método `main` de la clase creada por NetBeans:

- crear dos rectángulos
- asignarles valores a sus bases y altura
- mostrar por pantalla el perímetro y el área de cada uno de ellos

2.

Crear una clase llamada `Calculadora` que no va a tener campos y dispondrá de los siguientes **métodos**:

- double suma(double n1, double n2)
- double resta(double n1, double n2)
- double multiplicacion(double n1, double n2)
- double division(double n1, double n2)
- boolean divisible(int n1, int n2) --> Retornará true si n1 es divisible por n2

- double mayor(double n1, double n2) -> Retornará el mayor de los dos números

Hacer un programa con un método `main` que cree una calculadora y pruebe todos los métodos (haga cálculos y muestre el resultado)

3.

Crear una clase llamada `Empleado` que tenga los siguientes campos:

- String nombre
- int tipo: Un valor entre 1 y 3
- int horasTrabajadas
- double precioHora
- double retencion: El % que se lleva hacienda (por ejemplo, 0.20 para un 20%)

Definir en la misma los siguientes **métodos**:

- double sueldoBase() : El resultado de multiplicar las horas por el precioHora
- double sueldoNeto() : El resultado de aplicar la retención al sueldoBase
- String tipo() : Si el campo tipo es 1, retornará la cadena Operario, si es 2 retornará Encargado y si es 3 retornará Dirección
- boolean ganaMas(`Empleado` otro) : Retornará true si el empleado actual tiene un sueldo neto mayor o igual que el otro empleado

Hacer una aplicación que cree dos empleados, les asigne valores y muestre sus datos y muestre el nombre del que gana más

4.

Diseñar una clase y un programa que permita **gestionar la venta de entradas de un concierto**. La clase deberá poder **almacenar el nombre del grupo, el número y precio de entradas a la venta y vendidas** y disponer de **métodos que permitan comprar un número de entradas (deberá retornar el importe y descontarlas de las entradas disponibles y sumarlas a las disponibles) así como retornar el importe** de todas las entradas disponibles y vendidas y el **porcentaje** de entradas vendidas. Comprobar su correcto funcionamiento con un programa.

1.13. Entrada Salida de información por pantalla

Una aplicación no sería nada si no pudiera obtener información por parte del usuario. Para ello disponemos de **dos métodos**:

- Emplear la clase `java.util.Scanner` que nos permite leer diferentes tipos de datos desde la consola, ficheros y corrientes de datos
- Emplear la clase `javax.swing.JOptionPane` que nos permite leer cadenas del teclado mediante ventanas

Scanner

La clase `Scanner` tiene métodos para leer textos (`nextLine`), enteros (`nextInt`), dobles (`nextDouble`), etc. Supongamos que queremos leer el **nombre, apellidos y la edad** de una persona. El código podría ser igual al siguiente:

```
import java.util.Scanner;

public class PruebaScanner {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Nombre : ");
        String nombre = sc.nextLine();
        System.out.print("Apellidos : ");
        String apellidos = sc.nextLine();
        System.out.print("Edad : ");
        int edad = sc.nextInt();
        System.out.println("Usted es " + apellidos + ", " +
            nombre + " y su edad es " + edad + " años.");
    }
}
```

La salida:

```

15
16     /**
17      * @param args the command line arguments
18     */
19     public static void main(String[] args) {
20         Scanner sc = new Scanner(System.in);
21         System.out.print("Nombre : ");
22         String nombre = sc.nextLine();
23         System.out.print("Apellidos : ");
24         String apellidos = sc.nextLine();
25         System.out.print("Edad : ");
26         int edad = sc.nextInt();
27         System.out.println("Usted es " + apellidos + ", " +
28             nombre + " y su edad es " + edad + " años.");
29     }
30 }
31 }
32

```

Al crear la clase Scanner le tenemos que decir de dónde vienen los datos. En este caso le indicamos que del **teclado** (`` System.in)

Vamos a **cambiar el orden en el que pedimos** los datos (pedimos la edad antes de los apellidos):

```

package net.zabalburu.mascotas.app;

import java.util.Scanner;

public class PruebaScanner {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Nombre : ");
        String nombre = sc.nextLine();
        System.out.print("Edad : ");
        int edad = sc.nextInt();
        System.out.print("Apellidos : ");
        String apellidos = sc.nextLine();
        System.out.println("Usted es " + apellidos + ", " +
            nombre + " y su edad es " + edad + " años.");
    }
}

```

Si ejecutamos ahora:

```

1 package net.zabalburu.mascotas.app;
2
3 import java.util.Scanner;
4
5 public class PruebaScanner {
6
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9         System.out.print("Nombre : ");
10        String nombre = sc.nextLine();
11        System.out.print("Edad : ");
12        int edad = sc.nextInt();
13        System.out.print("Apellidos : ");
14        String apellidos = sc.nextLine();
15        System.out.println("Usted es " + apellidos + ", " +
16            nombre + " y su edad es " + edad + " años.");
17    }
18

```

Vemos que **no nos pide los apellidos** ¿por qué?. Esto es debido al funcionamiento de la clase Scanner. Cuando escribimos un dato en el teclado siempre finalizamos con un Enter ('\n'). El método `nextLine()` lee todos los caracteres incluido el carácter '\n' pero, el resto de los métodos (`nextInt()`, `nextDouble()`, etc) no lo leen (aunque lo descartan si lo encuentran en primer lugar). El proceso es el siguiente;

- Ejecutamos `sc.nextLine()` que espera a que escribamos algo (que acabe con un Enter)
- Escribimos Juan y pulsamos Enter por lo que se envía Juan\n a la memoria (buffer) del teclado.
- El método `readLine()` lee Juan\n y retorna "Juan" que es lo que se guarda en el nombre. En el buffer del teclado no queda nada
- Ejecutamos `sc.nextInt()` que espera a que escribamos algo (el buffer está vacío)
- Escribimos 21 y pulsamos Enter por lo que se envía 21\n al buffer

- El método `readInt()` lee 21 y lo retorna y el `\n` se queda en el buffer
- Ejecutamos `sc.nextLine()` que coge `\n` del teclado y retorna "" (lo que hay hasta `\n`)

Si después de la edad hubiéramos pedido otro número no habría habido problemas ya que, los números, al encontrarse un `\n` sin información se descarta. Por tanto, tendremos un problema cuando intentamos leer una cadena después de leer un número. Soluciones:

- Consumir el `\n` tras cada lectura de un número (en la práctica si leemos varios números seguidos bastaría con hacerlo en el último). Para ello, añadiríamos un `sc.readLine()` o un `sc.skip("\n")` después del `readInt()`

```

1 package net.zabalburu.mascotas.app;
2
3 import java.util.Scanner;
4
5 public class PruebaScanner {
6
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9         System.out.print("Nombre : ");
10        String nombre = sc.nextLine();
11        System.out.print("Edad : ");
12        int edad = sc.nextInt();
13        sc.nextLine(); //sc.skip("\n");
14        System.out.print("Apellidos : ");
15        String apellidos = sc.nextLine();
16        System.out.println("Usted es " + apellidos + ", " +
17                           nombre + " y su edad es " + edad + " años.");
18    }
}

```

- Emplear `nextLine()` para leer los números como cadenas y luego convertirlos con los métodos `parseType` adecuados de las clases envolventes.

```

1 package net.zabalburu.mascotas.app;
2
3 import java.util.Scanner;
4
5 public class PruebaScanner {
6
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9         System.out.print("Nombre : ");
10        String nombre = sc.nextLine();
11        System.out.print("Edad : ");
12        String strEdad = sc.nextLine();
13        int edad = Integer.parseInt(strEdad);
14        System.out.print("Apellidos : ");
15        String apellidos = sc.nextLine();
16        System.out.println("Usted es " + apellidos + ", " +
17                           nombre + " y su edad es " + edad + " años.");
18    }
}

```

En los ejemplos se comprueba que la clase `Scanner` no lee bien los **caracteres en castellano** (éñes, acentos). Para solucionarlo únicamente hay que indicarle que emplee un sistema de codificación que los incluya. Para ello, emplearemos el siguiente constructor:

```
Scanner sc = new Scanner(System.in, "ISO-8859-1");
```

JOptionPane

La clase `JOptionPane` se encuentra en el paquete `javax.swing` que es el encargado de **permitirnos crear aplicaciones con ventanas** y es la clase que se emplea cuando queremos pedir un dato simple a un usuario, mostrarle información o pedirle confirmación a la hora de hacer una tarea.

Pedir datos al usuario

Para pedir datos al usuario vamos a emplear el método `showInputDialog(mensaje)` que **muestra el mensaje en una ventana y retorna un objeto String** (que, en caso de pedir un valor numérico habrá que convertir). El mismo ejemplo de antes:

```

package net.zabalburu.mascotas.app;
import javax.swing.JOptionPane;

public class PruebaJOptionPane {

    public static void main(String[] args) {
        String nombre = JOptionPane.showInputDialog("Nombre");
        String strEdad = JOptionPane.showInputDialog("Edad");
        int edad = Integer.parseInt(strEdad);
        String apellidos = JOptionPane.showInputDialog("Apellidos");
        System.out.println("Usted es " + apellidos + ", " +
                           nombre + " y su edad es " + edad + " años.");
    }
}

```

```
        nombre + " y su edad es " + edad + " años.");  
    }  
}
```

La salida:

The screenshot shows the NetBeans IDE interface with the following details:

- Title Bar:** "Prueba JOptionPane.java" is displayed.
- Code Editor:** The Java code for "PruebaOptionPane.java" is shown, containing code to prompt for a name and age, and then print them together.
- Output Window:** Shows the output "Mascota (run-single)" with the message "Edad es = Apellidos".
- Running Application:** A small window titled "Mascota" is visible in the foreground, displaying the text "Edad es = Apellidos".

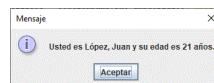
Mostrar Información

También podemos mostrar información en una ventana mediante el método `showMessageDialog`:

- `showMessageDialog(padre, mensaje)`: Muestra el mensaje como una ventana hija de la ventana padre. Si ponemos `null` en `padre` el mensaje se muestra de manera independiente y centrado en pantalla
 - `showMessageDialog(padre, mensaje, titulo, tipo)`: Muestra el mensaje con el título indicado. El tipo determina el ícono que se muestra y puede ser una de las siguientes constantes:

Tipo		Icono
JOptionPane.INFORMATION_MESSAGE		
JOptionPane.QUESTION_MESSAGE		
JOptionPane.WARNING_MESSAGE		
JOptionPane.ERROR_MESSAGE		
JOptionPane.PLAIN_MESSAGE		

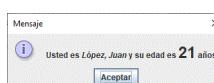
Si cambiamos el System.out por: JOptionPane.showMessageDialog(null, "Usted es " + apellidos + ", " + nombre + " y su edad es " + edad + " años."); La salida:



El texto a mostrar no tiene en cuenta saltos de tabulación ('\t') aunque sí saltos de línea ('\n')

Dado que se muestra como HTML podemos emplear algunas etiquetas básicas.

```
JOptionPane.showMessageDialog(null, "Usted es <i>" + apellidos + ", " + nombre + "</i> y su edad es <big>" + edad + "</big> años.</html>"); La salida:
```



Pedir Confirmación

En muchas ocasiones **lo que necesitamos es pedir confirmación** al usuario. Para ello la clase `JOptionPane` dispone del método `showConfirmDialog` en diferentes versiones. Este método **muestra un cuadro de diálogo con dos o tres botones** (por defecto Sí, No y Cancelar) para que el usuario **confirme o no una acción**. En función del valor pulsado retorna un entero que coincide con alguna de las constantes:

- `JOptionPane.YES, JOptionPane.OK`: El usuario ha confirmado la pregunta
 - `JOptionPane.NO, JOptionPane.CANCEL`: El usuario ha rechazado la pregunta (o el usuario ha cerrado el cuadro de diálogo)

Las versiones más habituales son:

- `JOptionPane.showConfirmDialog(padre, mensaje);`: Muestra el mensaje indicado con las opciones `Sí`, `No` y `Cancelar`.

- `JOptionPane.showConfirmDialog(padre, mensaje, titulo, opciones)`: Muestra el mensaje con las opciones indicadas. Las opciones es un entero que puede tener los siguientes valores:
 - `JOptionPane.YES_NO_CANCEL_OPTION`: Muestra los botones Si, No y Cancelar
 - `JOptionPane.YES_NO_OPTION`: Muestra los botones Si y No
 - `JOptionPane.OK_CANCEL_OPTION`: Muestra los botones Aceptar y Cancelar



1.14. Actividad 2-3

Actividad 2-3

Codificar y probar en JAVA los siguientes problemas

1.

Crear una clase `Persona` que disponga de una propiedad (`nombre`) para almacenar el nombre, otra (`peso`) para almacenar el peso en kg y otra (`altura`) para almacenar la altura en centímetros.

Además tendrá un método que retornará el peso en libras

una libra equivale a 0,453592 kg

y otro que retornará el IMC (Índice de Masa Corporal)

El IMC se calcula dividiendo el peso (en Kg) por la altura (en metros) elevada al cuadrado (multiplicada por sí misma).

Diseñar una aplicación que pida al usuario el nombre, peso y la altura de una persona y muestre su nombre, su altura en libras y su IMC.

Salida

Si introducimos a Carlos López con un peso de 73,3 kilos y una altura de 1,72, la salida debería ser similar a la siguiente:

```
Nombre : Carlos López
Peso (kg) : 73.300000
Peso (libras) : 161.598970
Altura : 1,720000
I.M.C. : 24.776906
```

Avanzado : Mostrar por pantalla el estado de la persona en función del IMC según la siguiente tabla:

IMC	Nivel de peso
Por debajo de 18.5	Bajo peso
18.5 – 24.9	Normal
25.0 – 29.9	Sobrepeso
30.0 o más	Obesidad

En el caso anterior debería mostrar:

Nivel de peso : Normal

Podéis comprobar los resultados en https://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmi-m.htm.

2.

Crear una clase `Calculadora` que tenga dos propiedades enteras (`x1` y `x2`) y que disponga de métodos para calcular su suma, su resta, multiplicación y división (con decimales). Hacer un programa que cree una calculadora, pida los dos valores al usuario y muestre el resultado de evaluar todos los métodos.

Salida

Si introducimos 17 y 8 como valores de `x1` y `x2` la salida debería ser similar a la siguiente

```
17 + 8 = 25
17 - 8 = 9
17 * 8 = 136
17 / 8 = 2.125000
```

3.

Crear una clase `Caja` para una tienda. La clase tendrá una propiedad `ingresos` y otra `gastos`. Además dispondrá de un método `ingresar(double cantidad)` que recibirá la cantidad de dinero a ingresar y la `sumará a los ingresos` (es decir, los `ingresos` tras llamar al método deben ser los que había más la `cantidad` recibida). La clase tendrá además otro método `gastar(double cantidad)` que será similar para los `gastos`. Por último tendrá un método `saldo()` que `retornará los ingresos menos los gastos`. Crear una caja y pedir al usuario dos cantidades a ingresar y dos a gastar. tras ello mostrar los `ingresos`, `gastos` y el `saldo` actual.

Salida

Si introducimos 100 y 200 como `ingresos` y 75 y 99 como `gastos`, la salida debería ser similar a la siguiente:

Ingresos : 300€
Gastos : 174€
Saldo Actual : 126€

4.

Hacer un programa (sólo el método `main`) que **recoja dos valores del teclado en dos variables diferentes** (`a` y `b`). Tras ello **intercambiará sus contenidos y finalmente los visualizará**. Es decir si tenemos dos variables `a` y `b` con diferentes valores, tras ejecutar el programa lo que había en `a` debería estar en `b` y viceversa..

Salida

Si introducimos 10 en `a` y 20 en `b` la salida debería ser similar a la siguiente:

Antes del intercambio
`a = 10`
`b = 20`

Después del intercambio
`a = 20`
`b = 10`

1.15. Ejercicios Adicionales

1.

Crear una clase `Rectangulo` que disponga de dos propiedades (`base` y `altura`) para almacenar las **dimensiones de un rectángulo**. Además, tendrá **métodos** para calcular el **área** y el **perímetro** del rectángulo. Diseñar una aplicación que pida al usuario la `base` y la `altura` de un rectángulo y muestre su `área` y `perímetro`.

Salida

Si introducimos 5 como `base` y 3 como `altura`, la salida debería ser similar a la siguiente:

Base: 5.0
Altura: 3.0
Área: 15.0
Perímetro: 16.0

2.

Crear una clase `Circulo` que tenga una propiedad `radio` y métodos para calcular el **área** y la **circunferencia** del círculo. El **área** se calcula como $\pi * radio^2$ y la **circunferencia** como $2 * \pi * radio$. Diseñar una aplicación que pida al usuario el `radio` de un círculo y muestre su `área` y `circunferencia`.

Salida

Si introducimos 4 como `radio`, la salida debería ser similar a la siguiente:

Radio: 4.0
Área: 50.265482
Circunferencia: 25.132741

3.

Crear una clase `Estudiante` que tenga propiedades para el `nombre`, la `nota de matemáticas`, la `nota de ciencias` y la `nota de historia`. La clase debe tener un método para calcular el **promedio** de las notas. Diseñar una aplicación que pida al usuario el `nombre` y las tres notas de un estudiante y muestre su `promedio`.

Salida

Si introducimos "Ana Pérez" como `nombre`, 85 como `nota de matemáticas`, 90 como `nota de ciencias` y 78 como `nota de historia`, la salida debería ser similar a la siguiente:

Nombre: Ana Pérez
Promedio: 84.333333

4.

Crear una clase `Pelicula` que tenga propiedades para el `título`, el `director` y la `duración` en minutos. La clase debe tener un método que **convierta la duración de minutos a horas y minutos**. Diseñar una aplicación que pida al usuario el `título`, el `director` y la `duración` de una película y muestre la `duración` en horas y minutos.

Salida

Si introducimos "Aventura Espacial" como `título`, "Ana López" como `director`, y 130 como `duración`, la salida debería ser similar a la siguiente:

Título: Aventura Espacial
Director: Ana López
Duración: 130 minutos
Duración en Horas y Minutos: 2 horas y 10 minutos

5.

Crear una clase `Bebida` que tenga propiedades para el `nombre`, el `volumen` en mililitros y el `porcentaje` de alcohol. La clase debe tener un método que calcule la **cantidad de alcohol puro en mililitros** (`volumen * porcentaje de alcohol / 100`). Diseñar una aplicación que pida al usuario el `nombre`, el `volumen` y el `porcentaje` de alcohol de una

bebida y muestre la cantidad de alcohol puro.

Salida

Si introducimos "Cerveza" como nombre, 500 como volumen, y 5 como porcentaje de alcohol, la salida debería ser similar a la siguiente:

```
Nombre: Cerveza
Volumen: 500 ml
Porcentaje de Alcohol: 5%
Alcohol Puro: 25 ml
```

1.16. Estructuras de Control Secuencial y Alternativa

Instrucciones y bloques de instrucciones en Java

Una instrucción es una unidad de ejecución de un programa. Toda instrucción en Java termina con un punto y coma (;). Existen los siguientes tipos de instrucciones:

- **Instrucciones de expresión.** Ejecutan una expresión. Se subdividen en:
- **Instrucciones de asignación.** Asignan un valor a una variable:

```
java valor = 12; suma = suma + valor;
```

- **Instrucciones de incremento.** Incrementan / decrementan una variable:

```
java valor++; --numero;
```

- **Instrucciones de llamada a un método.** Ejecutan un método de un objeto o clase:

```
java System.out.println("Hola");
```

- **Instrucciones de creación de objetos.**

```
java Integer i = new Integer(12);
```

- **Instrucciones de declaración.** Se emplean para declarar variables.

```
java double d; Integer i,j; float f = 12.5F;
```

- **Instrucciones de control de flujo.** Controlan el modo en el que se ejecutan las instrucciones. Las veremos a continuación.

- **Instrucciones de control de excepciones.** Permiten gestionar las excepciones que se pueden producir en el código. Las veremos en detalle más adelante.

Programación Estructurada

Como ya se ha comentado previamente, las aplicaciones modernas se basan en el uso de estructuras de control. Las estructuras de control determinan la forma en la que se van a ejecutar las instrucciones de un programa y básicamente se dividen en tres grandes grupos:

- **Secuencial:** Todas las instrucciones en esta estructura se ejecutan una detrás de otra en el orden que aparecen.
- **Alternativas:** Se ejecutan una serie de instrucciones u otras en función de que se cumpla una condición.
- **Repetitivas:** Se ejecutan varias veces las mismas instrucciones.

Secuencial

En Java la estructura secuencial puede ser una única instrucción finalizada en un ; o un conjunto de instrucciones encerradas entre llaves:

```
instrucción;
{
    instrucción1;
    instrucción2;
    ...
}
```

Una estructura secuencial se puede poner en cualquier sitio donde pueda ir una instrucción en Java.

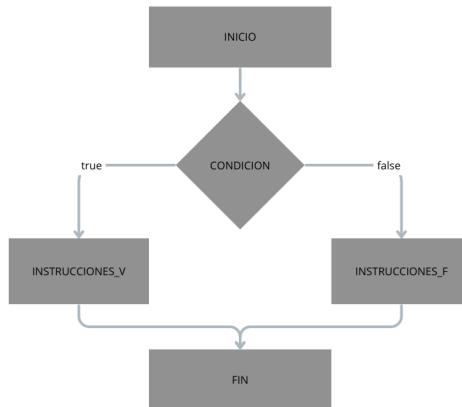
NOTA: Las llaves en Java también sirven para definir los límites del código de una clase o de un método de una clase.

Alternativa simple

Determina si se ejecuta una instrucción o no en función de una condición:

```
if (condición)
    instrucción_verdadero;
[else
    instrucción_falso;
]
```

La parte `else` es opcional. Si deseamos ejecutar varias instrucciones deberemos crear una estructura secuencial.



```

public class Main {
    public static void main(String[] args) {
        int cantidad = 120;
        double precio = 50.10;
        final double DTO = 0.05;

        System.out.println("Compra de " + cantidad + " unidades a " + precio + "€.");

        if (cantidad > 30) {
            System.out.println("Enhorabuena!. Por comprar más de 30 unidades se aplica un descuento del 5%");
            precio = precio * (1 - DTO);
            System.out.println("Nuevo Precio : " + precio);
        } else {
            System.out.println("Lástima!. No ha comprado suficientes unidades para tener un descuento");
        }

        double importe = cantidad * precio;
        System.out.println("Importe a Pagar : "+ importe);
    }
}
  
```

La salida de la aplicación:

```

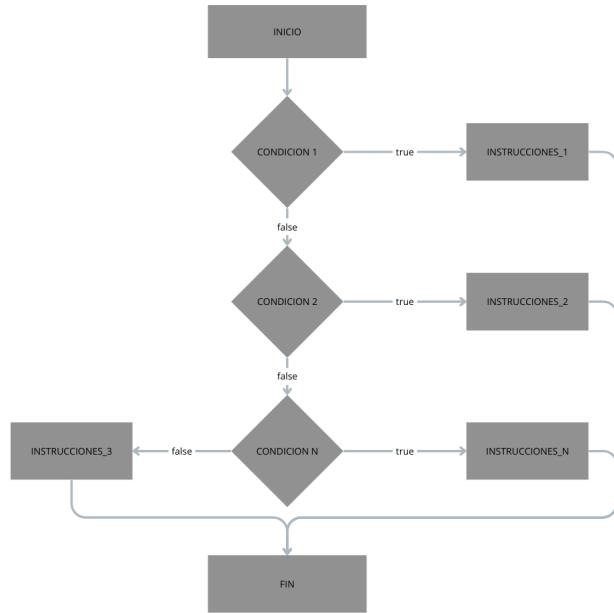
Compra de 120 unidades a 50.10.
Enhorabuena!. Por comprar más de 30 unidades se aplica un descuento del 5%
Nuevo Precio : 47.595
Importe a Pagar : 5711.4
  
```

Alternativa múltiple

En una alternativa múltiple se pueden comprobar varias condiciones. En su forma más completa se ejecuta mediante if anidados:

```

if (condición1)
    instrucción1;
else if (condición2)
    instrucción2;
[else if (condición3)
    instrucción3]
[else
    instrucción_falso;
]
  
```



NOTA: También es posible obtener el mismo resultado incluyendo alternativas dentro de otra alternativa:

```

public static void main(String[] args) {
    int edad = 17;
    String categoria;

    if (edad < 12){
        categoria = "Alevín";
    } else {
        if (edad < 16){
            categoria = "Infantil";
        } else {
            if (edad < 21){
                categoria = "Juvenil";
            } else {
                categoria = "Senior";
            }
        }
    }
    System.out.println("Con alternativas anidadas\nCategoria : " + categoria);

    if (edad < 12){
        categoria = "Alevín";
    } else if (edad < 16){
        categoria = "Infantil";
    } else if (edad < 21){
        categoria = "Juvenil";
    } else {
        categoria = "Senior";
    }
    System.out.println("Con alternativas múltiples\nCategoria : " + categoria);
}

```

La salida:

```

Con alternativas anidadas
Categoria : Juvenil
Con alternativas múltiples
Categoria : Juvenil

```

Si lo que tenemos que hacer es seleccionar uno entre varios posibles valores (que deben ser de tipo byte, char, short, int o String), podemos emplear la instrucción switch:

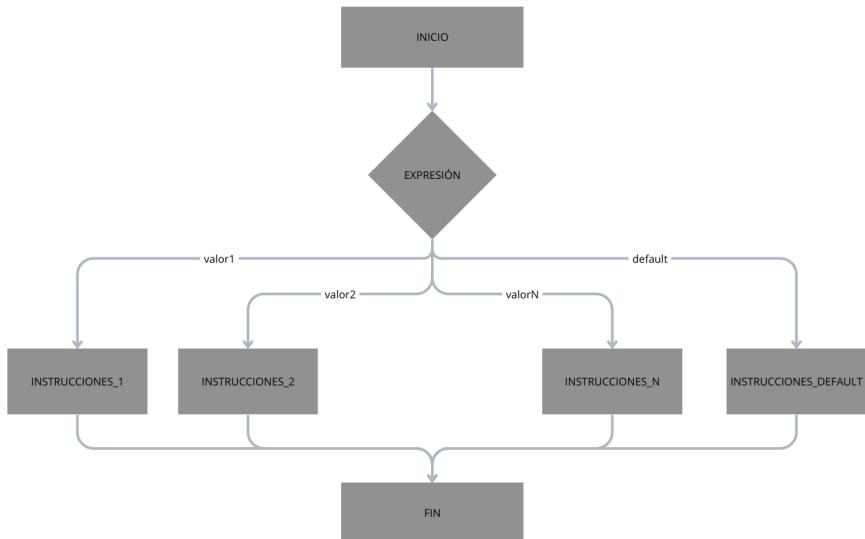
```

switch (variable) {
    case valor1:
        instrucciones1;
        break;
    case valor2:
        instrucciones2;
        break;
    default:
        instrucciones;
}

```

Se compara la variable con los diferentes valores empezando por el primero. Si coincide con alguno se ejecuta la instrucción asociada y continúa ejecutando instrucciones hasta encontrar una cláusula break o el final de la estructura.

default es la opción que se ejecuta si la variable no toma ninguno de los valores anteriores (puede ir en cualquier sitio de la estructura).



```

public class Main {
    public static void main(String[] args) {
        int mes = 2;
        boolean bisiesto = true;
        int dias;

        switch(mes) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
                dias = 31;
                break;
            case 2:
                dias = bisiesto ? 29 : 28;
                break;
            default:
                dias = 30;
                break;
        }

        System.out.println("El mes " + mes + " tiene " + dias + " días");
    }
}
  
```

La salida:

El mes 2 tiene 29 días

A partir de la versión 12 de JAVA se incluye una nueva forma de emplear los switchs.

Switch Expressions

En este caso, el switch retorna lo indicado en valorRetorno por lo que su valor debe asignarse a una variable para utilizarlo:

```

switch (variable) {
    case listaValores1 -> valorRetorno1;
    case listaValores2 -> valorRetorno2;
    default -> valorRetornoN;
}

public class NewClass {
    public static void main(String[] args) {
        /* A partir de la versión 12 de Java */
        int mes = 2;
        boolean bisiesto = true;
        int dias;

        dias = switch(mes){
            case 1, 3, 5, 7, 8 -> 31;
            case 2 -> bisiesto ? 29 : 28;
            default -> 30;
        };

        System.out.println("El mes " + mes + " tiene " + dias + " días");
    }
}
  
```

La salida:

El mes 2 tiene 29 días

En Java 13 se añadió la posibilidad de meter bloques de código. En caso de hacerlo el valor a retornar se debe poner después de una instrucción `yield`:

```
public class NewClass {
```

```

public static void main(String[] args) {
    /* A partir de la versión 13 de Java (yield) */
    int mes = 2;
    boolean bisiesto = true;
    int dias;

    dias = switch(mes){
        case 1, 3, 5, 7, 8 -> 31;
        case 2 -> {
            if (bisiesto) {
                yield 29;
            } else {
                yield 28;
            }
        }
        default -> 30;
    };

    System.out.println("El mes " + mes + " tiene " + dias + " días");
}
}

```

Evidentemente, la salida es la misma del caso anterior.

1.17. Actividad 3 - Alternativas

Instrucciones

Realizar los siguientes ejercicios:

1.

Hacer un programa (sin clase adicional) que **pida dos números al usuario y mostrar por pantalla el MAYOR de los dos**.

2.

Modificarlo para indicar, en el caso que suceda, **cuando ambos números sean iguales**

3.

Para un bar nos piden diseñar una clase llamada **Pintxo** que tendrá una propiedad **precio** de tipo **double**. Además deberemos definir un método (**porcDto(int cantidad)**) que **calcule el descuento a aplicar en función de la cantidad** de pintxos pedidos. Si la **cantidad** es de **5 pintxos o más** se retornará **5** (un **5%**) y, si es **más de 10 se retomará un 10**. Además se definirá otro método **importe(int cantidad)** que **retornará el importe de vender esa cantidad de pinchos teniendo en cuenta el posible descuento***.

Diseñar un **programa** que cree un objeto de tipo **Pincho** con un **precio** de **2,5€** (asumimos que todos los pinchos valen lo mismo), pida la **cantidad** de pintxos pedidos y muestre por pantalla el **porcentaje de descuento** y el **importe** de dicha venta.

Adicional

Modificar el programa para que el **descuento sólo si muestra si lo hay**.

4.

Crear un programa que **pida tres números** al usuario y muestre cuál es el **MENOR** de los tres

5.

Diseñar una clase **Alumno** que disponga de los siguientes **campos**:

- **nombre** : El nombre del alumno
- **notaENT, notaFOL, notaSI, notaBD, notaPRO, notaIT y notaLM**: las notas (enteros) en cada módulo

Definir en la clase los siguientes **métodos**:

- **double notaMedia()** : Retornará la media del alumno con decimales
- **int modulosAprobados()** : El número de módulos aprobados
- **int modulosSuspensados()** : El número de módulos suspendidos
- **boolean pasaCurso()** : Si el alumno pasa (true) o no (false) de curso. **NO** pasa de curso si tiene **más de dos módulos suspendidos o si el número de horas básicas es mayor de 300 horas**

Módulo	Horas Básicas
ENT	99
IT	33
SI	165
BD	198
PRO	264

Módulo	Horas Básicas
LM	132
FOL	99

Diseñar un programa que pida el **nombre** de un alumno y sus **notas** en todos los módulos. Tras ello mostrar su nota **media**, **aprobados**, **suspensos** y si **pasa de curso o no**.

6.

Diseñar una aplicación que pida al usuario el **año**, el **número del mes** como un número de 1 a 12 (Enero a Diciembre) y muestre el número de **días** de dicho mes. Tener en cuenta que **un mes es bisiesto si es divisible por 4 y no divisible por 100 o es divisible por 400**.

7.

En una empresa de transporte quieren diseñar una clase **Envío** que tenga los siguientes campos:

- **tipo**: Un entero que representa el **tipo de envío** como un valor entre 1 y 3.
- **peso**: Un double que representa el **peso en kilos** del envío

Además la clase dispondrá de los siguientes métodos:

-String **getTipoEnvio()**: que retornará el **tipo** de envío como un **texto** según la siguiente tabla:

tipo	
1	Normal
2	Expres
3	Rápido

- double **getPrecioKilo()**: que retornará el **precio** por kg de envío según la tabla:

tipo	
1	1,5 €/kg
2	2,0 €/kg
3	4,0 € / kg

- double **getImporte()**: retornará el **importe** del envío en función del **peso** y del **precio** por **kilo**.

Diseñar un programa que **pida** el **tipo** de envío (suponemos que es un valor entre 1 y 3) y el **peso** del envío, cree un objeto **Envío** con ellos y muestre por pantalla los **kilos**, el **tipo** de envío seleccionado (como **texto**), el **precio** por kilo y el **importe** del mismo.

Adicional

Modificar el código para que, en el caso del envío **Rápido**, se aplique un **descuento del 5% a partir de los 5 kg**.

1.18. Estructuras de Control Repetitivas

Repetitivas

Dentro de las **repetitivas**, disponemos de dos tipos de bucles: el **while** y el **for**.

El **bucle while** dispone de dos sintaxis:

Sintaxis 1:

```
while (condición)
    instrucción;
```

Si la **condición** se cumple, se ejecuta la instrucción (o el bloque de instrucciones si se ha puesto uno) y se vuelve a comprobar la **condición**. Si se sigue cumpliendo, se repite el proceso. A estas repetitivas se les denomina **repetitivas de 0 a n**, puesto que puede que no se ejecute nunca la instrucción (si la **condición** no se cumple la primera vez).

Sintaxis 2:

```
do
    instrucción;
while (condición);
```

En este caso, primero se ejecuta la instrucción y luego se comprueba la **condición**. Si se cumple, se repite el proceso. Es una **repetitiva de 1 a n** porque la instrucción al menos se ejecuta una vez (aunque la **condición** no se cumpla la primera vez).

Por su parte, la sintaxis del **bucle for** es la siguiente:

```
for (expresión1; condición; expresión2)
    instrucción;
```

Se comienza evaluando **expresión1** y después la **condición**. Si la **condición** se cumple, se ejecuta la instrucción. Tras ello, se evalúa **expresión2** y luego de nuevo la

condición. Es otra **repetitiva de 0 a n**. Habitualmente, **expresión1** se emplea para inicializar una (o más) variable(s), **expresión2** la(s) incrementa y la **condición** compara el valor de la variable con un límite determinado. Tanto en **expresión1** como en **expresión2** se pueden poner varias expresiones separadas por comas.

No es necesario incluir todas las partes de la instrucción. Por ejemplo, los siguientes bucles serían válidos:

```
for(; a < 20; a++);  
for(; a < 20;  
// Equivale a while (a < 20)  
for(;;)  
// En principio es un bucle infinito
```

Existe otra versión del **for** que permite recorrer colecciones de objetos y matrices:

```
for (var : colecciónOmatriz)  
    instrucción;
```

Se va recorriendo la colección de elementos y se va asignando cada uno de ellos a la variable **var**.

Usaremos las colecciones de elementos en una unidad didáctica más adelante.

Instrucciones de salto incondicional

Las **instrucciones de salto incondicional** provocan un salto en el flujo del programa.

break

La instrucción **break** puede llevar o no una **etiqueta** detrás. Cuando no lleva etiqueta, provoca un salto fuera del **switch** o de la repetitiva (**for** o **while**) en la que se encuentra (no afecta a las alternativas). Si lo que queremos es salir de más de un bucle anidado, deberemos emplear una etiqueta (**etiqueta:**) delante del bucle al que queremos salir y emplear el comando **break etiqueta**.

continue

La instrucción **continue** es similar a la instrucción **break**. La diferencia es que **continue** salta al final del bucle en el que se encuentra, forzando un recálculo de la **condición** e impidiendo que se ejecuten las instrucciones tras el **continue**. Al igual que en el caso del **break**, el **continue** puede llevar o no etiqueta.

return

La instrucción **return** provoca una salida del método en el que se encuentra, regresando a la instrucción siguiente a la que invocó dicho método. La instrucción **return** puede opcionalmente devolver un valor. Se verá más en detalle cuando veamos los métodos.

Ejemplo en Java con las estructuras mencionadas

A continuación se muestra un ejemplo sencillo en forma de una clase con un método **main** ejecutable, que demuestra el uso de los bucles **while**, **do while**, **for**, **for each** y las instrucciones **break**, **continue**, y **return**.

```
public class EjemplosRepetitivas {  
    public static void main(String[] args) {  
        // Ejemplo de bucle while (0 a n)  
        System.out.println("==== Bucle while (0 a n) ====");  
        int i = 0;  
        while (i < 3) {  
            System.out.println("while: i vale " + i);  
            i++;  
        }  
  
        // Ejemplo de bucle do while (1 a n)  
        System.out.println("\n==== Bucle do while (1 a n) ====");  
        int j = 0;  
        do {  
            System.out.println("do while: j vale " + j);  
            j++;  
        } while (j < 3);  
  
        // Ejemplo de bucle for (0 a n)  
        System.out.println("\n==== Bucle for (0 a n) ====");  
        for (int k = 0; k < 3; k++) {  
            System.out.println("for: k vale " + k);  
        }  
  
        // Ejemplo de bucle for each para recorrer un array  
        System.out.println("\n==== Bucle for each ====");  
        String[] frutas = {"Manzana", "Pera", "Naranja"};  
        for (String fruta : frutas) {  
            System.out.println("for each: " + fruta);  
        }  
  
        // Ejemplo de uso de break  
        System.out.println("\n==== Uso de break ====");  
        for (int x = 1; x <= 5; x++) {  
            if (x == 3) {  
                System.out.println("Se ha encontrado x == 3, saliendo del bucle.");  
                break;  
            }  
            System.out.println("Valor de x: " + x);  
        }  
  
        // Ejemplo de uso de continue  
        System.out.println("\n==== Uso de continue ====");
```

```

for (int y = 1; y <= 5; y++) {
    if (y == 3) {
        System.out.println("Se ha encontrado y == 3, saltando al siguiente ciclo.");
        continue;
    }
    System.out.println("Valor de y: " + y);
}

// Ejemplo de uso de return dentro de un método
System.out.println("\n==== Uso de return ===");
System.out.println("Valor devuelto por el método ejemploReturn(): " + ejemploReturn());
// Tras esto, el flujo vuelve a la instrucción siguiente, que imprime un mensaje final
System.out.println("Fin del método main, tras llamar a ejemploReturn().");
}

// Método que demuestra uso de return
private static int ejemploReturn() {
    int resultado = 42; // Simulamos algún cálculo
    return resultado; // Al llamar a return, se sale de este método
}

```

En este ejemplo, se puede observar cómo cada tipo de bucle (`while`, `do while`, `for` y `for each`) se comporta de forma diferente en la ejecución, así como el efecto que tienen las instrucciones `break` y `continue` en el flujo de un bucle, y la instrucción `return` para salir de un método.

1.19. Actividad 4 - Repetitivas

Instrucciones

1.

Pedir al usuario un número y mostrar su tabla de multiplicar. Por ejemplo, para el número 5 la salida debería ser:

```

5 x 1 : 5
5 x 2 : 10
...
5 x 9 : 45
5 x 10 : 50

```

Hacerlo con un `while`, con un `do while` y con un `for`.

2.

Mostrar la tabla de multiplicar de todos los números del 1 al 10

3.

Calcular el **factorial** de un número sabiendo que:

```
n! = n * (n - 1) * (n - 2) ... * 3 * 2
```

Por ejemplo: $4! = 4 \cdot 3 \cdot 2 \cdots \cdot 1 = 24$ El programa deberá pedir un número entero al usuario y mostrar el factorial. Por ejemplo:

```
El factorial de 10 es 10!=3628800
```

4.

Dado un **capital** a invertir, un **porcentaje de interés anual** (que se pagará **mensualmente**) y un número de **años** se pide mostrar por pantalla el **resultado de dicha inversión mensualmente**. Para **cada mes** se deberá calcular el **beneficio** obtenido en dicho mes (el **capital actual** por el **interés mensual**). Dicho **beneficio** se **acumulará** sobre el **capital actual** (de modo que el **siguiente mes el beneficio obtenido será un poco mayor**). Por ejemplo, para un capital de 1000€ al 1,5% de interés anual en 2 años la tabla empezaría como la siguiente:

```

Datos de la Inversión=====
Capital Inicial : 1000.0e.
Interés Anual : 1.5%
Años : 2

Mes Beneficio Capital Fina
1--- -----
1 1,250 1001,250
2 1,252 1002,502
3 1,253 1003,755
4 1,255 1005,009
...
24 1,286 1030,435

```

5.

Diseñar un programa que permita calcular **estadísticas de las notas** de una serie de alumnos en un examen. Se quiere saber, al final, **cuántos alumnos se han examinado, cuántos han aprobado y la nota media de todos ellos**.

El proceso a realizar será el siguiente:

- Inicializar variables para guardar los resultados (cuántos alumnos hay, cuántos han aprobado y la suma de sus notas)
- **Repetir** el siguiente proceso
- Pedir la nota de un alumno

- **Contar** un alumno más y, si **ha aprobado**, contar un **aprobado** más
- **Sumar** su nota a la **suma de notas**
- **Mientras** el usuario diga que quedan más alumnos (**showConfirmDialog**)
- Mostrar los **resultado** obtenidos

1.20. Ejemplos Prácticos de Uso de Repetitivas

Estructuras de Control Repetitivas en Java

Repetir un proceso un número determinado de veces (for)

Cuando **sabemos cuántas veces se va a repetir un proceso**, la estructura más adecuada suele ser el **for** dado que nos permite iterar tantas veces como queramos. Veamos algunos ejemplos:

Sabemos exactamente cuántas veces vamos a repetir el proceso

Enunciado

Supongamos que nos piden calcular la media de ingresos de una empresa en un año y nos van a proporcionar los ingresos mensuales.

Análisis

En este caso es evidente que tenemos que pedir doce ingresos e irlos sumando lo que implica que necesitamos una repetitiva. Como sabemos que se va a repetir exactamente 12 veces, emplearemos un **for**. ¿Qué es lo que debemos repetir? : pedir los ingresos y acumularlos sobre un total.

Para hacer el programa necesitamos emplear un acumulador. Veamos lo que es:

Contadores y acumuladores

Un contador es una variable que permite cuantificar cuántas veces se repite un proceso, mientras que un acumulador nos permite calcular un total calculado a partir de diferentes valores. El funcionamiento de ambos es muy similar (de hecho los contadores son un caso particular de acumuladores en los que el incremento en cada paso del proceso es 1):

- Inicializar el contador o acumulador a 0 antes del proceso a repetir
- Cada vez que se repita el proceso (dentro de la repetitiva) incrementar el contador en 1 o sumar lo que queremos totalizar al acumulador
- Una vez fuera de la repetitiva, tendremos en las variables los valores buscados

NOTA: Es bastante habitual que contadores y acumuladores estén asociados a condiciones. En este caso irán dentro de una alternativa (cuántos alumnos han aprobado, el total de ventas de un determinado vendedor, cuántas facturas impagadas tenemos...)

Solución

Para este ejemplo, supondremos que vamos a pedir los datos directamente al usuario por lo que no vamos a crear ninguna clase:

```
import javax.swing.JOptionPane;

public class MediaIngresos {
    public static void main(String[] args) {
        // ----- ESTO SE HACE 1 VEZ -----
        double totalIngresos = 0; // Acumulador

        // Repetimos el proceso 12 veces (una por mes)
        for (int mes = 1; mes <= 12; mes++) {
            // ----- TODO ESTO SE HACE 12 VECES -----
            // Vamos a pedir los ingresos de la empresa en el mes indicado
            // Para facilitarle al usuario introducir la información vamos a mostrar el nombre del mes
            // Obtenemos el nombre del mes con una expresión switch
            String nombreMes = switch(mes){
                case 1 -> "Enero";
                case 2 -> "Febrero";
                case 3 -> "Marzo";
                case 4 -> "Abril";
                case 5 -> "Mayo";
                case 6 -> "Junio";
                case 7 -> "Julio";
                case 8 -> "Agosto";
                case 9 -> "Septiembre";
                case 10 -> "Octubre";
                case 11 -> "Noviembre";
                default -> "Diciembre";
            };

            // Pedimos los ingresos del mes en una variable temporal
            String resp = JOptionPane.showInputDialog("Ingresos en " + nombreMes);
            // Convertimos la respuesta a un double
            double ingresos = Double.parseDouble(resp);
            // Y los sumamos al total (acumulador)
            totalIngresos += ingresos; // También totalIngresos = totalIngresos + ingresos
        }

        // ----- ESTO SE HACE 1 VEZ -----
        // Calculamos la media de ingresos
        double mediaIngresos = totalIngresos / 12;
        // Y mostramos el resultado
        JOptionPane.showMessageDialog(null, "La media de ingresos de los 12 meses ha sido : " + mediaIngresos);
    }
}
```

Nos dicen el número de veces que vamos a repetir el proceso

En el caso anterior sabíamos exactamente el número de veces que se iba a repetir el proceso. Pero el for también podemos usarlo cuando nos dicen el número de veces que se debe repetir.

Enunciado

Supongamos que nos piden mostrar un listado de las notas de unos alumnos en un módulo. El programa deberá pedir, en primer lugar cuántos alumnos hay. Tras ello, pedirá los datos de cada alumno y los mostrará por pantalla. Al final se indicará el número y porcentaje de alumnos aprobados y la media del curso.

Análisis

- Tenemos que mostrar los aprobados (contador de aprobados), el porcentaje (además del contador anterior necesitamos el total de alumnos que es lo primero que tenemos que pedir) y la media del curso (necesitamos la suma de las notas de todos los alumnos y el total de alumnos). Por tanto, aparte del total de alumnos que lo pedimos al principio necesitamos un contador (aprobados) y un acumulador (sumaNotas)
- Como tenemos varios alumnos, necesitamos repetir el proceso tantas veces como alumnos. En este caso es tantas veces como indique el total de alumnos. En este caso un for es adecuado ya que podemos contar hasta una variable.
- En cada repetición (por tanto, dentro de la repetitiva) habrá que pedir los datos de un alumno, mostrarlos y acumular su nota a la suma de las notas
- Dado que nos piden contar los aprobados sólo incrementaremos el contador si el alumno está aprobado. Esto implica que tenemos que poner una alternativa dentro de la repetitiva.
- Cuando acabe la repetitiva mostramos los totales.

NOTA: Cuando se mete una estructura dentro de otra como en este caso hay que tener mucho cuidado con las llaves. La instrucción más interna (alternativa en este caso) debe estar completamente incluida dentro de la más externa (repetitiva). Por ello es importante emplear adecuadamente la indentación del código.

Solución

Vamos a crear primero la clase Alumno con los campos nombre, apellidos, nota y un método que retorne si ha aprobado o no:

```
public class Alumno {  
    public String nombre;  
    public String apellidos;  
    public int nota;  
  
    public boolean aprobado(){  
        return nota >= 5;  
    }  
}
```

Y ahora el programa principal:

```
import javax.swing.JOptionPane;  
  
/**  
 *  
 * @author IñigoChueca  
 */  
public class Notas {  
    public static void main(String[] args) {  
        // Inicialización de totales y acumuladores  
        int aprobados = 0;  
        int sumaNotas = 0;  
  
        // Pedimos el total de alumnos a procesar (FUERA de la repetitiva dado que SÓLO LO HACEMOS UNA VEZ  
        String resp = JOptionPane.showInputDialog("Número de Alumnos a Procesar");  
        int totalAlumnos = Integer.parseInt(resp);  
  
        // Vamos a mostrar los datos por consola  
        // En este paso podríamos mostrar la información que nos interese que aparezca ANTES de los datos del alumno  
        // Por ejemplo, las cabeceras  
        System.out.println("Listado de Alumnos\n");  
        System.out.println("Alumno\tNota\tAprobado");  
        System.out.println("-----\t-----\t-----");  
  
        // Ahora repetimos el proceso UNA VEZ POR CADA ALUMNO  
        for (int i = 0; i < totalAlumnos; i++) {  
            // Pedimos los datos;  
            String nombre = JOptionPane.showInputDialog("Nombre Alumno");  
            String apellidos = JOptionPane.showInputDialog("Apellidos Alumno");  
            resp = JOptionPane.showInputDialog("Nota (1-10)"); // Reutilizamos la variable resp  
            // Convertimos la respuesta a un entero  
            int nota = Integer.parseInt(resp);  
  
            // Creamos un nuevo alumno y le asignamos los datos  
            Alumno a = new Alumno();  
            a.nombre = nombre;  
            a.apellidos = apellidos;  
            a.nota = nota;  
  
            // Queremos mostrar Si / No si está o no aprobado  
            // Queremos sumar uno a aprobados SÓLO si está aprobado  
            // Podemos hacer las dos cosas a la vez:  
            String aprobado; // Aquí guardaremos "Si" o "No"  
            if (a.aprobado()) {  
                aprobado = "Sí";  
                // Hay un aprobado más  
                aprobados++; // aprobados = aprobados + 1;  
            } else {  
                aprobado = "No";  
            }  
            System.out.println(a.nombre + "\t" + a.nota + "\t" + aprobado);  
        }  
    }  
}
```

```

        aprobado = "No";
    }

    // Mostramos los datos del alumno
    System.out.println(a.apellidos + ", " + a.nombre + "\t" + a.notas + "\t" + a.aprobado);
    // Sumamos su nota al total
    sumaNotas += a.notas;
}

// Mostramos los totales
System.out.println("\nResumen");
System.out.println("Total Alumnos : " + totalAlumnos);
// Calculamos el porcentaje de aprobados
double porcAprobados = aprobados * 100.0 / totalAlumnos;
// Lo mostramos
System.out.println("Aprobados : " + aprobados + " (" + porcAprobados + "%)");
// Calculamos la media de notas (casting para obtener decimales)
double media = (double) sumaNotas / totalAlumnos;
// La mostramos
System.out.println("Nota media del curso : " + media);
}
}

```

Repetir un proceso en base a una condición (while / do while)

En este caso el proceso a repetir depende de que se cumpla o no una condición. La diferencia entre el while y el do-while está en que, en el primer caso, puede que el proceso no llegue a ejecutarse, mientras que en el segundo sí se ejecuta al menos una vez.

Pedir múltiples datos a un usuario hasta que introduzca un valor especial (while)

Enunciado

Se desea diseñar una aplicación que permita calcular el total de múltiples facturas. Se desea que el proceso sea lo más rápido posible por lo que el sistema irá pidiendo cada uno de los totales a sumar. Cuando se introduzca un valor 0 (valor bandera) significará que no hay más facturas con lo que se mostrará el total facturado, el número total de facturas y la media del importe por factura.

Análisis

En este caso tenemos una repetitiva simple que depende del valor que se pida al usuario. Además necesitamos un contador (para ver cuántas facturas se han procesado) y un acumulador (para ir incrementando con el total de cada factura). El proceso a repetir será pedir el total de la factura, contar una factura más y sumar el importe al total final.

Dado que la repetitiva se basa en una condición (si el total introducido por el usuario es 0 o no) se trata de un while o un do-while. ¿Cuál de ellos? Hay que plantearse si estamos seguros de que el proceso se va a hacer al menos una vez. ¿Qué pasa si cuando pedimos el primer total el usuario escribe un 0? Que el proceso debería finalizar. Esto implica que no se repetiría el proceso ninguna vez. Por tanto, la estructura adecuada sería un while.

Dado que el while necesita el primer total para la condición, habrá que pedirlo ANTES del mismo. Pero, una vez totalizado, habrá que pedirlo DE NUEVO (dentro de la repetitiva) dado que necesitamos que el usuario lo introduzca múltiples veces.

Solución

```

import javax.swing.JOptionPane;

public class FacturasWhile {
    public static void main(String[] args) {
        // Inicializamos totales
        double importeTotal = 0;
        int numeroFacturas = 0;

        // Pedimos el primer importe (Lo necesitamos para el while)
        String resp = JOptionPane.showInputDialog("Importe Factura (0 para finalizar)");
        double importe = Double.parseDouble(resp);

        // Repetimos mientras el importe NO SEA 0
        while (importe != 0){
            // Sumamos el importe al total
            importeTotal += importe;
            // Contamos una factura más
            numeroFacturas++;

            // Hay que pedir otra vez el NUEVO IMPORTE AL FINAL
            // Tiene que estar DENTRO de la repetitiva porque hay que pedirlo varias veces
            resp = JOptionPane.showInputDialog("Importe Factura (0 para finalizar)");
            importe = Double.parseDouble(resp);
        }

        // Salimos porque el importe es 0
        // Calculamos la media
        double mediaFactura = importeTotal / numeroFacturas;

        // Mostramos el resumen
        String listado = "<html><table border=1>";
        listado += "<tr><td colspan=2>Resumen Facturación</td></tr>";
        listado += "<tr><td>Facturas Emitidas</td><td>" + numeroFacturas + "</td></tr>";
        listado += "<tr><td>Importe Total</td><td>" + importeTotal + "</td></tr>";
        listado += "<tr><td>Importe Promedio</td><td>" + mediaFactura + "</td></tr>";
        listado += "</table></html>";

        JOptionPane.showMessageDialog(null,
            listado,
            "Resumen Facturación",
            JOptionPane.PLAIN_MESSAGE);
    }
}

```

```
}
```

Mejora

¿Qué pasa si el usuario introduce 0 la primera vez?

NOTA: NaN significa Not a Number y es la forma que tiene Java de indicar que la operación retorna un resultado que no es numérico (al dividir entre 0 el resultado es infinito y no hay forma de representarlo).

Podemos mejorar la aplicación haciendo que, si no hay facturas, muestre un mensaje más claro. En este caso tenemos que poner una alternativa tras salir de la repetitiva para ver si el proceso se ha repetido o no. En caso de que no se haya repetido sabremos que importeTotal y numeroFactura serán 0, así que podemos comprobar cualquiera de las dos.

```
import javax.swing.JOptionPane;

public class FacturasWhile {
    public static void main(String[] args) {
        // Inicializamos totales
        double importeTotal = 0;
        int numeroFacturas = 0;

        // Pedimos el primer importe (Lo necesitamos para el while)
        String resp = JOptionPane.showInputDialog("Importe Factura (0 para finalizar)");
        double importe = Double.parseDouble(resp);

        // Repetimos mientras el importe NO SEA 0
        while (importe != 0){
            // Sumamos el importe al total
            importeTotal += importe;
            // Contamos una factura más
            numeroFacturas++;

            // Hay que pedir otra vez el NUEVO IMPORTE AL FINAL
            // Tiene que estar DENTRO de la repetitiva porque hay que pedirlo varias veces
            resp = JOptionPane.showInputDialog("Importe Factura (0 para finalizar)");
            importe = Double.parseDouble(resp);
        }

        if (numeroFacturas > 0){
            // Salimos porque el importe es 0
            // Calculamos la media
            double mediaFactura = importeTotal / numeroFacturas;

            // Mostramos el resumen
            String listado = "<html><table border=1>";
            listado += "<tr><td colspan=2>Resumen Facturación</td></tr>";
            listado += "<tr><td>Facturas Emisiones</td><td>" + numeroFacturas + "</td></tr>";
            listado += "<tr><td>Importe Total</td><td>" + importeTotal + "</td></tr>";
            listado += "<tr><td>Importe Promedio</td><td>" + mediaFactura + "</td></tr>";
            listado += "</table></html>";

            JOptionPane.showMessageDialog(null,
                listado,
                "Resumen Facturación",
                JOptionPane.PLAIN_MESSAGE);
        } else {
            JOptionPane.showMessageDialog(null,
                "No se ha procesado ninguna factura",
                "Sin Facturas",
                JOptionPane.INFORMATION_MESSAGE);
        }
    }
}
```

Repetir un proceso mientras el usuario quiera (do-while)

Enunciado

Queremos hacer el mismo listado de alumnos que en el segundo ejemplo (el de notas). Pero, en este caso, en lugar de pedirle al usuario que nos diga cuántos alumnos hay le vamos a ir pidiendo los datos alumno a alumno. Cuando acabemos con un alumno le preguntaremos si quiere procesar el siguiente alumno. Cuando diga que no, mostraremos el resumen.

Análisis

En este caso, el enunciado indica que pedimos los datos de un alumno, los procesamos y luego preguntamos al usuario si tiene más datos. Mientras diga que sí repetimos el proceso. Es evidente que la repetición depende de una condición (que el usuario confirme que hay más alumnos) y que es un do while dado que, como mínimo, se procesa un alumno.

Solución

```
import javax.swing.JOptionPane;

public class Notas2 {
    public static void main(String[] args) {
        // Inicialización de totales y acumuladores
        int aprobados = 0;
        int sumaNotas = 0;
        int totalAlumnos = 0;

        // Vamos a mostrar los datos por consola
        // En este paso podríamos mostrar la información que nos interese que aparezca ANTES de los datos del alumno
        // Por ejemplo, las cabeceras
```

```

System.out.println("Listado de Alumnos\n");
System.out.println("Alumno\t\tNota\tAprobado");
System.out.println("-----\t\t-----\t-----");

// Ahora repetimos el proceso UNA VEZ POR CADA ALUMNO
do {
    // Pedimos los datos;
    String nombre = JOptionPane.showInputDialog("Nombre Alumno");
    String apellidos = JOptionPane.showInputDialog("Apellidos Alumno");
    String resp = JOptionPane.showInputDialog("Nota (1-10)"); // Reutilizamos la variable resp
    // Convertimos la respuesta a un entero
    int nota = Integer.parseInt(resp);

    // Creamos un nuevo alumno y le asignamos los datos
    Alumno a = new Alumno();
    a.nombre = nombre;
    a.apellidos = apellidos;
    a.nota = nota;

    // Queremos mostrar Si / No si está o no aprobado
    // Queremos sumar uno a aprobados SÓLO si está aprobado
    // Podemos hacer las dos cosas a la vez:
    String aprobado; // Aquí guardaremos "Si" o "No"
    if (a.aprobado()) {
        aprobado = "Si";
        // Hay un aprobado más
        aprobados++; // aprobados = aprobados + 1;
    } else {
        aprobado = "No";
    }

    // Mostramos los datos del alumno
    System.out.println(a.apellidos + ", " + a.nombre + "\t" + a.nota + "\t" + aprobado);
    // Sumamos su nota al total
    sumaNotas += a.nota;
    // contamos un alumno más
    totalAlumnos++;
} while (JOptionPane.showConfirmDialog(
    null,
    "Desea Introducir los Datos de Otro Alumno",
    "Más alumnos",
    JOptionPane.YES_NO_OPTION)
    ==
    JOptionPane.YES_NO_OPTION);

// Mostramos los totales
System.out.println("\nResumen");
System.out.println("Total Alumnos : " + totalAlumnos);
// Calculamos el porcentaje de aprobados
double porcAprobados = aprobados * 100.0 / totalAlumnos;
// Lo mostramos
System.out.println("Aprobados : " + aprobados + " (" + porcAprobados + "%)");
// Calculamos la media de notas (casting para obtener decimales)
double media = (double) sumaNotas / totalAlumnos;
// La mostramos
System.out.println("Nota media del curso : " + media);
}
}

```

Validar una entrada de datos del usuario (while)

En una aplicación es habitual que tengamos que comprobar que los datos del usuario son válidos. Por ejemplo, que una nota esté entre 1 y 10 o que un importe sea mayor que 0.

Enunciado

Modificar la aplicación anterior para que compruebe que la nota que introduce el usuario es un valor entre 1 y 10. De no ser así, se mostrará un mensaje de error y se volverá a pedir la nota.

Análisis

Aunque puede dar la sensación de que la validación es una alternativa (si el dato es erróneo se vuelve a pedir) en realidad es una repetitiva dado que el usuario puede introducir el dato incorrectamente varias veces (mientras el dato sea incorrecto se vuelve a pedir). Como la repetitiva se basa en una condición es un while o un do while. En este caso es evidente que es un while dado que puede que el usuario no cometa ningún error.

Solución

```

import javax.swing.JOptionPane;

public class Notas2ConValidacion {
    public static void main(String[] args) {
        int aprobados = 0;
        int sumaNotas = 0;
        int totalAlumnos = 0;

        System.out.println("Listado de Alumnos\n");
        System.out.println("Alumno\t\tNota\tAprobado");
        System.out.println("-----\t\t-----\t-----");

        do {
            String nombre = JOptionPane.showInputDialog("Nombre Alumno");
            String apellidos = JOptionPane.showInputDialog("Apellidos Alumno");
            // Pedimos la nota

```

```

String resp = JOptionPane.showInputDialog("Nota (1-10)");
int nota = Integer.parseInt(resp);
// Mientras la nota NO SEA VÁLIDA
while (nota < 1 || nota > 10) {
    // Mostramos un mensaje de error
    JOptionPane.showMessageDialog(null,
        "La nota debe ser un valor entre 1 y 10.",
        "Dato Erróneo",
        JOptionPane.ERROR_MESSAGE);
    // Y la volvemos a pedir
    resp = JOptionPane.showInputDialog("Nota (1-10)");
    nota = Integer.parseInt(resp);
}

Alumno a = new Alumno();
a.nombre = nombre;
a.apellidos = apellidos;
a.nota = nota;
String aprobado;
if (a.aprobado()) {
    aprobado = "Sí";
    aprobados++;
} else {
    aprobado = "No";
}
System.out.println(a.apellidos + ", " + a.nombre + "\t" + a.nota + "\t" + aprobado);
sumaNotas += a.nota;
totalAlumnos++;
} while (JOptionPane.showConfirmDialog(
    null,
    "Desea Introducir los Datos de Otro Alumno", // El mensaje
    "Más alumnos", // El título de la ventana
    JOptionPane.YES_NO_OPTION) // Las opciones (Sí/No)
    ==
    JOptionPane.YES_NO_OPTION); // Comprobamos si la respuesta es
                                // Sí

System.out.println("\nResumen");
System.out.println("Total Alumnos : " + totalAlumnos);
double porcAprobados = aprobados * 100.0 / totalAlumnos;
System.out.println("Aprobados : " + aprobados + " (" + porcAprobados + "%)");
double media = (double) sumaNotas / totalAlumnos;
System.out.println("Nota media del curso : " + media);
}
}

```

1.21. Ejercicios Adicionales

Ejemplos de Repetitivas

A continuación, se presentan **tres** ejemplos de estructuras repetitivas en Java que pueden resultar muy útiles en la práctica.

Fibonacci

Para calcular los **n** primeros números de la **serie de Fibonacci**, debemos recordar que cada valor es la suma de los dos anteriores. El resultado esperado para **n = 10** sería:

Primeros 10 números de la serie de Fibonacci:
0 1 1 2 3 5 8 13 21 34

Solución en Java:

```

public class SerieFibonacci {
    public static void main(String[] args) {
        int n = 10;
        int t1 = 0;
        int t2 = 1;

        System.out.println("Primeros " + n + " números de la serie de Fibonacci:");

        for (int i = 1; i <= n; i++) {
            System.out.print(t1 + " ");
            int sum = t1 + t2;
            t1 = t2;
            t2 = sum;
        }
    }
}

```

En este ejemplo, la variable **n** indica cuántos números de la serie mostraremos. Inicialmente, **t1** es 0 y **t2** es 1. Dentro del bucle **for**, se calcula la suma de **t1** y **t2** para ir avanzando en la serie.

Ejemplo adicional con un bucle **while** que hace casi lo mismo, para mostrar de forma alternativa cómo usar un **while**:

```

public class SerieFibonacciWhile {
    public static void main(String[] args) {
        int n = 10;
        int t1 = 0;
        int t2 = 1;
        int i = 1;

        System.out.println("Serie de Fibonacci (while) hasta " + n + " términos:");
    }
}

```

```

        while (i <= n) {
            System.out.print(t1 + " ");
            int sum = t1 + t2;
            t1 = t2;
            t2 = sum;
            i++;
        }
    }
}

```

Adivina el número

En este juego, el ordenador "piensa" un número que el usuario debe adivinar. Se indicará cuántos **intentos** ha necesitado el usuario para acertar. Para obtener un número aleatorio entre 1 y 100, se empleará la expresión:

```
(int) (Math.random() * 100) + 1
```

donde:

- `Math.random()` retorna un valor de tipo `double` cuyo valor es ≥ 0 y < 1 .
- Al multiplicarlo por 100, se obtiene un valor entre 0 y 99.9999...
- Al convertirlo a `int`, se redondea a un valor entre 0 y 99.
- Al sumarle 1, obtenemos un valor entre 1 y 100.

Solución en Java:

```

import javax.swing.JOptionPane;

public class AdivinaNumero {
    public static void main(String[] args) {
        int numeroSecreto = (int) (Math.random() * 100) + 1;
        int numero;
        int intentos = 1;

        String resp = JOptionPane.showInputDialog("Introduce un número entre 1 y 100");
        numero = Integer.parseInt(resp);

        while (numero != numeroSecreto) {
            if (numero > numeroSecreto) {
                JOptionPane.showMessageDialog(null, ";Incorrecto!. El número es MENOR");
            } else {
                JOptionPane.showMessageDialog(null, ";Incorrecto!. El número es MAYOR");
            }
            resp = JOptionPane.showInputDialog("Introduce un número entre 1 y 100");
            numero = Integer.parseInt(resp);
            intentos++;
        }

        JOptionPane.showMessageDialog(null,
            ";Correcto!. Has necesitado " + intentos + " intentos");
    }
}

```

En este programa, se usa un **bucle while** que se mantendrá activo **mientras** (`while`) el `numero` ingresado no coincida con `numeroSecreto`. Cada vez que el usuario se equivoca, se indica si debe probar un número **menor** o **mayor**. Al acertar, se muestra el total de `intentos`.

Menú Interactivo

Este ejemplo muestra cómo diseñar un **menú interactivo** que permite seleccionar varias opciones, incluyendo una para **salir**. En función de la opción elegida, se mostrará un mensaje y se volverá al menú hasta que el usuario seleccione la opción para salir.

Solución en Java:

```

import javax.swing.JOptionPane;

public class MenuInteractivo {
    public static void main(String[] args) {
        int opcion;
        do {
            String resp = JOptionPane.showInputDialog("""
                Menú:
                1. Opción 1
                2. Opción 2
                3. Salir
                Elige una opción:
                """);

            opcion = Integer.parseInt(resp);

            switch (opcion) {
                case 1:
                    JOptionPane.showMessageDialog(null, "Has elegido la Opción 1");
                    break;
                case 2:
                    JOptionPane.showMessageDialog(null, "Has elegido la Opción 2");
                    break;
                case 3:
                    JOptionPane.showMessageDialog(null, "Saliendo...");
                    break;
            }
        } while (opcion != 3);
    }
}

```

```

        default:
            JOptionPane.showMessageDialog(null, "Opción no válida");
        }
    } while (opcion != 3);
}
}

```

En este fragmento de código, se emplea un **bucle do while**:

- Primero se muestra el **menú** al usuario.
- Se convierte la cadena introducida en un valor **int** para manejar la **opción**.
- Según la **opción**, se ejecuta un bloque de código dentro de un **switch**.
- Si la **opción** es 3, el bucle finaliza con la condición `opcion != 3`.

Este tipo de bucle es un ejemplo de **repetitiva de 1 a n**, ya que al menos una vez se mostrará el menú independientemente de la opción, y continuará repitiéndose hasta que se seleccione **Salir**.

1.22. Actividad 4_2 - Repetitivas y Alternativas

Instrucciones

En una empresa se deben calcular los sueldos de los empleados. Para ello vamos a crear una clase `Empleado` con los siguientes campos:

- `nombre del empleado`
- `tipo de empleado`
- `número de hijos` del empleado
- `pluses`

Además dispondrá de los siguientes métodos

- `String getDescripcion()` : Retornará la descripción del `tipo` de empleado según la siguiente tabla:

Tipo	Descripción	Sueldo Base
1	Operario	1200
2	Obrero Especialista	1450
3	Administrativo	1300
4	Licenciado	1450

- `double getSueldoBase()` : Retornará el sueldo base (de la misma tabla)
- `void nuevoPlus(double plus)` : Sumará el plus recibido al campo `pluses`
- `double getSueldoBruto()` : Retornará el sueldo base más los pluses
- `double getPorcIrpf()` : Retornará el % de IRPF en base al **sueldo bruto** y los hijos según la siguiente tabla (por ejemplo, para un empleado con un sueldo bruto de 2000 y 1 hijo, retornaría 16):

	Número de hijos	0	1	2	3 o más
Sueldo Bruto	0	1200	1200	1200	1200
< 1350	13%	12,5%	11%	10%	
1350-1450	14%	13%	12%	11,5%	
>=1450	17%	16%	15%	14,5%	

- `double getImporteIRPF()` : El **sueldo bruto multiplicado por el porcentaje** de IRPF (habrá que dividirlo entre 100)
- `double getSueldoNeto()` : El **sueldo bruto menos el importe** correspondiente al IRPF

Crear un programa que, en su clase `main`, realice las siguientes tareas:

Repetir

- Pedir los siguientes datos
- `nombre del empleado`
- `tipo de empleado`
- `número de hijos`
- Crear un objeto `Empleado` con esos datos y poner sus `pluses` a 0
- Mostrar por pantalla los datos que tenemos hasta el momento. Por ejemplo:

```

Empleado : Juan Marin      Tipo : Obrero Especialista
Hijos : 3
Sueldo Base : 1450

```

- Calcular los **pluses** del empleado. Los pluses son ingresos adicionales que puede tener por trabajar de noche, en sitios peligrosos... Dado que **pueden ser varios**, se realizará el siguiente proceso:
 - Pedir un plus
 - Mientras el plus no sea 0
 - Incrementar los pluses del empleado con el plus (empleando el método definido en la clase)
 - Pedir otro plus
 - Tras ello, ya podemos mostrar el resto de los datos del empleado. Por ejemplo:

```
Pluses: 350.5
Sueldo Bruto: 1800.5
IRPF: 14.5% Importe IRPF: 251.0725
Sueldo Neto: 1549,4275
```

- Preguntar al usuario si desea introducir los datos de un nuevo empleado

Mientras el usuario conteste afirmativamente

ADICIONAL

Cuando se acabe con todos los empleados se mostrará por pantalla **cuántos empleados se han procesado, cuántos de ellos tienen un sueldo neto de más de 2000 € y el sueldo medio** de los empleados de la empresa.

AVANZADO

- los datos de todos los empleados en **una tabla HTML** en un `JOptionPane`. Para ello, se deberá crear una variable de tipo `String` en la que se irá generando el código HTML necesario que se mostrará al final.
- Mostrar los datos del **empleado que más gana** de la empresa.

1.23. Ejercicio Adicional

Instrucciones

Queremos controlar los envíos de una empresa de mensajería que dispone de cuatro tipo de servicios diferentes para lo que crearemos una clase Envío. De cada envío almacenará la siguiente información:

- destinatario : una cadena con información del destinatario
- remitente : idem con información del remitente
- peso : peso en gramos del paquete
- tipo : tipo de envío. Un valor entre 1 y 4 que corresponde con los datos de la siguiente tabla (emplearemos cuatro constantes para representar cada tipo de envío).

La tabla con la información de las tarifas es la siguiente (donde la columna precio/200 gramos indica un coste adicional a pagar en función del peso del envío :

Tipo	Descripción	Coste Base	Coste / 200 gramos
1	Normal	7€	0,50 €
2	Express	10€	0,70 €
3	24 Horas	12€	0,80 €
4	8 Horas	15€	0,90 €

Es decir, un paquete de 4.500 g enviado por modalidad Express costaría 10 € fijos más 16,1 € por el peso ($4500 / 200 \cdot 22,5 + 23 * 0,7 = 16,1$ €). Por tanto el total sería 26,1€. Nota: si el resto (%) de dividir el peso entre 200 no es cero se debe tomar una unidad más.

Se pide diseñar una clase Envío que contenga los cuatro campos identificados previamente, así como los siguientes métodos:

- un método `getDescripcionTipo` que retornará la descripción del envío como una cadena
- un método `getCosteBase` que retornará el coste base asociado al envío (en el ejemplo 10)
- un método `getCoste200` que retornará el coste asociado a cada 200 gramos (en el ejemplo 0,7)
- un método `getCostePeso` que retornará el coste asociado al peso del envío. (en el ejemplo 16,1€)
- un método `getCoste` que retornará el coste total del envío (26,1€)

La empresa, además, tiene estipulados una serie de **descuentos** en función del importe total a aplicar a cada envío (que usaremos en el programa). Dichos descuentos se indican en la siguiente tabla:

Importe Total	% descuento
< 50 €	0
Entre 50 y 100 €	3%
100 € o más	5%

Se pide diseñar un programa que permita obtener los importes correspondientes a los envíos en los diferentes días de la semana (sólo se consideran los días laborables). El proceso a realizar será el siguiente:

- Para cada día de la semana:
 - Se pedirá el tipo del servicio a utilizar.
 - Cuando se introduzca un valor 0 se darán por finalizados los envíos del día.
 - Si no es el caso:
 - Se pedirá el remitente, el destinatario del envío
 - Se pedirá el peso (en gramos) del envío.
 - Se deberá validar que el peso esté entre 0 y 1.000.000 gramos (mientras no sea así se dará un mensaje de error y se volverá a pedir el peso)
 - Se creará un nuevo envío condichos datos
 - Se mostrará el tipo (descripción) de servicio elegido, el precio base y el precio por cada 200 gramos.
 - Se mostrará por pantalla el importe total del envío, el descuento y el importe con el descuento incluido.
- Cuando se haya acabado con los envíos del día:
 - se mostrará el nombre del día, la suma de los importes (sin el descuento), la suma de los descuentos y la suma de los importes con el descuento incluido.
- Al acabar con los cinco días laborables:
 - se mostrará la suma de todos los importes, descuentos e importes con descuento de toda la semana.
 - Se mostrará el peso total correspondiente a todos los envíos
 - Por último se indicará cuántos días ha habido en los que se hayan obtenido importes totales (sin descuento) superiores a 200€.

ADICIONAL:

Modificar el programa para que, al final, muestre cuál ha sido el **envío con mayor coste total**. Para ello:

- se creará una variable de tipo Envío, llamada maximo
- dentro de la **repetitiva por envío** y si es la **primera vez** (emplear una variable booleana), se **asignará el envío actual al máximo**
- si no lo es se comprobará **si el coste del envío actual es mayor que el máximo** y, en ese caso, se asignará como máximo el actual

1.24. Ejercicio Adicional-II

Ejercicio: Gestión de Inventario de Productos

En una tienda, se debe gestionar el inventario de productos. Para ello, vamos a crear una clase Producto con los siguientes campos:

- nombre
- categoría (un entero de 1 a 4)
- precio unitario
- stock (unidades en el almacén)

Además, dispondrá de los siguientes métodos:

- String getDescripcionCategoria(): Retornará la descripción de la categoría del producto según la siguiente tabla:

Categoría	Descripción
1	Electrónica
2	Ropa
3	Alimentos
4	Hogar

- double getPrecioStock(): Retornará el precio total del producto en stock (precio unitario multiplicado por la cantidad en stock).
- void agregarStock(int cantidad): Incrementará la cantidad en stock del producto con la cantidad recibida.
- void reducirStock(int cantidad): Reducirá la cantidad en stock del producto con la cantidad recibida, siempre que haya suficiente stock disponible (si no, no hará nada)
- boolean hayStockSuficiente(int cantidad): Retornará true si hay suficiente stock para la cantidad solicitada, de lo contrario, retornará false.

Programa Principal

Crear un programa que, en su clase main, realice las siguientes tareas:

- Repetir:
- Pedir los siguientes datos:
 - nombre del producto
 - categoría del producto
 - precio unitario
 - unidades en stock
- Crear un objeto Producto con esos datos.
- Mostrar por pantalla los datos que tenemos hasta el momento. Por ejemplo: Producto: Televisor Categoría: Electrónica Precio Unitario: 500.0
Unidades en Stock: 10
- Calcular el importe total del producto (`stock * precio`) y mostrarlo.

- Permitir al usuario **modificar** el `stock` del producto:
 - Pedir una cantidad para agregar al stock.
 - Mientras la cantidad **no sea 0**:
 - Incrementar el stock del producto con la cantidad (empleando el método definido en la clase)
 - Pedir otra cantidad para agregar
 - Mostrar el nuevo stock y el importe total actualizado.
- Preguntar si se desea introducir los datos de un nuevo producto.
- Mientras el usuario conteste afirmativamente.

Adicional

- Al finalizar, mostrar cuántos productos se han procesado y el valor total del inventario de la tienda.

Avanzado

- Mostrar los datos de todos los productos en una tabla HTML en un `JOptionPane`. Para ello, se deberá crear una variable de tipo `String` en la que se irá generando el código HTML necesario que se mostrará al final.
- Mostrar los datos del producto con el mayor valor total en el inventario.

1.25. Programación Modular

Programación Modular en Java

La **programación modular** divide un programa complejo en múltiples **unidades funcionales**, cada una encargada de realizar una **tarea** bien definida. En el caso de una aplicación Java, podemos emplear los **métodos** para modularizarla.

Un método es un **bloque de código** al que se le asigna un **nombre** y, **opcionalmente**, se le pueden pasar una serie de **parámetros**. Además, los métodos pueden **retornar resultados** aunque, en el caso de modularizar una aplicación, lo habitual es que **no retornen nada** (lo que se indica con la palabra clave `void`).

Para **dividir** una aplicación en **métodos**, estos deben ser **estáticos** (lo explicaremos con más detalle en otra unidad). Si necesitamos **guardar información** que se pueda emplear en todos los métodos, la declararemos como **propiedades de la clase** mediante:

```
private static tipo propiedad;
```

Vamos a diseñar una aplicación que permita controlar los **ingresos / gastos** de una empresa en un día dado. La aplicación dispondrá de un menú principal con las siguientes opciones:

1. Nuevo Ingreso
2. Nuevo Gasto
3. Resultado Actual
4. Salir

Opción [1-4]:

A continuación, se muestra el **código de la aplicación** que implementa este menú modularizado en métodos estáticos:

```
import javax.swing.JOptionPane;

public class ControlIngresosGastos {

    // Propiedades de la clase (estáticas para que sean accesibles en los métodos)
    private static double totalIngresos = 0.0;
    private static double totalGastos = 0.0;

    public static void main(String[] args) {
        int opcion;

        do {
            // Construimos el texto del menú
            String menuText = """
                === Menú Principal ===
                1. Nuevo Ingreso
                2. Nuevo Gasto
                3. Resultado Actual
                4. Salir

                Elige una opción [1-4]:
                """;

            // Mostramos el menú y recogemos la opción
            String opcionStr = JOptionPane.showInputDialog(menuText);

            // Controlamos la posibilidad de cancelar o cerrar el cuadro de diálogo
            if (opcionStr == null) {
                // Si se cierra la ventana o se pulsa Cancelar, salimos directamente
                opcion = 4;
            } else {
                // Convertimos la opción a int
                opcion = Integer.parseInt(opcionStr);
            }

            switch (opcion) {
                case 1 -> nuevoIngreso();
                case 2 -> nuevoGasto();
                case 3 -> mostrarResultado();
                case 4 -> ;
                default -> JOptionPane.showMessageDialog(null, "Opción no válida. Intenta de nuevo.");
            }
        } while (opcion != 4);
    }

    private static void nuevoIngreso() {
        // Implementación del método
    }

    private static void nuevoGasto() {
        // Implementación del método
    }

    private static void mostrarResultado() {
        // Implementación del método
    }
}
```

```

        } while (opcion != 4);
    }

    // Método para registrar un nuevo Ingreso
    private static void nuevoIngreso() {
        String ingresoStr = JOptionPane.showInputDialog("Introduce el importe del ingreso:");
        if (ingresoStr == null) {
            // Si se cancela/ cierra el diálogo
            return;
        }

        double ingreso = Double.parseDouble(ingresoStr);
        totalIngresos += ingreso;

        JOptionPane.showMessageDialog(null,
            "Ingreso registrado correctamente.\nTotal Ingresos: " + totalIngresos);
    }

    // Método para registrar un nuevo Gasto
    private static void nuevoGasto() {
        String gastoStr = JOptionPane.showInputDialog("Introduce el importe del gasto:");
        if (gastoStr == null) {
            // Si se cancela/ cierra el diálogo
            return;
        }

        double gasto = Double.parseDouble(gastoStr);
        totalGastos += gasto;

        JOptionPane.showMessageDialog(null,
            "Gasto registrado correctamente.\nTotal Gastos: " + totalGastos);
    }

    // Método para mostrar el Resultado (Ingresos - Gastos)
    private static void mostrarResultado() {
        double balance = totalIngresos - totalGastos;

        JOptionPane.showMessageDialog(null,
            """
            === Resultado Actual ===
            Ingresos: %.2f
            Gastos: %.2f
            Balance: %.2f
            """.formatted(totalIngresos, totalGastos, balance)
        );
    }
}

```

1.26. Actividad 5 - Programación Modular

Instrucciones

Se desea diseñar una aplicación para llevar el bote de una cuadrilla de amigos. La aplicación, al iniciarse, mostrará el siguiente menú:

1. Añadir Amigos
 2. Nueva Aportación
 3. Pagar Ronda
 4. Quitar Amigos
 5. Salir
- Opción [1-5]

La aplicación realizará las siguientes tareas:

- Se definirá una propiedad **dentro de la clase pero fuera del método main** para almacenar el número de `amigos` y otra para almacenar el `bote` (ambas inicializadas a 0):

```

private static int amigos = 0;
private static double bote = 0.0;

private static final int BOTE_PERSONA = 10;

```

- Se definirá una **constante** con la cantidad a aportar como bote (por ejemplo 10 euros)
- Se mostrará el menú mientras el usuario no seleccione la opción de salir y se ejecutará un método distinto para cada opción (como en el ejemplo visto en clase)

Añadir Amigos

En esta opción :

- Se preguntará cuántos amigos se han añadido y se guardarán en una variable llamada `añadidos`.
- Si **no hay bote**:
- Se añadirá al bote el resultado de **multiplicar los amigos añadidos por el BOTE_PERSONA**
- Si **hay bote**:

- Se dividirá el bote entre los amigos actuales para ver **cuánto tienen que poner los amigos que vamos a añadir ahora**
- Se añadirá al bote el resultado de **multiplicar los amigos añadidos por la cantidad calculada en el paso anterior**
- Se añadirán los amigos **añadidos** a la propiedad **amigos**
- Se mostrará por pantalla cuántos amigos se han añadido y el bote actual.

Nueva Aportación

En esta opción

- Se añadirá al **bote** el resultado de **multiplicar los amigos por el `BOTE_PERSONA`**
- Se mostrará por pantalla cuántos amigos hay y el **bote** actual.

Pagar Ronda

En esta opción

- Se pedirá el **importe** de la ronda
- Si es **mayor que el bote** actual
- Se dará un mensaje de **error**
- Si **no lo es**
- Se descontará el **importe** de la ronda del **bote**
- Se mostrará el **bote** actual

Quitar Amigos

En esta opción

- Se preguntará cuántos amigos se van a quitar y se guardarán en una variable llamada **quitados**.
- Si hay bote:
 - Se dividirá el **bote** entre los **amigos** actuales para ver **cuánto hay que devolver** a los amigos que se van
 - Se restará al **bote** el resultado de **multiplicar los amigos quitados por la cantidad** calculada en el paso anterior
 - Se mostrará por pantalla la **cantidad** a devolver a cada amigo y el **bote** final.
- Si no lo hay
- Se dará un mensaje indicando que no queda bote

1.27. Estructuras de Control Control de Excepciones

Instrucciones de control de excepciones

En Java, al igual que en otros lenguajes, existe un **mecanismo estructurado para la gestión de excepciones** (condiciones que provocarían un error en la ejecución del programa). El formato habitual para **controlar excepciones** es el siguiente:

```
try {
    // Instrucciones que pueden provocar una excepción
} catch (tipoExcepción nombre) {
    // Instrucciones de gestión de la excepción
} [catch (tipoExcepción2 nombre) {
    // Instrucciones de gestión de la excepción
}] [finally {
    // Instrucciones a ejecutar de cualquier modo
}]
```

Si en el **bloque try** se produce una **excepción**, el flujo salta al **bloque catch**, comprobando si el **tipo de la excepción** coincide con alguno de los especificados. Si coincide, se ejecuta el código del **catch**. Si no coincide con ninguno de los tipos, se genera una excepción que interrumpe el programa.

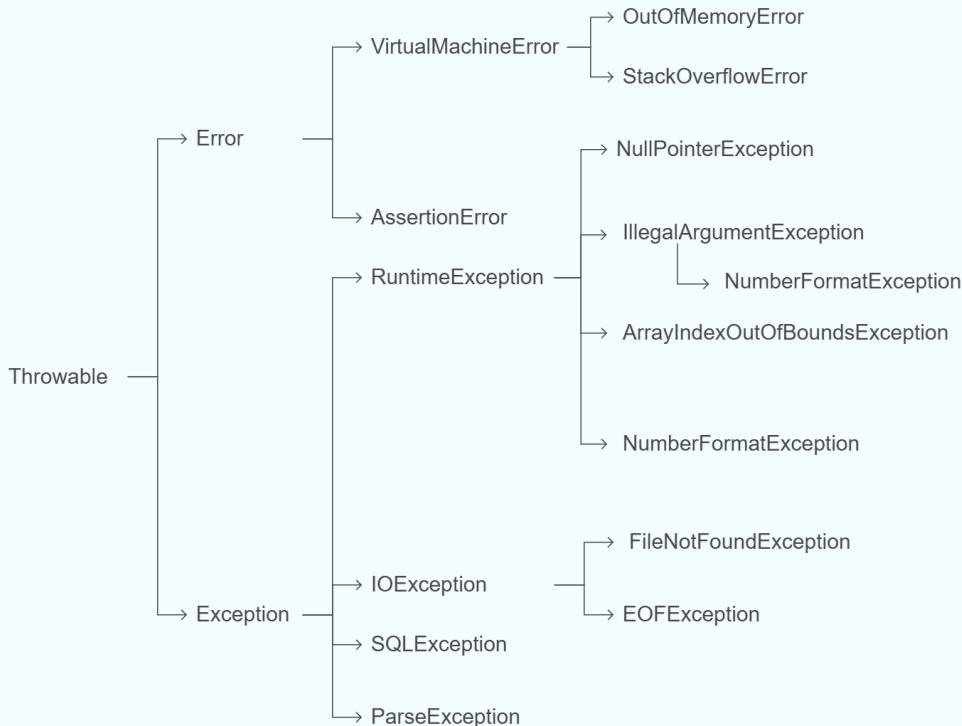
Si se incluye un **bloque finally**, este se ejecutará **siempre**, tanto si se ha producido una excepción como si no.

Cuando una clase o un método no desea capturar una excepción, puede **lanzarla** con la sintaxis:

```
class NombreClase ... throws TipoExcepción
```

Los tipos de excepciones se pueden ver en el siguiente diagrama.

Jerarquía de Excepciones en Java



Explicación de la jerarquía:

1. **Throwable**: Es la clase base de todas las excepciones y errores en Java. Cualquier clase que herede de `Throwable` puede ser lanzada y capturada en un bloque `try-catch`.
2. **Error**: Representa errores graves que generalmente no deben ser capturados ni manejados por la aplicación. Estos errores suelen ser irrecuperables y están relacionados con problemas en la JVM (Java Virtual Machine). Ejemplos comunes incluyen `OutOfMemoryError` y `StackOverflowError`.
3. **Exception**: Es la clase base para todas las excepciones que pueden ser manejadas por la aplicación. Las excepciones se dividen en dos categorías principales:
4. **RuntimeException**: Estas son **excepciones no verificadas** (`unchecked exceptions`) que generalmente indican errores de programación, como `NullPointerException`, `ArrayIndexOutOfBoundsException`, etc. **No es obligatorio capturarlas**.
5. **Otras excepciones**: Estas son **excepciones verificadas** (`checked exceptions`) que **deben ser capturadas explícitamente** en el código, como `IOException`, `SQLException`, etc.

Ejemplos de excepciones comunes:

- **NullPointerException**: Se lanza cuando se intenta acceder a un objeto que es `null`.
- **ArrayIndexOutOfBoundsException**: Se lanza cuando se intenta acceder a un índice fuera de los límites de un array.
- **NumberFormatException**: Se lanza cuando se intenta convertir una cadena que no representa un número en un número (con los métodos `parse` de las clases numéricas).
- **IOException**: Se lanza cuando ocurre un error durante la entrada/salida de datos, como al leer o escribir archivos.
- **FileNotFoundException**: Se lanza cuando se intenta acceder a un archivo que no existe.
- **SQLException**: Se lanza cuando se detectan errores en las operaciones con Bases de Datos.
- **ParseException**: Se lanza cuando se intenta *convertir* una cadena que no representa un número / fecha en un número / fecha.

Podemos **crear nuestras propias excepciones** extendiendo `Exception` o `RuntimeException`. Para **lanzar** una excepción (del tipo que sea) se emplea:

```
throw excepcion;
```

Ejemplo: Manejo de excepciones en Java

A continuación se muestra un ejemplo de cómo capturar excepciones generadas al **parsear** un número proporcionado por el usuario. Se incluye un **bloque try-catch-finally** y una excepción personalizada:

```
import javax.swing.JOptionPane;

// Excepción personalizada NO CHEQUEADA
class NumeroErroneoException extends RuntimeException {
    public NumeroErroneoException(String mensaje) {
        super(mensaje);
    }
}
```

```

        }

    }

public class ManejoExcepciones JOptionPane {
    public static void main(String[] args) {
        try {
            solicitaNumero();
        } catch (NumeroErroneoException e) {
            // Captura de nuestra excepción personalizada
            JOptionPane.showMessageDialog(null,
                "Se ha capturado la ExcepcionPersonalizada: " + e.getMessage());
        } catch (Exception e) {
            // Captura de cualquier otra excepción
            JOptionPane.showMessageDialog(null,
                "Ocurrió un error inesperado: " + e.getMessage());
        } finally {
            // Bloque que se ejecuta siempre
            JOptionPane.showMessageDialog(null,
                "Fin de la ejecución de la aplicación (bloque finally).");
        }
    }

    // Método que lanza una excepción personalizada si el valor no está dentro de un rango específico
    private static void solicitaNumero() throws ExcepcionPersonalizada {
        String valorStr = JOptionPane.showInputDialog("Introduce un número entero:");

        if (valorStr == null) {
            // El usuario canceló o cerró el diálogo; no hacemos nada
            return;
        }

        int valor = Integer.parseInt(valorStr); // Puede lanzar NumberFormatException

        if (valor < 1 || valor > 100) {
            // Lanzamos nuestra excepción personalizada
            throw new NumeroErroneoException("El número debe estar entre 1 y 100");
        }

        JOptionPane.showMessageDialog(null,
            "Número válido: " + valor);
    }
}

```

En este ejemplo, se emplea `showInputDialog` para solicitar un número entero al usuario. Si el usuario introduce un valor no convertible a `int`, se produce una `NumberFormatException`, capturada por un `catch (Exception e)` si no está capturada de forma más específica. Si el usuario introduce un número fuera del rango [1..100], se lanza la excepción personalizada (`NumeroErroneoException`). El bloque `finally` se ejecuta una vez finalizados los bloques `try` y `catch`, tanto si se ha lanzado o no una excepción.

2. Programación Orientada a Objetos con Java

2.1. Desarrollo de Clases Propiedades

Creación de Clases

Introducción a las Propiedades

Una clase, básicamente, está formada por una serie de propiedades (variables asociadas a la misma que almacenan la información asociada al objeto) y por métodos que realizan algún tipo de proceso sobre dichas propiedades. A las propiedades también se les denomina campos de la clase.

Las propiedades de una clase pueden ser de dos tipos:

- **de instancia:** Se definen dentro de la clase pero fuera de cualquier método. Cada vez que se crea (instancia) un objeto se le asigna una copia de dicha propiedad. La sintaxis de una propiedad de instancia es:

```
[acceso] tipo nombrePropiedad[=valorInicial];
```

- **de clase:** Se definen en el mismo sitio que las propiedades de instancia pero, a diferencia de ellas, sólo hay una copia por cada clase. La sintaxis es similar a la anterior pero precediendo al tipo de la propiedad con la palabra `static`.

```
[acceso] static tipo nombrePropiedad[=valorInicial];
```

Como ya hemos dicho las propiedades de instancia están asociadas a los objetos así que se accede a ellas a través del objeto:

```
objeto.propiedad
```

Sin embargo, las propiedades de la clase están asociadas a la propia clase, así que se accede a ellas a través de la clase:

```
Clase.propiedad
```

El modificador de acceso de una propiedad, método o clase determina desde donde es accesible dicho elemento. Existen 4 modificadores de acceso posibles a la hora de definir propiedades y métodos:

- **private:** Una propiedad o método con ámbito `private` sólo es accesible desde la propia clase.
- **[package]:** Si una propiedad o método no tiene un ámbito definido, entonces es de tipo paquete. En este caso, la propiedad o método es accesible desde la propia clase y desde cualquier otra clase que esté en el mismo paquete (directorio).
- **protected:** Las propiedades o métodos de tipo `protected` son accesibles desde la clase, desde las otras clases en el mismo paquete y desde clases hijas que estén en otros paquetes (veremos la herencia y las clases hijas más adelante en el curso).
- **public:** Las propiedades o métodos de tipo `public` son accesibles desde cualquier otra clase.

En el caso de las clases sólo pueden tener dos modificadores: `public` o `[package]` dado que los otros no tienen sentido.

NOTA: Es importante tener en cuenta que las propiedades privadas (y el resto evidentemente) son accesibles desde cualquier otra instancia de la misma clase.

Ejemplo:

Vamos a crear una clase (`Candidato`) para gestionar los resultados de un test que se pasa a una serie de candidatos. El test consta de tres pruebas cuya nota (un valor entre 1 y 10) vale un 40%, un 35% y un 25% respectivamente. Además, sabemos que el porcentaje mínimo para aprobar es de un 60%. Además de los resultados de las pruebas se desea almacenar el nombre del candidato.

Dado que cada candidato tendrá un nombre y unos resultados distintos, las propiedades `nombre`, `prueba1`, `prueba2` y `prueba3` serán propiedades de instancia (una copia para cada candidato). Sin embargo el valor del aprobado es el mismo para todos así que lo guardaremos a nivel de la clase (una propiedad estática). De hecho, dado que su valor no va a cambiar, lo definiremos como una constante.

```
// Clase Candidato
public class Candidato {
    // Propiedad estática (de clase) - constante
    public static final double APROBADO = 60.0;

    // Propiedades de instancia
    public String nombre;
    public double prueba1;
    public double prueba2;
    public double prueba3;

    // Método para calcular la nota final
    public double calcularNotaFinal() {
        return prueba1 * 0.4 + prueba2 * 0.35 + prueba3 * 0.25;
    }

    // Método para determinar si el candidato ha aprobado
    public boolean haAprobado() {
        return calcularNotaFinal() >= APROBADO;
    }
}

// Clase Programa (para probar la clase Candidato)
public class Programa {
    public static void main(String[] args) {
        // Crear un objeto de la clase Candidato
        Candidato candidato1 = new Candidato();

        // Asignar valores a las propiedades
        candidato1.nombre = "Juan Pérez";
        candidato1.prueba1 = 7.5;
```

```

candidato1.prueba2 = 6.8;
candidato1.prueba3 = 5.9;

// Crear otro candidato
Candidato candidato2 = new Candidato();
candidato2.nombre = "María López";
candidato2.prueba1 = 4.2;
candidato2.prueba2 = 5.5;
candidato2.prueba3 = 6.0;

// Mostrar resultados
System.out.println("Resultados de los candidatos:");
System.out.println("-----");

System.out.println("Candidato: " + candidato1.nombre);
System.out.println("Nota final: " + candidato1.calcularNotaFinal());
System.out.println("¿Ha aprobado?: " + (candidato1.haAprobado() ? "Sí" : "No"));
System.out.println();

System.out.println("Candidato: " + candidato2.nombre);
System.out.println("Nota final: " + candidato2.calcularNotaFinal());
System.out.println("¿Ha aprobado?: " + (candidato2.haAprobado() ? "Sí" : "No"));

// Acceder a la propiedad estática (a través de la clase)
System.out.println("\nNota mínima para aprobar: " + Candidato.APROBADO);
}
}

```

La salida del programa sería:

Resultados de los candidatos:

Candidato: Juan Pérez
Nota final: 6.895
¿Ha aprobado?: Sí

Candidato: María López
Nota final: 5.08
¿Ha aprobado?: No

Nota mínima para aprobar: 60.0

En este ejemplo:

- La clase `Candidato` tiene una propiedad estática `APROBADO` que es constante (`final`) y accesible desde cualquier clase (`public`).
- Las propiedades de instancia (`nombre, prueba1, prueba2, prueba3`) son públicas para que puedan ser accedidas desde la clase `Programa`.
- Los métodos `calcularNotaFinal()` y `haAprobado()` realizan operaciones sobre las propiedades de instancia.
- En la clase `Programa`, creamos dos objetos de tipo `Candidato`, les asignamos valores y mostramos sus resultados.
- Accedemos a la propiedad estática `APROBADO` a través de la clase `Candidato`.

2.2. Actividad 6 - Propiedades

Instrucciones

Se pide diseñar una clase `Email` para **gestionar los envíos de correo electrónico** de la empresa. De cada correo se almacenará la siguiente información:

- **Para** : Una cadena para almacenar el correo electrónico de la persona a la que se envía el correo
- **Asunto** : Idem para el tema del que trata el mensaje
- **Texto** : El texto del correo propiamente dicho

Dichas propiedades deberán ser **accesibles desde cualquier clase que esté en el mismo paquete**. Además, la clase contará con una **propiedad de clase** de que estará asignada a la dirección `noreply@zabalburu.org` y que deberá ser **accesible desde cualquier clase**. Asimismo se dispondrá de otra propiedad de clase llamada `numCorreos` de tipo `entero` e inicializada a `0`. Diseñar, además, una **clase ejecutable** llamada `EnvioCorreos` que realizará las siguientes tareas en su método `main`:

- Declarará una variable `email` de tipo `Email` (`Email email;`)
- **Repetirá** el siguiente proceso **mientras el usuario quiera**
- Asignar a `email` un nuevo `Email` (`email = new Email();`)
- Pedir el `destinatario`, `asunto` y `texto` del mensaje y asignarlo a las propiedades correspondientes del objeto `email`
- Incrementar la propiedad `numCorreos` de la clase `Email` en `1`
- Mostrar la información del `email` según el siguiente formato (los valores entre corchetes deben sustituirse por las propiedades correspondientes)

```

Mensaje : [numCorreos]
De : [de]
Para : [para]
Asunto : [asunto]
[texto]

-----

```

- Editar

-

[editar](#)

Tarea

- Tipo: Ejercicio escrito online (Ensayo)
- Máxima puntuación individual: 100
- Calificación: Cuenta para la media
- Categoría: Ningunos

[deshacer dar](#)

Programa

- Comenzar:

4 Nov, 9:11 am

- Fecha límite:

16 Nov, 9:11 am

- Asignado:

17 Oct

[libro de calificaciones](#)

Calificación

- 66 trabajos por calificar
- Fecha límite: 32, Enviado: 66
- calificadas: 0

[editar](#)

Opciones

Biblioteca: Personal Intentos máximos: 1 Permitir entregas fuera de plazo: Sí

Grados de publicación: Instant

Permita que los estudiantes comenten las entregas de otros estudiantes.: No Desactivar fecha de entrega: No

Previo

[Volver a la lista de las lecciones](#) [Continuar](#)

2.3. Desarrollo de Clases Métodos

Introducción a los Métodos

Como ya se ha comentado, un método permite añadir funcionalidad a una clase. Los métodos es donde escribimos el código Java y pueden acceder a las propiedades/métodos de la clase y a la información que le proporcione el usuario a través de los parámetros.

Un método se define por su firma que tiene el siguiente aspecto:

```
[acceso] [static] tipoRetorno nombreMétodo([listaArgumentos]) {  
    instrucciones;  
  
    [return [valorRetorno];]  
}
```

Donde:

- **acceso:** Es el modificador de acceso visto anteriormente que determina desde dónde se puede ejecutar el método.
- **static:** Al igual que las propiedades podemos tener métodos de dos tipos:
- **instancia:** NO llevan la palabra static. Se ejecutan sobre un objeto y pueden acceder a las propiedades/métodos de instancia y a las propiedades/métodos de clase (static).
- **clase:** Llevan la palabra static. Se ejecutan directamente sobre la clase y sólo pueden acceder a las propiedades/métodos de clase (static).
- **tipoRetorno:** Indica el tipo de dato que va a retornar el método. Si el método simplemente realiza una tarea y no retorna información se debe indicar con la palabra reservada void.
- **nombreMétodo:** El nombre del método.
- **listaParámetros:** Una lista opcional con la información que nos debe proporcionar el usuario de la clase para poder ejecutar el método. Los nombres de dichos datos se denominan parámetros, mientras que los valores que se pasan en un momento dado se denominan argumentos. La listaParámetros tendrá el siguiente formato:

`java tipo parámetro1[, tipo parámetro2...]`

donde, para cada parámetro se indicará de qué tipo es.

- **instrucciones:** Las instrucciones que se ejecutarán cuando se llame al método.
- **return:** Si el método no retorna nada (`void`) es opcional y se pondrá simplemente `return` (si no se indica, el método retorna tras ejecutar su última instrucción). Si el método retorna algo, en la instrucción se pondrá `return valorRetorno;` donde `valorRetorno` deberá ser una expresión del mismo tipo que el `tipoRetorno` indicado en la firma del método.

Sobrecarga de métodos

En una misma clase podemos tener múltiples métodos con el mismo nombre siempre que se diferencien en el número o el tipo de parámetros. Por ejemplo si tenemos el método:

```
void metodo1(int num1, int num2)
```

También podríamos tener los siguientes métodos:

```
void metodo1(int num1, int num2, int num3) // Tiene distinto número de parámetros  
int metodo1(String num1, int num2) // Tiene parámetros de distinto tipo
```

Pero no éste:

```
int metodo1(int num1, int num2) // Java no sabría a cuál llamar
```

Invocación de métodos

Al igual que con las propiedades, para ejecutar un método de instancia necesitamos un objeto:

```
objeto.metodo([listaArgumentos])
```

Y si es estático, lo ejecutaremos sobre la clase:

```
Clase.metodo([listaArgumentos])
```

Si un método no devuelve nada, lo emplearemos como una instrucción:

```
objeto.metodo([listaArgumentos]);
```

Si devuelve algo, lo emplearemos en cualquier sitio donde pueda ir un dato del mismo tipo del que devuelve el método. Por ejemplo, si `método1()` retorna un entero podríamos incluirlo en las siguientes instrucciones:

```
System.out.println("Resultado : " + objeto.metodo1());  
int rdo = objeto.metodo1();  
for(int i=0; i<objeto.metodo1();i++){  
    ...  
}  
if (objeto.metodo1()>10){  
    ...  
}
```

En este caso también podríamos ponerlo como una instrucción (pero su valor de retorno se perdería):

```
objeto.metodo1(); // Se ejecutan las instrucciones del método pero su valor devuelto se pierde
```

Invocación de métodos y propiedades desde la propia clase

Si queremos acceder a una propiedad/método de instancia desde cualquier método de la propia clase podemos emplear la palabra reservada `this` para indicar el objeto actual:

```
this.propiedad  
this.metodo(...)
```

Si queremos acceder a una propiedad/método de clase (`static`) desde cualquier método de la propia clase podemos emplear el nombre de la clase:

```
Clase.propiedad  
Clase.metodo(...)
```

NOTA: Si no indicamos explícitamente `this` o el nombre de la clase en la llamada a los métodos, Java asumirá esos valores si encuentra dicha propiedad/método definida/o en la clase. Es decir que si ponemos:

```
java propiedad metodo(...)
```

y están definidos en la clase (como propiedades de instancia o de clase), Java las usará correctamente.

El uso de `this` y del nombre de la clase, sin embargo, será obligatorio si tenemos una variable local o un parámetro en un método que se llaman igual que una propiedad de instancia/de clase.

En cualquier caso es recomendable indicar `this` y el nombre de la clase al objeto de clarificar el código.

NOTA: En cuanto a los métodos estáticos, Java admite que se invoquen no sólo a través de la clase sino a través de cualquier objeto de la misma. Es decir que si `metodo()` es estático podríamos emplear:

```
java Clase.metodo() objeto.metodo()
```

Y, desde métodos de la misma clase:

```
java Clase.metodo() this.metodo()
```

Es MUY RECOMENDABLE emplear siempre la sintaxis de la clase, dado que la otra puede llevar a confusión.

Ejemplo:

Vamos a añadir a nuestro ejemplo una serie de métodos para calcular el porcentaje de los resultados sacados en las notas y si han aprobado o no:

```
// Clase Candidato con métodos adicionales
public class Candidato {
    // Propiedad estática (de clase) - constante
    public static final double APROBADO = 60.0;

    // Propiedades de instancia
    public String nombre;
    public double prueba1;
    public double prueba2;
    public double prueba3;

    // Método para calcular el porcentaje de la prueba 1 (40%)
    public double calcularPorcentajePrueba1() {
        return this.prueba1 * 0.4;
    }

    // Método para calcular el porcentaje de la prueba 2 (35%)
    public double calcularPorcentajePrueba2() {
        return this.prueba2 * 0.35;
    }

    // Método para calcular el porcentaje de la prueba 3 (25%)
    public double calcularPorcentajePrueba3() {
        return this.prueba3 * 0.25;
    }

    // Método para calcular la nota final (suma de los porcentajes)
    public double calcularNotaFinal() {
        return this.calcularPorcentajePrueba1() +
            this.calcularPorcentajePrueba2() +
            this.calcularPorcentajePrueba3();
    }

    // Método para determinar si el candidato ha aprobado
    public boolean haAprobado() {
        return this.calcularNotaFinal() >= Candidato.APROBADO;
    }

    // Método estático para convertir una nota numérica a calificación
    public static String obtenerCalificación(double nota) {
        if (nota >= 90) {
            return "Sobresaliente";
        } else if (nota >= 70) {
            return "Notable";
        } else if (nota >= Candidato.APROBADO) {
            return "Aprobado";
        } else {
            return "Suspensivo";
        }
    }

    // Método de instancia que usa el método estático
    public String obtenerCalificaciónCandidato() {
        return Candidato.obtenerCalificación(this.calcularNotaFinal());
    }

    // Método sobrecargado: permite establecer todas las notas a la vez
    public void establecerNotas(double nota1, double nota2, double nota3) {
        this.prueba1 = nota1;
        this.prueba2 = nota2;
        this.prueba3 = nota3;
    }

    // Método sobrecargado: permite establecer una nota específica
    public void establecerNotas(int numeroPrueba, double nota) {
        switch (numeroPrueba) {
            case 1:
                this.prueba1 = nota;
                break;
            case 2:
                this.prueba2 = nota;
                break;
            case 3:
                this.prueba3 = nota;
                break;
            default:
                System.out.println("Número de prueba inválido");
        }
    }
}

// Clase Programa para probar los métodos
public class Programa {
    public static void main(String[] args) {
        // Crear un objeto de la clase Candidato
        Candidato candidato1 = new Candidato();

        // Asignar valores usando el método sobrecargado
        candidato1.nombre = "Juan Pérez";
```

```

candidato1.establecerNotas(7.5, 6.8, 5.9);

// Crear otro candidato
Candidato candidato2 = new Candidato();
candidato2.nombre = "María López";

// Asignar valores uno a uno usando el otro método sobrecargado
candidato2.establecerNotas(1, 4.2);
candidato2.establecerNotas(2, 5.5);
candidato2.establecerNotas(3, 6.0);

// Mostrar resultados detallados
System.out.println("Resultados de los candidatos:");
System.out.println("-----");

mostrarResultadosCandidato(candidato1);
System.out.println();
mostrarResultadosCandidato(candidato2);

// Usar el método estático directamente
System.out.println("\nEjemplo de uso del método estático:");
System.out.println("Calificación para 85: " + Candidato.obtenerCalificacion(85));
System.out.println("Calificación para 55: " + Candidato.obtenerCalificacion(55));
}

// Método auxiliar para mostrar los resultados de un candidato
private static void mostrarResultadosCandidato(Candidato candidato) {
    System.out.println("Candidato: " + candidato.nombre);
    System.out.println("Prueba 1: " + candidato.prueba1 +
        " (Porcentaje: " + candidato.calcularPorcentajePrueba1() + ")");
    System.out.println("Prueba 2: " + candidato.prueba2 +
        " (Porcentaje: " + candidato.calcularPorcentajePrueba2() + ")");
    System.out.println("Prueba 3: " + candidato.prueba3 +
        " (Porcentaje: " + candidato.calcularPorcentajePrueba3() + ")");
    System.out.println("Nota final: " + candidato.calcularNotaFinal());
    System.out.println("¿Ha aprobado?: " + (candidato.haAprobado() ? "Sí" : "No"));
    System.out.println("Calificación: " + candidato.obtenerCalificacionCandidato());
}
}

```

La salida del programa sería:

```

Resultados de los candidatos:
-----
Candidato: Juan Pérez
Prueba 1: 7.5 (Porcentaje: 3.0)
Prueba 2: 6.8 (Porcentaje: 2.38)
Prueba 3: 5.9 (Porcentaje: 1.475)
Nota final: 6.855
¿Ha aprobado?: Sí
Calificación: Aprobado

Candidato: María López
Prueba 1: 4.2 (Porcentaje: 1.68)
Prueba 2: 5.5 (Porcentaje: 1.925)
Prueba 3: 6.0 (Porcentaje: 1.5)
Nota final: 5.105
¿Ha aprobado?: No
Calificación: Suspenso

Ejemplo de uso del método estático:
Calificación para 85: Notable
Calificación para 55: Suspenso

```

En este ejemplo:

1. Hemos añadido métodos para calcular los porcentajes de cada prueba.
2. El método `calcularNotaFinal()` ahora utiliza los métodos de porcentaje.
3. Hemos añadido un método estático `obtenerCalificacion()` que convierte una nota numérica en una calificación textual.
4. Hemos creado un método de instancia `obtenerCalificacionCandidato()` que utiliza el método estático.
5. Hemos implementado sobrecarga de métodos con dos versiones de `establecerNotas()`.
6. En los métodos, utilizamos `this` para acceder a las propiedades de instancia y `Candidato` para acceder a la propiedad estática.
7. En la clase `Programa`, hemos creado un método auxiliar para mostrar los resultados de un candidato.

2.4. Actividad 7 - Métodos

Instrucciones

Se desea diseñar una clase `Producto` para gestionar las **ventas de los productos** de una empresa. La clase dispondrá de los siguientes **campos de instancia** (declararlos **públicos**):

- `nombre`: El nombre del producto (una cadena)
- `unidadesAlmacen`: Cuántas unidades hay actualmente en el almacén (un entero)
- `precio`: El precio del producto (un doble)
- `unidadesVendidas`: Las unidades que se han vendido del producto (entero)
- `ventasProducto`: El total de todas las ventas realizadas del producto (doble)

Todos los campos numéricos estarán **inicializados** a 0. Además, la clase dispondrá de los siguientes **campos de clase (públicos)**:

- **IVA**: Será una **constante** y almacenará el valor 21
- **descuento**: Un valor con decimales (initialmente 5) que representa el % de descuento a aplicar a las ventas cuando se **supera un importe** (ver el siguiente campo)
- **importeMinimoDto**: Un valor con decimales (initialmente 100) que representa **si una venta tendrá descuento o no**. Lo tendrá si el importe de la venta es superior a ese importe mínimo
- **ventasTotales**: El importe de todas las ventas de todos los productos (initialmente 0).

Por otro lado, la clase contará con los siguientes métodos (se indica si son de instancia o de clase):

- **[Instancia] - aumentarUnidades(int unidades)**: Este método incrementará las unidadesAlmacen con las unidades que nos proporciona el usuario. Será público y no retornará nada
- **[Instancia] - importe(int unidades)**: Este método retornará el resultado de multiplicar las unidades que nos pasan por el precio del producto teniendo en cuenta el IVA. Será privado [Instancia]
- **[Clase] - descuento(double importe)**: Este método retornará el descuento asociado al importe que se le pasa:
- Si el importe es menor que importeMinimoDto retornará 0
- Si no, retornará el resultado de calcular el descuento asociado al importe empleando la propiedad descuento
- **[Instancia] - vender(int unidades)**: Este método es el que realizará la venta propiamente dicha. El proceso será el siguiente:
- Si las unidades son más que las unidadesAlmacen (no hay suficientes unidades) retornará un -1
- Si hay suficientes unidades:
 - Obtendrá el importe con IVA (empleando el método importe definido previamente)
 - Obtendrá el descuento (empleando el método descuento)
 - Calculará el importeFinal (importe - descuento)
 - Incrementará el campo unidadesVendidas en las unidades recibidas
 - Descontará las unidades del campo unidadesAlmacen
 - Incrementará los campos ventasProducto y ventasTotales con el importeFinal calculado
 - Retornará el importeFinal

Diseñar una clase ejecutable llamada Almacén que, en su método main realice las siguientes tareas:

1. Declarar e inicializar **tres productos** con los datos que queráis (nombre, unidadesAlmacen y precio)
2. Repita el siguiente proceso **mientras el usuario quiera**: 2.1. Mostrará el siguiente cuadro de diálogo con los nombres de los tres productos:
 1. Producto
 2. Producto
 3. Producto

Seleccione el producto [1-3] o pulse 0 para finalizar

2.2. Almacenará en una variable de tipo Producto llamada p el producto que ha seleccionado el usuario (el resto del programa se hará empleando dicha variable)

2.3. Mostrará el siguiente cuadro de diálogo :

Producto : nombre
Unidades : unidadesAlmacen
Precio : precio

Desea [V]ender o [A]umentar las unidades del producto:

2.4 Si el usuario contesta con una A (mayúscula o minúscula): 2.4.1 Se preguntará cuántas unidades se han añadido al almacén 2.4.2 Se llamará al método aumentarUnidades de p 2.4.2 Se mostrará un mensaje mostrando las unidadesAlmacen de p

2.5 Si contesta con una V: 2.5.1 Se preguntará cuántas unidades se han vendido 2.5.2 Se llamará al método vender y se almacenará el resultado en una variable importe 2.5.3 Si importe es -1 se dará un **mensaje de error** (NO hay suficientes unidades) 2.5.4 Si no lo es se mostrará un **mensaje** confirmando la venta e indicando dicho importe.

1. Al salir de la repetitiva se mostrará el siguiente resumen:

```console Producto Unidades Almacen Precio Unidades Vendidas Ventas ====== ====== ====== ====== ====== ====== nombre unidadesAlmacen precio unidadesVendidas ventasProducto nombre unidadesAlmacen precio unidadesVendidas ventasProducto nombre unidadesAlmacen precio unidadesVendidas ventasProducto

Ventas Totales : ventasTotales

...

### 2.5. Constructores Métodos de Acceso y Otros Métodos Habituales

# Constructores

Dentro de los métodos que podemos incluir en una clase tenemos los constructores. Un constructor es un método que se ejecuta cuando se va a crear un objeto

```
```java
[public] Clase([lista_parámetros]) {
    ...
}
```

Los constructores se diferencian del resto de los métodos en:

- Su nombre debe coincidir con el nombre de la clase, lo que quiere decir que, si seguimos las recomendaciones de Java, deberían ser los únicos métodos que comiencen por una mayúscula.
- No se especifica ningún valor de retorno (ni siquiera void).

Al igual que el resto de los métodos pueden recibir parámetros y se pueden sobrecargar (tener múltiples constructores con distintos parámetros).

NOTA: Desde un constructor se puede llamar a otro empleando `this([lista-argumentos])`. Si se emplea, esta instrucción debe ser la primera en el constructor.

Constructor por Defecto

Dado que los constructores son imprescindibles para crear un objeto, en caso de no indicar ninguno, Java proporcionará uno por defecto sin parámetros. Es por eso que hemos podido crear objetos aunque no hayamos definido ninguno.

Eso sí, en el momento en que se define un constructor en la clase, Java ya no proporciona el constructor sin argumentos así que, si lo necesitamos, deberemos especificarlo nosotros.

Modificar la clase Candidato para añadir varios constructores

```
public class Candidato {
    // Propiedad estática (de clase) - constante
    public static final double APROBADO = 60.0;

    // Propiedades de instancia
    public String nombre;
    public double prueba1;
    public double prueba2;
    public double prueba3;

    // Constructor por defecto
    public Candidato() {
        // Inicialización por defecto
        this.nombre = "Sin nombre";
        this.prueba1 = 0;
        this.prueba2 = 0;
        this.prueba3 = 0;
    }

    // Constructor con nombre
    public Candidato(String nombre) {
        this(); // Llama al constructor por defecto
        this.nombre = nombre;
    }

    // Constructor con nombre y notas
    public Candidato(String nombre, double prueba1, double prueba2, double prueba3) {
        this(nombre); // Llama al constructor con nombre
        this.prueba1 = prueba1;
        this.prueba2 = prueba2;
        this.prueba3 = prueba3;
    }

    // Resto de métodos...
    // (Los métodos anteriores se mantienen igual)
}
```

Métodos de Acceso

La Programación Orientada al Objeto debe tener tres características:

- **Herencia:** Es la propiedad de que una clase herede de otra. Lo veremos en un apartado posterior.
- **Polimorfismo:** Es la propiedad de tener clases y métodos que se llamen igual. Se puede conseguir mediante los paquetes y la sobrecarga de métodos.
- **Encapsulación:** Es la propiedad de ocultar el estado (los campos) de los objetos al exterior. Para acceder a dichos campos se deberán emplear métodos de acceso.

Aunque hasta ahora hemos estado accediendo a los campos (estado) de los objetos directamente desde el exterior, esta forma de trabajar está altamente desaconsejada. Si permitimos el acceso directo a las propiedades de los objetos podemos tener varios problemas:

- No podemos validar la información que se introduce en las propiedades lo que puede dar lugar a trabajar con objetos inconsistentes. Por ejemplo, no podemos evitar notas negativas.
- No podemos evitar que el usuario de la clase pueda leer/escribir en la propiedad lo que nos impide tener propiedades que sólo sean de lectura o de escritura.
- No podemos cambiar el nombre o el tipo de las propiedades de la clase o almacenar la información en algún otro sitio que no sea una variable.

Para solucionar estos problemas se emplea la encapsulación que básicamente consiste en:

- Definir todas las propiedades como privadas evitando el acceso exterior a las mismas.
- Definir métodos para obtener o cambiar la información importante del objeto.

NOTA: La encapsulación no se aplica a las constantes dado que su valor no es modificable. Por tanto, es correcto definir una constante como pública (además de estática dado que no tiene sentido almacenar una copia en cada objeto).

Métodos de acceso a la información (accessors)

La recomendación es que la firma de los métodos sea como la siguiente:

```
[acceso] tipo getPropiedad() {  
    ...  
    return this.propiedad;  
}
```

Por ejemplo, para acceder al campo nombre se definiría un método `String getNombre()` y para la prueba1 sería `double getPrueba1()`.

NOTA: Para los campos booleanos se recomienda emplear `is` en lugar de `get`. Por ejemplo, suponiendo que tengamos una propiedad booleana llamada aprobado, se recomienda como método `boolean isAprobado()`.

NOTA: Dado que el usuario no tiene acceso a la información de la clase, esta forma de llamar a los métodos se suele emplear también para métodos que retorne resultados. Por ejemplo `getNotaFinal()`.

Métodos que modifican la información (mutators)

En este caso el método simplemente cambia el valor de una propiedad por lo que recibe el nuevo valor (del mismo tipo que la propiedad) y no retorna nada:

```
[acceso] void setPropiedad(tipo nuevoValor) {  
    ...  
    this.propiedad = nuevoValor;  
}
```

NOTA: Es habitual que nuevoValor se llame igual que la propiedad. En ese caso es imprescindible el uso de `this` para distinguir la propiedad que tenemos que modificar del valor del argumento que nos han pasado.

Añadir métodos set/get a la clase Candidato

```
public class Candidato {  
    // Propiedad estática (de clase) - constante  
    public static final double APROBADO = 60.0;  
  
    // Propiedades de instancia (ahora privadas)  
    private String nombre;  
    private double prueba1;  
    private double prueba2;  
    private double prueba3;  
  
    // Constructores  
    public Candidato() {  
        this.nombre = "Sin nombre";  
        this.prueba1 = 0;  
        this.prueba2 = 0;  
        this.prueba3 = 0;  
    }  
  
    public Candidato(String nombre) {  
        this();  
        this.nombre = nombre;  
    }  
  
    public Candidato(String nombre, double prueba1, double prueba2, double prueba3) {  
        this(nombre);  
        this.setPrueba1(prueba1); // Usamos los setters para validar  
        this.setPrueba2(prueba2);  
        this.setPrueba3(prueba3);  
    }  
  
    // Métodos getter  
    public String getNombre() {  
        return this.nombre;  
    }  
  
    public double getPrueba1() {  
        return this.prueba1;  
    }  
  
    public double getPrueba2() {  
        return this.prueba2;  
    }  
  
    public double getPrueba3() {  
        return this.prueba3;  
    }  
  
    // Métodos setter con validación  
    public void setNombre(String nombre) {  
        if (nombre != null && !nombre.trim().isEmpty()) {  
            this.nombre = nombre;  
        } else {  
            throw new IllegalArgumentException("El nombre no puede estar vacío");  
        }  
    }  
  
    public void setPrueba1(double prueba1) {
```

```

        if (prueba1 >= 0 && prueba1 <= 10) {
            this.prueba1 = prueba1;
        } else {
            throw new IllegalArgumentException("La nota debe estar entre 0 y 10");
        }
    }

    public void setPrueba2(double prueba2) {
        if (prueba2 >= 0 && prueba2 <= 10) {
            this.prueba2 = prueba2;
        } else {
            throw new IllegalArgumentException("La nota debe estar entre 0 y 10");
        }
    }

    public void setPrueba3(double prueba3) {
        if (prueba3 >= 0 && prueba3 <= 10) {
            this.prueba3 = prueba3;
        } else {
            throw new IllegalArgumentException("La nota debe estar entre 0 y 10");
        }
    }

    // Método para establecer todas las notas a la vez
    public void establecerNotas(double nota1, double nota2, double nota3) {
        this.setPrueba1(nota1);
        this.setPrueba2(nota2);
        this.setPrueba3(nota3);
    }

    // Método sobrecargado: permite establecer una nota específica
    public void establecerNotas(int numeroPrueba, double nota) {
        switch (numeroPrueba) {
            case 1:
                this.setPrueba1(nota);
                break;
            case 2:
                this.setPrueba2(nota);
                break;
            case 3:
                this.setPrueba3(nota);
                break;
            default:
                throw new IllegalArgumentException("Número de prueba inválido");
        }
    }

    // Resto de métodos de cálculo
    public double calcularPorcentajePrueba1() {
        return this.prueba1 * 0.4;
    }

    public double calcularPorcentajePrueba2() {
        return this.prueba2 * 0.35;
    }

    public double calcularPorcentajePrueba3() {
        return this.prueba3 * 0.25;
    }

    public double calcularNotaFinal() {
        return this.calcularPorcentajePrueba1() +
               this.calcularPorcentajePrueba2() +
               this.calcularPorcentajePrueba3();
    }

    public boolean haAprobado() {
        return this.calcularNotaFinal() >= Candidato.APROBADO;
    }

    public static String obtenerCalificacion(double nota) {
        if (nota >= 90) {
            return "Sobresaliente";
        } else if (nota >= 70) {
            return "Notable";
        } else if (nota >= Candidato.APROBADO) {
            return "Aprobado";
        } else {
            return "Suspensivo";
        }
    }

    public String obtenerCalificacionCandidato() {
        return Candidato.obtenerCalificacion(this.calcularNotaFinal());
    }
}

```

Método `toString`

El método `toString` es un método que toda clase tiene (heredado de la clase `Object`) y que se supone que debería retornar una representación del objeto actual como una cadena. Habitualmente se indica el nombre de la clase y los valores de los campos. El método `toString` se usa en ciertos lugares de una aplicación Java cuando no se indica qué método queremos emplear.

Si no indicamos este método se empleará el definido en la clase Object que por defecto muestra:

```
Clase@referencia
```

donde referencia es la referencia del objeto (como si fuera la dirección de memoria).

Añadir método `toString` a la clase `Candidato`

```
@Override  
public String toString() {  
    return "Candidato{" +  
        "nombre='" + this.nombre + '\'' +  
        ", prueba1=" + this.prueba1 +  
        ", prueba2=" + this.prueba2 +  
        ", prueba3=" + this.prueba3 +  
        ", notaFinal=" + this.calcularNotaFinal() +  
        ", aprobado=" + this.haAprobado() +  
        '}';  
}
```

Método `equals` / `hashCode`

Al igual que el método anterior, toda clase hereda de la clase Object el método `equals`. Por defecto, el método `equals` hace lo mismo que `==`, es decir dos variables de tipo objeto son iguales si contienen la misma información. En el caso de los objetos lo que se almacena es la referencia al mismo por lo que, por defecto, dos variables son iguales si tienen la misma referencia, es decir, si apuntan al mismo objeto.

Modificamos ahora el método `equals` para que diga que dos candidatos son iguales si tienen el mismo nombre:

```
@Override  
public boolean equals(Object otro) {  
    if (otro == null) {  
        return false;  
    }  
    if (this.getClass() != otro.getClass()) {  
        return false;  
    }  
    Candidato cOtro = (Candidato) otro;  
    return this.nombre.equalsIgnoreCase(cOtro.nombre);  
}
```

El método `hashCode` retorna un entero basándose en la información de la clase. Es importante definirlo cuando vamos a trabajar con colecciones basadas en hashes (como `HashSet` y `HashMap`) como veremos más adelante. El `hashCode` se emplea en estas colecciones para agrupar valores con el mismo hash de modo que sea más fácil localizarlas.

Si vamos a emplear ese tipo de colecciones es imprescindible que si dos objetos son iguales (método `equals`) deben retornar el mismo `hashCode`. De este modo, se deben emplear los mismos campos de la clase que se emplean en el `equals` para calcular el `hashCode`.

De este modo, es recomendable redefinir el `hashCode` si redefinimos el método `equals`. A partir de la versión 8 de Java, disponemos de una clase `Objects` que tiene un método `hash` al que se le pueden pasar una colección de objetos (campos) y nos retorna un hash.

En nuestro caso, necesitamos un `hashCode` que incluya únicamente el nombre:

```
@Override  
public int hashCode() {  
    return Objects.hash(this.nombre);  
}
```

Clase `Candidato` completa con todos los cambios

```
import java.util.Objects;  
  
public class Candidato {  
    // Propiedad estática (de clase) - constante  
    public static final double APROBADO = 60.0;  
  
    // Propiedades de instancia (ahora privadas)  
    private String nombre;  
    private double prueba1;  
    private double prueba2;  
    private double prueba3;  
  
    // Constructores  
    public Candidato() {  
        this.nombre = "Sin nombre";  
        this.prueba1 = 0;  
        this.prueba2 = 0;  
        this.prueba3 = 0;  
    }  
  
    public Candidato(String nombre) {  
        this();  
        this.setNombre(nombre);  
    }  
  
    public Candidato(String nombre, double prueba1, double prueba2, double prueba3) {  
        this(nombre);  
        this.setPrueba1(prueba1);  
        this.setPrueba2(prueba2);  
        this.setPrueba3(prueba3);  
    }  
}
```

```

// Métodos getter
public String getNombre() {
    return this.nombre;
}

public double getPrueba1() {
    return this.prueba1;
}

public double getPrueba2() {
    return this.prueba2;
}

public double getPrueba3() {
    return this.prueba3;
}

// Métodos setter con validación
public void setNombre(String nombre) {
    if (nombre != null && !nombre.trim().isEmpty()) {
        this.nombre = nombre;
    } else {
        throw new IllegalArgumentException("El nombre no puede estar vacío");
    }
}

public void setPrueba1(double prueba1) {
    if (prueba1 >= 0 && prueba1 <= 10) {
        this.prueba1 = prueba1;
    } else {
        throw new IllegalArgumentException("La nota debe estar entre 0 y 10");
    }
}

public void setPrueba2(double prueba2) {
    if (prueba2 >= 0 && prueba2 <= 10) {
        this.prueba2 = prueba2;
    } else {
        throw new IllegalArgumentException("La nota debe estar entre 0 y 10");
    }
}

public void setPrueba3(double prueba3) {
    if (prueba3 >= 0 && prueba3 <= 10) {
        this.prueba3 = prueba3;
    } else {
        throw new IllegalArgumentException("La nota debe estar entre 0 y 10");
    }
}

// Método para establecer todas las notas a la vez
public void establecerNotas(double nota1, double nota2, double nota3) {
    this.setPrueba1(nota1);
    this.setPrueba2(nota2);
    this.setPrueba3(nota3);
}

// Método sobrecargado: permite establecer una nota específica
public void establecerNotas(int numeroPrueba, double nota) {
    switch (numeroPrueba) {
        case 1:
            this.setPrueba1(nota);
            break;
        case 2:
            this.setPrueba2(nota);
            break;
        case 3:
            this.setPrueba3(nota);
            break;
        default:
            throw new IllegalArgumentException("Número de prueba inválido");
    }
}

// Métodos de cálculo
public double calcularPorcentajePrueba1() {
    return this.prueba1 * 0.4;
}

public double calcularPorcentajePrueba2() {
    return this.prueba2 * 0.35;
}

public double calcularPorcentajePrueba3() {
    return this.prueba3 * 0.25;
}

public double calcularNotaFinal() {
    return this.calcularPorcentajePrueba1() +
        this.calcularPorcentajePrueba2() +
        this.calcularPorcentajePrueba3();
}

```

```

public boolean haAprobado() {
    return this.calcularNotaFinal() >= Candidato.APROBADO;
}

public static String obtenerCalificacion(double nota) {
    if (nota >= 90) {
        return "Sobresaliente";
    } else if (nota >= 70) {
        return "Notable";
    } else if (nota >= Candidato.APROBADO) {
        return "Aprobado";
    } else {
        return "Suspensos";
    }
}

public String obtenerCalificacionCandidato() {
    return Candidato.obtenerCalificacion(this.calcularNotaFinal());
}

// Método toString
@Override
public String toString() {
    return "Candidato{" +
        "nombre=" + this.nombre +
        ", pruebal=" + this.pruebal +
        ", prueba2=" + this.prueba2 +
        ", prueba3=" + this.prueba3 +
        ", notafinal=" + this.calcularNotaFinal() +
        ", aprobado=" + this.haAprobado() +
        '}';
}

// Método equals
@Override
public boolean equals(Object otro) {
    if (otro == null) {
        return false;
    }
    if (this.getClass() != otro.getClass()) {
        return false;
    }
    Candidato cOtro = (Candidato) otro;
    return this.nombre.equalsIgnoreCase(cOtro.nombre);
}

// Método hashCode
@Override
public int hashCode() {
    return Objects.hash(this.nombre);
}
}

```

Clase Programa actualizada para usar la nueva versión de Candidato

```

public class Programa {
    public static void main(String[] args) {
        // Crear candidatos usando los constructores
        Candidato candidato1 = new Candidato("Juan Pérez", 7.5, 6.8, 5.9);
        Candidato candidato2 = new Candidato("María López");

        // Establecer notas para el segundo candidato
        candidato2.establecerNotas(1, 4.2);
        candidato2.establecerNotas(2, 5.5);
        candidato2.establecerNotas(3, 6.0);

        // Crear un tercer candidato con el mismo nombre que el primero
        Candidato candidato3 = new Candidato("Juan Pérez", 8.0, 7.5, 9.0);

        // Mostrar resultados detallados
        System.out.println("Resultados de los candidatos:");
        System.out.println("-----");

        mostrarResultadosCandidato(candidato1);
        System.out.println();
        mostrarResultadosCandidato(candidato2);
        System.out.println();

        // Probar el método toString
        System.out.println("Usando toString():");
        System.out.println(candidato1);
        System.out.println();

        // Probar equals y hashCode
        System.out.println("Comparación de candidatos:");
        System.out.println("candidato1 equals candidato2: " + candidato1.equals(candidato2));
        System.out.println("candidato1 equals candidato3: " + candidato1.equals(candidato3));
        System.out.println("hashCode candidato1: " + candidato1.hashCode());
        System.out.println("hashCode candidato3: " + candidato3.hashCode());
    }

    // Método auxiliar para mostrar los resultados de un candidato
    private static void mostrarResultadosCandidato(Candidato candidato) {

```

```

        System.out.println("Candidato: " + candidato.getNombre());
        System.out.println("Prueba 1: " + candidato.getPrueba1() +
                           " (Porcentaje: " + candidato.calcularPorcentajePrueba1() + ")");
        System.out.println("Prueba 2: " + candidato.getPrueba2() +
                           " (Porcentaje: " + candidato.calcularPorcentajePrueba2() + ")");
        System.out.println("Prueba 3: " + candidato.getPrueba3() +
                           " (Porcentaje: " + candidato.calcularPorcentajePrueba3() + ")");
        System.out.println("Nota final: " + candidato.calcularNotaFinal());
        System.out.println("¿Ha aprobado?: " + (candidato.haAprobado() ? "Sí" : "No"));
        System.out.println("Calificación: " + candidato.obtenerCalificacionCandidato());
    }
}

```

La salida del programa sería:

```

Resultados de los candidatos:
-----
Candidato: Juan Pérez
Prueba 1: 7.5 (Porcentaje: 3.0)
Prueba 2: 6.8 (Porcentaje: 2.38)
Prueba 3: 5.9 (Porcentaje: 1.475)
Nota final: 6.855
¿Ha aprobado?: Sí
Calificación: Aprobado

Candidato: María López
Prueba 1: 4.2 (Porcentaje: 1.68)
Prueba 2: 5.5 (Porcentaje: 1.925)
Prueba 3: 6.0 (Porcentaje: 1.5)
Nota final: 5.105
¿Ha aprobado?: No
Calificación: Suspenso

Usando toString():
Candidato{nombre='Juan Pérez', prueba1=7.5, prueba2=6.8, prueba3=5.9, notaFinal=6.855, aprobado=true}

Comparación de candidatos:
candidato1 equals candidato2: false
candidato1 equals candidato3: true
hashCode candidato1: -1290376179
hashCode candidato3: -1290376179

```

En este ejemplo completo:

1. Hemos encapsulado las propiedades haciendo las privadas y proporcionando métodos getter y setter.
2. Los setters incluyen validación para asegurar que los datos son correctos.
3. Hemos implementado tres constructores diferentes.
4. Hemos sobrescrito el método `toString()` para proporcionar una representación textual útil del objeto.
5. Hemos sobrescrito los métodos `equals()` y `hashCode()` para que dos candidatos se consideren iguales si tienen el mismo nombre.
6. La clase Programa demuestra el uso de todos estos métodos y muestra cómo los candidatos con el mismo nombre son considerados iguales y tienen el mismo `hashCode`.

2.6. Actividad 8 - Constructores

Instrucciones

Se pide crear una clase `Persona` teniendo en cuenta lo siguiente:

- Las propiedades de instancia (privadas) de la clase son :
- `nombre`: String
- `edad`: entero
- `DNI`: String (inicializado a "00000000X")
- `sexo`: char (será una `H` para hombre, una `M` para mujer o una `N` para no binario). Por defecto será una `'M'`
- `peso`: en kilos. double
- `altura`: en metros. double
- Se definirá una propiedad de clase llamada `numPersonas` inicializada a 0.
- Se definirán **métodos de acceso** (`set/get`) para cada uno de los campos excepto el método `setDni` que se indicará cómo hacerlo más adelante.

En el caso del `sexo`:

- si se pasa alguno de los tres caracteres indicados se asignará el carácter en **Mayúsculas**
- si no es ninguno, **no se modificará el campo**

Se definirán los siguientes constructores:

- Un constructor por defecto.
- Un constructor con el nombre, edad y sexo.
- Un constructor con todos los campos como parámetro.

NOTA: En todos los casos se incrementará el campo `numPersonas` en 1.

Los métodos que se implementaran son (hay que pensar si son de instancia o de clase!):

- `getIMC()`: El **IMC** (**Índice de Masa Corporal**) es un indicador para medir la obesidad. Se calcula mediante la expresión

$$\frac{\text{peso}(kg)}{\text{altura}(m)^2}$$

El método retornará el IMC de la persona actual

- `getComposicionCorporal(double imc)` : Este método recibirá el **IMC** y retornará una cadena en función de la siguiente tabla:

Composición corporal	Índice de masa corporal (IMC)
Peso inferior al normal	Menos de 18.5
Normal	18.5 – 24.9
Peso superior al normal	25.0 – 29.9
Obesidad	Más de 30.0

- `isMayorDeEdad()`: indica si es mayor de edad, devuelve un booleano.
- `getSexoString()`: Retornará el sexo de la persona como una cadena (**Hombre**, **Mujer**, **No Binario**)
- `isValidoDNI(String dni)` : este método retornará un valor booleano indicando si el DNI es correcto o no. Para comprobar el DNI:
 - cogemos las primeras **8 cifras** del `dni` (`dni.substring(0,8)`)
 - las convertimos a un `int` (`Integer.parseInt`)
 - obtenemos el `resto` de dividir el número entre `23`
 - obtenemos la letra de la siguiente tabla (como un `char`):

Tabla de Correspondencia entre Restos y Letras

RESTO	LETRA
0	T
1	R
2	W
3	A
4	G
5	M
6	Y
7	F
8	P
9	D
10	X
11	B
12	N
13	J
14	Z
15	S
16	Q
17	V
18	H
19	L
20	C
21	K
22	E

NOTA: Una forma sencilla de hacerlo consiste en almacenar todas las letras en un String ("TRWA..."). La letra que buscamos será la que esté en la posición

indicada por el resto (`charAt(resto)`)

Si la letra obtenida coincide con la última del dni (`dni.charAt(8)`) retornamos true y, si no, false

- `toString()`: devuelve toda la información del objeto.
- `equals(Persona otra)` : Retornará true si la otra persona tiene el mismo DNI que la actual

Definir el método `setDni` para que:

- Convierta el dni recibido a mayúsculas (`toUpperCase`)
- Compruebe si el dni es válido con el método definido previamente:
- Si es válido, lo almacenamos en el campo dni
- Si no lo es, no hacemos nada

Definir una clase ejecutable que realice las siguientes tareas:

- Repetir el siguiente proceso mientras el usuario quiera
- Pedir los datos de una Persona y crearlos empleando alguno de los tres constructores
- NOTA: Dado que el sexo lo queremos como un char y al pedirlo al usuario obtendremos un String (imaginemos que lo guardamos en resp), para convertir la cadena en un char emplearemos `resp.charAt(0)`.
- Mostrar la información de la persona según el siguiente formato:

```
Número de Personas Introducidas : numPersonas
Nombre : nombre
DNI : dni
Sexo : sexoComoTexto
Edad : edad      Mayor de Edad : Sí/No
Altura : altura   Peso : peso
IMC : imc        Composición Corporal : texto
```

Adicional (para los que acaben antes)

- Al pedir los datos, validar el campo sexo para que sólo se pueda introducir un carácter válido (dar un mensaje de error en caso contrario y volver a pedirlo)
- Modificar el método `setDni` para que si la longitud del dni es de 8 caracteres (método `length()`) calcule la letra que le correspondería y la añada al dni antes de guardarla. Probar a meter una persona sin indicar la letra y comprobar que le asigna la letra adecuada
- Hacer un pequeño programa en el que se creen tres personas con diferentes datos (los introducís vosotros mismos). Pedir al usuario un dni y:
- Si alguna de las tres personas tiene ese dni, mostrar sus datos por pantalla (`toString`)
- Si ninguna lo tiene, dar un mensaje de error.

2.7. Actividad 9 - Creación de Clases

Instrucciones

Queremos controlar los envíos de una empresa de mensajería que dispone de cuatro tipo de servicios diferentes. De cada envío se almacenará la siguiente información:

- id : un entero con el id del envío
- destinatario : una cadena con información del destinatario
- remitente : idem con información del remitente
- peso : peso en gramos del paquete
- tipo : tipo de envío. Un valor entre 1 y 4 que corresponde con los datos de la siguiente tabla (emplearemos cuatro constantes para representar cada tipo de envío).

La tabla con la información de las tarifas es la siguiente (donde la columna precio/200 gramos indica un coste adicional a pagar en función del peso del envío :

Código	Tipo	Coste Base	Coste / 200 gramos
1	Normal	7€	0,50 €
2	Express	10€	0,70 €
3	24 Horas	12€	0,80 €
4	8 Horas	15€	0,90 €

Es decir, un paquete de 4.500 g enviado por modalidad Express costaría 10 € fijos más 16,1 € por el peso ($4500 / 200 \Rightarrow 22,5 \Rightarrow 23 * 0,7 \Rightarrow 16,1 \text{ €}$). Por tanto el total sería 26,1€. Nota: si el resto (%) de dividir el peso entre 200 no es cero se debe tomar una unidad más. Se pide diseñar una clase Envío que contenga los cuatro campos identificados previamente, una constante por cada tipo de envío, así como los siguientes métodos:

- un constructor sin argumentos y otro que reciba todos los datos. En ambos casos se asignará el id del envío automáticamente de la manera habitual (empleando una propiedad estática para almacenar el número de envíos).
- métodos set/get para cada campo excepto el id que sólo tendrá método get. En el caso del método `setTipoEnvio` si el tipo de envío proporcionado no es válido (no es un valor entre 1 y 4) se asignará como código el correspondiente a un envío normal.
- un método `getDescripcionTipo` que retornará el tipo de envío como una cadena

- un método getCosteBase que retornará el coste base asociado al envío (en el ejemplo 10)
- un método getCoste200 que retornará el coste asociado a cada 200 gramos (en el ejemplo 0,7)
- un método getCostePeso que retornará el coste asociado al peso del envío. (en el ejemplo 15,1€)
- un método getCoste que retornará el coste total del envío (26,1€)
- un método compareTo que recibirá otro objeto de tipo Envío y retornará un valor mayor que 0 si el coste de éste envío es mayor que el del que nos pasan, igual a 0 si ambos costes son iguales y menor que 0 si el coste del envío actual es menor que el del que nos pasan.
- un método estático denominado mismoRemitente que recibirá dos Envíos y retornará true si ambos son del mismo remitente y false en caso contrario.
- un método equals en base al id

La empresa, además, tiene estipulados una serie de descuentos en función del coste total a aplicar a cada envío (que usaremos en el programa). Dichos descuentos se indican en la siguiente tabla:

Coste	% descuento
< 50 €	0
Entre 50 y 100 €	3%
100 € o más	5%

Se pide diseñar un programa que permita obtener los costes correspondientes a los envíos en los diferentes días de la semana (sólo se consideran los días laborables). El proceso a realizar será el siguiente:

- Para cada día de la semana (de 1 a 5):
 - Se repetirá el siguiente proceso:
 - Se pedirá el remitente, el destinatario, el tipo de envío y el peso (en gramos) del envío. Se deberá validar que el peso esté entre 0 y 1.000.000 gramos (mientras no sea así se dará un mensaje de error y se volverá a pedir el peso)
 - Se creará un nuevo envío condichos datos
 - Se mostrará el tipo (descripción) de servicio elegido, el coste base y el precio por cada 200 gramos.
 - Se mostrará por pantalla el coste del envío, el descuento y el coste con el descuento incluido.
 - Se preguntará si se desea procesar otro envío
- Cuando no haya más envíos se mostrará por pantalla el nombre del día, la suma de los costes (sin el descuento), la suma de los descuentos y la suma de los costes totales
- Al acabar con los cinco días laborables:
 - Se mostrará la suma de todos los costes, descuentos y costes con descuentos de toda la semana.
 - Se mostrará la suma de los pesos correspondiente a todos los envíos
 - Por último se indicará cuántos días ha habido en los que se hayan obtenido costes totales (sin descuento) superiores a 200€.

ADICIONAL: Modificar el programa para que, al final, muestre cuál ha sido el envío con mayor coste total. Para ello:

- se creará una variable de tipo Envío, llamada maximo
- dentro de la repetitiva por envío y si es la primera vez (emplear una variable booleana), se asignará el envío actual al máximo
- si no lo es se comprobará (compareTo) si el envío actual es mayor que el máximo y, en ese caso, se asignará como máximo el actual

2.8. Algunas clases útiles de Java

Clases útiles

La mayor parte de los lenguajes de programación se basan en **librerías preconstruidas de clases** para proveer de cierta funcionalidad a los programas. En la plataforma Java 2 estos conjuntos de clases relacionadas, denominados **paquetes**, dependen de la edición de Java que se emplee.

Cada una de las ediciones dispone de un **kit de desarrollo de software (SDK)** y de un **entorno de ejecución Java (JRE)** diferentes.

Las ediciones que vamos a ver en este ciclo son:

- **Edición Estándar (J2SE)**: Contiene las clases que conforman el núcleo de Java y es útil para la mayor parte de las aplicaciones.
- **Edición Empresarial (J2EE)**: Incluye las clases de J2SE y clases adicionales específicas para desarrollos empresariales en varias capas incluyendo servlets Java, Java Server Pages (JSP), XML, Enterprise Beans y un control de transacciones flexible.

Paquetes incluidos en J2SE

En la tabla siguiente se pueden ver algunos de los paquetes que componen la edición J2SE:

Paquete	Nombre	Descripción
Lenguaje	java.lang	Clases que forman el núcleo principal de Java
Utilidades	java.util	Soporte para utilidades de estructuras de datos y de manipulación de fechas y horas
Fecha / Hora	java.time	Nuevas clases para manipular fechas / horas e intervalos de tiempo

Paquete	Nombre	Descripción
I/O	java.io	Soporte para diferentes tipos de E/S incluyendo ficheros
Texto	java.text	Soporte para la localización regional de texto, fechas, números y mensajes
Matemáticas	java.math	Clases para la realización de cálculos con enteros de tamaño indefinido y valores en coma flotante de precisión indefinida
Abstract Windows Toolkit (AWT)	java.awt	Clases para el diseño de interfaces de usuario, manipulación de imágenes y gráficos y los modelos de manipulación de eventos
Swing	javax.swing	Clases para crear componentes ligeros que se comporten de manera similar en cualquier plataforma (versiones ligeras de los componentes AWT con nuevos componentes)
Javax	javax	Extensiones al lenguaje Java (además de javax.swing existen otros como javax.sound, javax.rmi, javax.accessibility y es posible incorporar y/o desarrollar nuevas extensiones).
Beans	java.beans	Clases para desarrollar Java Beans que son clases que sirven como componentes reusables
Reflejo	java.reflection	Clases usadas para examinar y manipular clases en tiempo de ejecución
SQL	java.sql	Clases para acceder y manipular diversas bases de datos relacionales. También se conoce como JDBC 2.0 (java DataBase Connectivity)
RMI	java.rmi	Clases de soporte para la programación distribuida. RMI (Remote Method Invocation) permite a un objeto ejecutándose en una JVM invocar métodos de otro objeto ejecutándose en otra JVM
Red	java.net	Clases para el desarrollo de aplicaciones de red basadas en el protocolo TCP/IP
Seguridad	java.security	Soporte para seguridad criptográfica. Se incluyen clases para implementar el control de acceso y prevenir la ejecución de código que no sea de confianza y clases para autenticar otras clases y objetos

Clases en java.lang

Object (java.lang.Object)

La clase **Object** es la clase padre de todas las demás clases de Java.

Método	Descripción
Object()	Constructor
protected Object clone()	Retorna una copia del objeto. Hay que sobreescribirlo para que pueda ser usado en las clases hijas
boolean equals(Object)	Retorna <code>true</code> si el objeto actual es igual al que se recibe como argumento. En principio hace lo mismo que el operador == (dos objetos son iguales si son el mismo objeto). Está pensado para que sea sobreescrito por las clases
String toString()	Retorna una cadena que representa el objeto. Por defecto retorna <code>clase@referencia</code> . Se emplea en aquellos sitios donde se requiere una representación del objeto como una cadena y cuando se concatena el objeto con una cadena. Está pensado para que sea sobreescrito por las clases

Ejemplo:

```
public class MiClase {
    private int valor;

    public MiClase(int valor) {
        this.valor = valor;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof MiClase) {
            return this.valor == ((MiClase)obj).valor;
        }
        return false;
    }

    @Override
    public String toString() {
        return "MiClase[valor=" + valor + "]";
    }
}

// Uso
MiClase obj1 = new MiClase(5);
MiClase obj2 = new MiClase(5);
System.out.println(obj1.equals(obj2)); // true
System.out.println(obj1.toString()); // MiClase[valor=5]
```

Character (java.lang.Character)

La clase **Character** permite manipular caracteres individuales y se considera una clase envolvente del tipo básico `char`.

Método	Descripción
Character(char)	Constructor
int compareTo(char \ Object)	Retorna >0 si el carácter es mayor que el que se pasa como parámetro, 0 si son iguales y <0 si es menor
char charValue()	Retorna un char que representa el carácter
static boolean isDigit(char)	Retorna true si el carácter es un dígito
static boolean isLowerCase(char)	Retorna true si el carácter está en minúsculas
static boolean isUpperCase(char)	Retorna true si el carácter está en mayúsculas
static char toLowerCase(char)	Retorna el carácter en minúsculas
static char toUpperCase(char)	Retorna el carácter en mayúsculas
static String toString(char)	Retorna el carácter como un String

Ejemplo:

```
Character c1 = new Character('A');
Character c2 = new Character('a');

System.out.println(c1.charValue()); // A
System.out.println(Character.isUpperCase(c1)); // true
System.out.println(Character.toLowerCase('Z'));// z
System.out.println(Character.isDigit('5'));// true
System.out.println(c1.compareTo(c2)); // valor negativo porque 'A' < 'a' en Unicode
```

String (java.lang.String)

La clase **String** permite manipular cadenas de texto no modificables.

Método	Descripción
String(byte[] \ char[] \ String \ StringBuffer ...)	Construye un objeto String a partir de una matriz de bytes, de caracteres, de otro objeto String o de un objeto StringBuffer
char charAt(int)	Retorna el carácter que se encuentra en la posición indicada (empezando desde 0)
int compareTo(String)	Retorna <0 si la cadena es lexicográficamente menor, 0 si son iguales y >0 si el objeto String es mayor que el proporcionado como argumento
int compareToIgnoreCase(String)	Similar a compareTo pero sin considerar mayúsculas y minúsculas
String concat(String)	Añade al String actual el String proporcionado
boolean equals(String \ Object)	Retorna true si las cadenas son iguales
boolean equalsIgnoreCase(String)	Retorna true si las cadenas son iguales, ignorando mayúsculas y minúsculas
int indexOf(char)	Retorna la posición de la primera aparición de un carácter en la cadena
int indexOf(String)	Retorna la posición de la primera aparición de una subcadena
int indexOf(char, int)	Retorna la posición de la primera aparición de un carácter a partir de la posición indicada
int indexOf(String, int)	Retorna la posición de la primera aparición de una subcadena a partir de la posición indicada
int length()	Devuelve la longitud de la cadena
String substring(int, int)	Devuelve un objeto String con los caracteres entre las dos posiciones indicadas (incluyendo la primera y excluyendo la segunda)
String toLowerCase()	Retorna la cadena en minúsculas
String toUpperCase()	Retorna la cadena en mayúsculas
static String valueOf(char \ int \ double \ float \ boolean \ long \ Object)	Retorna el dato pasado como un objeto String

Ejemplo:

```
String str = new String();
str = "Hola a Todos";
System.out.println(str); // Hola a Todos

String s1 = "Java";
String s2 = "java";
System.out.println(s1.equals(s2)); // false
System.out.println(s1.equalsIgnoreCase(s2)); // true
System.out.println(s1.charAt(0)); // J
System.out.println(s1.indexOf('a'));// 1
System.out.println(s1.substring(1, 3)); // av
System.out.println(s1.toUpperCase()); // JAVA
```

```
System.out.println(String.valueOf(123.45)); // 123.45
```

StringBuffer (java.lang.StringBuffer)

La clase **StringBuffer** es similar a **String**, pero con la diferencia de que es **modificable**.

Método	Descripción
StringBuffer()	Crea un objeto vacío
StringBuffer(int)	Crea un objeto vacío pero con capacidad inicial para int caracteres
StringBuffer(String)	Crea un objeto StringBuffer a partir de un String
StringBuffer append(boolean \ char \ char[] \ int \ long \ float \ double \ String \ StringBuffer)	Añade el dato al final del objeto
int capacity()	Indica el espacio reservado disponible para el objeto
char charAt(int)	Devuelve el carácter de la posición indicada
StringBuffer delete(int, int)	Elimina los caracteres entre las posiciones dadas
int indexOf(String)	Retorna la posición de una subcadena dentro del objeto
StringBuffer insert(tipo, int)	Inserta el dato en la posición indicada
int length()	Retorna el número de caracteres del objeto
StringBuffer reverse()	Retorna el objeto al revés
StringBuffer setLength(int)	Especifica el tamaño del objeto
String toString()	Retorna un String a partir del objeto

Ejemplo:

```
StringBuffer sb = new StringBuffer("Hola");
sb.append(" Mundo");
System.out.println(sb); // Hola Mundo

sb.insert(5, "Gran ");
System.out.println(sb); // Hola Gran Mundo

sb.delete(4, 9);
System.out.println(sb); // Hola Mundo

System.out.println(sb.reverse()); // odnuM aloH
System.out.println(sb.capacity()); // Muestra la capacidad actual
System.out.println(sb.charAt(0)); // o (después del reverse)
```

Number (java.lang.Number)

La clase **Number** es una clase abstracta y es la superclase de las clases **Byte**, **Double**, **Float**, **Integer**, **Long** y **Short**.

Método	Descripción
byte byteValue()	Retorna el valor como byte
double doubleValue()	Retorna el valor como double
float floatValue()	Retorna el valor como float
int intValue()	Retorna el valor como int
long longValue()	Retorna el valor como long
short shortValue()	Retorna el valor como short

Ejemplo:

```
Integer num = new Integer(42);
System.out.println(num.doubleValue()); // 42.0
System.out.println(num.byteValue()); // 42

Double d = new Double(123.456);
System.out.println(d.intValue()); // 123
System.out.println(d.isNaN()); // false

// BigInteger y BigDecimal
import java.math.BigInteger;
import java.math.BigDecimal;

BigInteger bi1 = new BigInteger("123456789012345678901234567890");
BigInteger bi2 = new BigInteger("987654321098765432109876543210");
System.out.println(bi1.add(bi2)); // 1111111101111111101111111100
```

```

BigDecimal bd1 = new BigDecimal("123.456");
BigDecimal bd2 = new BigDecimal("789.012");
System.out.println(bd1.multiply(bd2)); // 97408.757472

```

Math (java.lang.Math)

Es una clase **final** que proporciona métodos estáticos para las operaciones matemáticas más habituales.

Método	Descripción
static tipo abs(tipo)	Devuelve el valor absoluto del dato especificado
static double cos\ sin\ tan\ acos\ asin\ atan(double)	Devuelve el coseno, seno, tangente, arcocoseno, arcoseno o arcotangente del valor especificado
static double ceil(double)	Devuelve el valor más bajo que represente un entero y que sea superior al valor
static double exp(double)	Devuelve (e^{\text{double}})
static double floor(double)	Devuelve el valor más alto que represente un entero y que sea menor que el valor
static double log(double)	Devuelve el logaritmo natural del valor
static tipo max(tipo, tipo)	Devuelve el mayor de dos valores del mismo tipo
static tipo min(tipo, tipo)	Devuelve el menor de dos valores del mismo tipo
static double pow(double, double)	Devuelve el primer valor elevado al segundo
static double random()	Devuelve un valor pseudoaleatorio mayor o igual a 0.0 y menor que 1.0
static long round(double)	Devuelve el long más cercano al argumento
static int round(float)	Devuelve el int más cercano al argumento
static double sqrt(double)	Devuelve la raíz cuadrada del dato

Ejemplo:

```

System.out.println(Math.abs(-10)); // 10
System.out.println(Math.sqrt(25)); // 5.0
System.out.println(Math.pow(2, 3)); // 8.0
System.out.println(Math.max(10, 20)); // 20
System.out.println(Math.ceil(4.3)); // 5.0
System.out.println(Math.floor(4.7)); // 4.0
System.out.println(Math.round(4.5)); // 5

// Generar número aleatorio entre 1 y 100
int aleatorio = (int) (Math.random() * 100) + 1;
System.out.println(aleatorio);

```

System (java.lang.System)

La clase **System** permite acceder a los recursos de la plataforma en la que se ejecuta Java.

Método	Descripción
static void arraycopy(Object, int, Object, int, int)	Copia una matriz de objetos a partir de una posición sobre otra posición de la matriz de destino
static long currentTimeMillis()	Retorna la fecha/hora actual en milisegundos desde el 1 de enero de 1970
static void exit(int)	Finaliza la ejecución de la JVM con el código de estado proporcionado
static void gc()	Invoca al garbage collector
static Properties getProperties()	Retorna las propiedades del sistema
static String getProperty(String)	Recupera la propiedad del sistema especificada por el objeto String
static void runFinalization()	Ejecuta el método finalize() de todos los objetos pendientes de ejecutarlo
static String setProperty(String, String)	Modifica una propiedad del sistema con un nuevo valor

Ejemplo:

```

// Copiar un array
int[] origen = {1, 2, 3, 4, 5};
int[] destino = new int[5];
System.arraycopy(origen, 0, destino, 0, origen.length);

for (int i : destino) {
    System.out.print(i + " "); // 1 2 3 4 5
}

// Obtener tiempo actual
long tiempoActual = System.currentTimeMillis();
System.out.println("\nTiempo actual en ms: " + tiempoActual);

// Obtener propiedades del sistema
System.out.println("Sistema operativo: " + System.getProperty("os.name"));

```

```
System.out.println("Versión de Java: " + System.getProperty("java.version"));
```

Clases en java.util

Date (java.util.Date)

La clase **Date** representa una fecha y hora con precisión de milisegundos.

Método	Descripción
public Date()	Constructor que crea un objeto con la fecha y hora actuales
boolean after(Date)	Retorna true si la fecha del objeto es posterior a la indicada
boolean before(Date)	Retorna true si la fecha del objeto es anterior a la indicada
long getTime()	Retorna el número de milisegundos desde el 1 de enero de 1970
void setTime(long)	Establece la fecha correspondiente a los milisegundos indicados
String toString()	Retorna una representación en formato GMT de la fecha

Ejemplo:

```
Date fechaActual = new Date();
System.out.println("Fecha actual: " + fechaActual);

// Crear fecha específica (milisegundos desde 1970)
Date fechal = new Date(1000000000000L); // 9 de septiembre de 2001
System.out.println("Fecha 1: " + fechal);

// Comparar fechas
System.out.println("¿Fecha actual es posterior a fechal? " + fechaActual.after(fechal));
System.out.println("¿Fecha actual es anterior a fechal? " + fechaActual.before(fechal));

// Obtener milisegundos
System.out.println("Milisegundos desde 1970: " + fechaActual.getTime());

### Calendar / GregorianCalendar (java.util.Calendar / java.util.GregorianCalendar)
```

La clase **Calendar** es una clase abstracta para manipular fechas. La clase **GregorianCalendar** deriva de **Calendar**.

```
| Método | Descripción |
| --- | --- |
| `abstract void add(int, int)` | Suma al campo indicado el valor especificado |
| `void clear()` | Limpia todos los campos del calendario |
| `void clear(int)` | Limpia el campo indicado |
| `int get(int)` | Obtiene el valor del campo indicado |
| `int getFirstDayOfWeek()` | Retorna el primer día de la semana según la localización |
| `static Locale[] getAvailableLocales()` | Retorna una matriz con todas las configuraciones regionales disponibles |
| `static Calendar getInstance()` | Obtiene un calendario basado en la configuración regional del equipo |
| `static Calendar getInstance(TimeZone, Locale)` | Obtiene un calendario basado en la zona horaria y la configuración regional especificada |
| `Date getTime()` | Retorna la fecha actual basada en los campos del calendario |
| `long getTimeInMillis()` | Retorna la fecha en milisegundos |
| `void set(int, int)` | Establece el valor del campo indicado |
| `void set(int, int, int)` | Establece el año, mes y día |
| `void set(int, int, int, int, int)` | Establece el año, mes, día, hora y minuto |
| `void set(int, int, int, int, int, int)` | Establece el año, mes, día, hora, minuto y segundo |
| `void setTime(Date)` | Actualiza los campos del calendario según la fecha proporcionada |
| `void setTimeInMillis(long)` | Actualiza los campos del calendario según los milisegundos proporcionados |
| `String toString()` | Retorna una cadena con los valores de todos los campos del calendario |
```

Ejemplo:

```
```java
// Obtener calendario actual
Calendar cal = Calendar.getInstance();
System.out.println("Fecha actual: " + cal.getTime());

// Obtener componentes individuales
int año = cal.get(Calendar.YEAR);
int mes = cal.get(Calendar.MONTH) + 1; // +1 porque enero es 0
int dia = cal.get(Calendar.DAY_OF_MONTH);
System.out.println("Fecha: " + dia + "/" + mes + "/" + año);

// Modificar fecha (sumar un mes)
cal.add(Calendar.MONTH, 1);
System.out.println("Fecha en un mes: " + cal.getTime());

// Establecer una fecha específica
Calendar fechaEspecifica = Calendar.getInstance();
fechaEspecifica.set(2023, Calendar.DECEMBER, 31, 23, 59, 59);
System.out.println("Fin de año: " + fechaEspecifica.getTime());

// GregorianCalendar
GregorianCalendar gc = new GregorianCalendar();
System.out.println("¿Es año bisiesto? " + gc.isLeapYear(2024)); // true
```

### Locale (java.util.Locale)

La clase **Locale** permite gestionar las diferencias regionales.

Método	Descripción
Locale(String)	Crea un nuevo objeto <code>Locale</code> para el código de lenguaje especificado
Locale(String, String)	Crea un objeto <code>Locale</code> especificando el lenguaje y el país
Locale(String, String, String)	Crea un objeto <code>Locale</code> especificando el lenguaje, el país y una variante
static Locale[] getAvailableLocales()	Retorna una matriz con las configuraciones locales disponibles
String getCountry()	Retorna el código del país del objeto <code>Locale</code>
static Locale getDefault()	Retorna el objeto <code>Locale</code> por defecto
String getDisplayCountry([Locale])	Retorna el nombre del país del objeto <code>Locale</code> actual o del correspondiente al <code>Locale</code> especificado
String getDisplayLanguage([Locale])	Retorna el nombre del lenguaje del objeto <code>Locale</code> actual o del correspondiente al <code>Locale</code> especificado
static String[] getISOCountries()	Retorna una matriz de los códigos ISO de los países
static String[] getISOLanguages()	Retorna una matriz de los códigos ISO de los lenguajes disponibles
String getLanguage()	Retorna el código del lenguaje del objeto <code>Locale</code>
static voidsetDefault(Locale)	Establece el objeto indicado como <code>Locale</code> por defecto para la JVM
String toString()	Retorna una cadena formada por el código de lenguaje, país y variante separados por guiones bajos

**Ejemplo:**

```
// Crear locales
Locale localeES = new Locale("es", "ES");
Locale localeUS = new Locale("en", "US");
Locale localeFR = new Locale("fr", "FR");

// Obtener información
System.out.println("Locale por defecto: " + Locale.getDefault());
System.out.println("Idioma español: " + localeES.getDisplayLanguage());
System.out.println("País España en inglés: " + localeES.getDisplayCountry(localeUS));

// Listar algunos países disponibles
String[] paises = Locale.getISOCountries();
System.out.println("Algunos códigos ISO de países:");
for (int i = 0; i < 5; i++) {
 System.out.println(paises[i]);
}
```

## Clases en java.text

### NumberFormat (java.text.NumberFormat)

La clase `NumberFormat` permite formatear valores numéricos.

Método	Descripción
String format(tipo)	Formatea el dato pasado y lo retorna como un <code>String</code>
int getMaximumFractionDigits() / void setMaximumFractionDigits(int)	Retorna / asigna el número máximo de decimales
int getMaximumIntegerDigits() / void setMaximumIntegerDigits(int)	Retorna / asigna el número máximo de enteros
int getMinimumFractionDigits() / void setMinimumFractionDigits(int)	Retorna / asigna el número mínimo de decimales
int getMinimumIntegerDigits() / void setMinimumIntegerDigits(int)	Retorna / asigna el número mínimo de enteros
boolean isGroupingUsed() / void setGroupingUsed(boolean)	Retorna / especifica si se deben emplear o no separadores de miles
Object parse(String)	A partir de una cadena que representa un número formateado, retorna el valor correspondiente
static Locale[] getAvailableLocales()	Retorna una matriz con todas las configuraciones locales disponibles
static NumberFormat getCurrencyInstance()	Retorna un objeto para formatear números como moneda
static NumberFormat getCurrencyInstance(Locale)	Retorna un objeto para formatear números como moneda con el <code>Locale</code> especificado
static NumberFormat getInstance()	Retorna un objeto para formatear números
static NumberFormat getInstance(Locale)	Retorna un objeto para formatear números con el <code>Locale</code> especificado

**Ejemplo:**

```
// Formatear números en diferentes locales
double numero = 1234567.89;
Locale localeES = new Locale("es", "ES");
```

```

Locale localeUS = new Locale("en", "US");

NumberFormat nfES = NumberFormat.getInstance(localeES);
NumberFormat nfUS = NumberFormat.getInstance(localeUS);

System.out.println("Número en formato español: " + nfES.format(numero)); // 1.234.567,89
System.out.println("Número en formato inglés: " + nfUS.format(numero)); // 1,234,567.89

// Formatear moneda
NumberFormat monedaES = NumberFormat.getCurrencyInstance(localeES);
NumberFormat monedaUS = NumberFormat.getCurrencyInstance(localeUS);

System.out.println("Moneda en formato español: " + monedaES.format(numero)); // 1.234.567,89 €
System.out.println("Moneda en formato inglés: " + monedaUS.format(numero)); // $1,234,567.89

// Personalizar formato
NumberFormat nf = NumberFormat.getInstance();
nf.setMaximumFractionDigits(2);
nf.setMinimumFractionDigits(2);
nf.setGroupingUsed(true);
System.out.println("Número personalizado: " + nf.format(123456.789)); // 123,456.79

// Parsear un número
try {
 Number n = nfES.parse("1.234,56");
 System.out.println("Número parseado: " + n.doubleValue()); // 1234.56
} catch (java.text.ParseException e) {
 System.out.println("Error al parsear: " + e.getMessage());
}

```

#### **DateFormat / SimpleDateFormat (java.text.DateFormat / java.text.SimpleDateFormat)**

Estas clases permiten controlar el formato de fechas y horas.

Método (DateFormat)	Descripción
String format(Date)	Formatea la fecha indicada y la retorna como un String
static DateFormat getDateInstance()	Retorna un formateador de fecha con formato medio
static DateFormat getDateInstance(int)	Retorna un formateador de fecha con el formato indicado
static DateFormat getDateInstance(int, Locale)	Retorna un formateador de fecha con el formato y el Locale indicados
static DateFormat getDateTimeInstance()	Retorna un formateador de fecha y hora con formato medio
static DateFormat getDateTimeInstance(int, int)	Retorna un formateador de fecha y hora con los formatos indicados
static DateFormat getDateTimeInstance(int, int, Locale)	Retorna un formateador de fecha y hora con los formatos y el Locale indicados
static DateFormat getTimeInstance()	Retorna un formateador de hora con formato medio
static DateFormat getTimeInstance(int)	Retorna un formateador de hora con el formato indicado
static DateFormat getTimeInstance(int, Locale)	Retorna un formateador de hora con el formato y el Locale indicados
Date parse(String)	Obtiene un objeto Date a partir de la cadena indicada

Método (SimpleDateFormat)	Descripción
SimpleDateFormat()	Constructor con formato por defecto
SimpleDateFormat(String)	Emplea la cadena de formato indicada como parámetro
SimpleDateFormat(String, Locale)	Emplea la cadena de formato especificada y el objeto Locale
void applyPattern(String)	Aplica el formato indicado al objeto

#### **Ejemplo:**

```

Date fecha = new Date();
Locale localeES = new Locale("es", "ES");
Locale localeUS = new Locale("en", "US");

// Formatear fecha con DateFormat
DateFormat dfES = DateFormat.getDateInstance(DateFormat.LONG, localeES);
DateFormat dfUS = DateFormat.getDateInstance(DateFormat.LONG, localeUS);

System.out.println("Fecha en español: " + dfES.format(fecha));
System.out.println("Fecha en inglés: " + dfUS.format(fecha));

// Formatear fecha y hora
DateFormat dfTimeES = DateFormat.getTimeInstance(
 DateFormat.MEDIUM, DateFormat.MEDIUM, localeES);
System.out.println("Fecha y hora en español: " + dfTimeES.format(fecha));

// Usar SimpleDateFormat con patrones personalizados
SimpleDateFormat sdf1 = new SimpleDateFormat("dd/MM/yyyy");
SimpleDateFormat sdf2 = new SimpleDateFormat("EEEE, d 'de' MMMM 'de' yyyy", localeES);

```

```

SimpleDateFormat sdf3 = new SimpleDateFormat("HH:mm:ss");

System.out.println("Formato simple: " + sdf1.format(fecha)); // 17/03/2025
System.out.println("Formato completo: " + sdf2.format(fecha)); // lunes, 17 de marzo de 2025
System.out.println("Hora: " + sdf3.format(fecha)); // 14:30:45

```

## 2.9. Actividad 10 - String Date y GregorianCalendar

### Instrucciones

Empleando las clases `String, Date y GregorianCalendar` se pide crear un programa que:

- Crear una función estática llamada `mostrar` que reciba un `GregorianCalendar` y retorne la cadena `dd-mm-yyyy (diaSemana)` en función de sus datos.
- En el método `main`
- Pedir una fecha al usuario en formato `dd/mm/aaaa` y, a partir de ella, crear un objeto `GregorianCalendar`(hay que extraer previamente la parte del día, del mes y del año con los métodos `indexOf, lastIndexOf y substring`). Mostrarla con el método anterior
- Sumar a esa fecha 30 días y, si la fecha resultante cae en `Sábado o en Domingo`, pasarla al lunes siguiente. Mostrarla.
- Supongamos que una empresa paga las facturas los días 15 de cada mes. Modificar la fecha del calendario para que apunte al siguiente día 15, es decir, si es 12 de Marzo se pasaría a 15 de Marzo, si es 24 de Marzo se pasaría a 15 de Abril. Mostrarla
- Indicar mediante mensajes :
  - A qué trimestre corresponde
  - Si es un día laborable o no
  - Cuántos días han pasado desde principio de año y cuántos días quedan hasta el final del año
  - Cuántos días hasta fin de mes
  - ¿Es o ha sido este año bisiesto?
- Hacer que el calendario apunte al siguiente lunes. Mostrarlo
- Indicar qué fecha será vuestro próximo cumpleaños
- ¿Cuántos días han transcurrido desde hoy hasta la fecha especificada en el calendario? Se dará el mensaje `Desde fecha hasta hoy han transcurrido dias dias`. Si la fecha es anterior el mensaje será `Hace dias dias que fue fecha`. Donde `fecha` es la fecha del calendario y `dias` el número de días calculado.

## 2.10. Actividad 11 - Formateadores

### Instrucciones

#### [GitHub](#)

Se pide diseñar una clase que nos permita **gestionar los pagos de un préstamo hipotecario**. En un préstamo hipotecario se emplea el sistema francés que calcula un pago constante para todos los `periodos` (meses) del préstamo. Dicho pago incluye los intereses del capital pendiente de devolver y una cantidad a descontar (amortizar) de dicho capital. Para el siguiente mes la cantidad a pagar de intereses es menor (dado que el capital ha disminuido) con lo que la cantidad a amortizar será mayor (dado que el pago, como ya se ha dicho es siempre el mismo)

La clase `Prestamo` dispondrá de los siguientes campos:

- `capital`: la cantidad de dinero prestada
- `interés`: el tipo de interés (un % anual). Por ejemplo  $0,015 \rightarrow 1,5\%$
- `años`: el número de años a los que se ha concedido el préstamo
- `fecha`: la fecha de formalización del préstamo.

La clase dispondrá de los métodos `set/get` habituales y de un constructor sin argumentos y otro que reciba toda la información.

Además en la clase se deberá definir el siguiente método:

**pago()**: Esta función retornará el pago mensual de la hipoteca. Este pago es el mismo todos los meses y se calcula según la siguiente fórmula (emplear la función `pow` de la clase `Math`):

```
pago = (Co * i) / (1 - (1 + i)^-n)
```

`Co` es el capital `i` es el interés mensual (el interés que tenemos dividido entre 12) `n` es el número de períodos (meses), por tanto `años` multiplicado por 12

Se pide diseñar un programa que nos permita mostrar la tabla de amortización de un préstamo. Es decir cuánto se debe pagar cada periodo, cuánto se amortiza (qué parte del préstamo se paga) y cuánto se paga de intereses.

El programa solicitará los datos de un préstamo. Para ello se empleará un **formateador de números decimales** para el capital, un **formateador de tipo porcentaje** (con dos decimales) para el interés anual y un **formateador de fecha** corta para la fecha de formalización del préstamo. Con esos datos se creará un objeto de tipo `Prestamo`.

Al final del proceso deberá mostrarse el siguiente cuadro de amortización:

Periodo	Fecha	Capital	Pago	Amortización	Intereses
1	fecha	capital	pago	amortización	intereses
...					

Total Pagado : sumaPagos

Intereses Pagados : sumaIntereses

El proceso a realizar será el siguiente

- Guardamos en una variable llamada `capital` el capital inicial y en otra llamada `fecha` la fecha de formalización del préstamo
- Dado que el pago es fijo, **conviene guardarla en una variable**
- **Repetimos** el siguiente proceso **tantas veces como periodos** tenga el préstamo (empezando desde 1)
- Calculamos los `intereses` a pagar en el periodo (el capital multiplicado por el tipo de interés mensual (recordar dividirlo entre 12) )
- Obtenemos la `amortización` (el pago menos los intereses)
- **Mostramos** los datos por pantalla correctamente formateados
- Restamos la `amortización` del `capital`
- Sumamos un `mes` más a la `fecha`

## 2.11. Herencia

### Herencia

Para indicar que una clase desciende de otra debemos incluir la palabra clave `extends` y el nombre de la clase padre (denominada **superclase**) a la hora de declarar la clase hija (denominada **subclase**).

```
public class Clase extends Superclase {
 ...
}
```

- Todas las clases de Java descienden directa o indirectamente de la clase `Object`.
- No se permite que una clase tenga más de una clase padre (no hay herencia múltiple).

Las subclases heredan funcionalidades (propiedades y métodos de instancia) de todas las superclases de las que descienden. Una clase hereda las propiedades y métodos de sus superclases que hayan sido definidos como `public` o `protected`, o en las que no se haya especificado el acceso, siempre y cuando ambas clases pertenezcan al mismo paquete.

#### Propiedades y métodos heredados

- Si una clase declara una propiedad con el mismo nombre que otra de la superclase, la propiedad de la clase oculta a la propiedad de la superclase. Al acceder a la propiedad, se accederá a la propiedad de la clase hija. Para acceder a la propiedad de la superclase, se utiliza:

```
super.variable;
```

- Una clase puede definir un método con el mismo nombre que un método de su superclase. En este caso, el método de la clase sobrescribe el método de la superclase. Para ejecutar el método de la superclase, se utiliza:

```
super.metodo();
```

**Nota:** Tanto en el caso de las propiedades como de los métodos con el mismo nombre que la superclase, no pueden tener menor acceso que en dicha superclase. Es decir, la propiedad o método de la clase hija debe tener el mismo o mayor acceso (por ejemplo, si la propiedad/método de la clase padre es `protected`, en la clase hija sólo puede ser `protected` o `public`).

#### Restricciones en la herencia

- Una subclase no puede sobrescribir los métodos declarados como `final`.
- Tampoco se pueden sobrescribir los métodos declarados como `static` (métodos de clase). Sin embargo, se puede ocultar el método de la superclase definiendo un método `static` con el mismo nombre en la subclase.
- Una subclase debe sobrescribir los métodos declarados como `abstract` en su superclase. Si no lo hace, debe declararse la clase como `abstract`.

#### Constructores en la herencia

- Todos los constructores de las subclases deben llamar a algún constructor de la clase padre. Si no se especifica explícitamente, Java llamará al constructor sin argumentos de la clase padre.
- Para invocar explícitamente el constructor de la clase padre dentro del constructor de la clase hija, se utiliza el método `super`:

```
super([argumentos]);
```

**Nota:** Esta instrucción debe ser la primera en el constructor. Por lo tanto, no se pueden usar `this()` y `super()` a la vez en el mismo constructor.

---

## Polimorfismo

Si una clase `B` deriva de otra `A`, es posible emplear objetos de `B` allí donde podríamos emplear objetos de `A`.

#### Ejemplo:

```
ClaseA b = new ClaseB(); // Construimos B como un objeto A (OK)
b.metodo1(); // OK
b.metodo3(); // ERROR, b no puede ejecutar métodos de B
((ClaseB) b).metodo3(); // OK. Usamos b como objeto de ClaseB
```

**Nota:**

- Si se utiliza un objeto de la clase `B` como un objeto de la clase `A`, sólo tendrá acceso directo a los métodos de la clase `A` (y no a los de su propia clase).
- Para acceder a los métodos de su clase, será necesario realizar una conversión explícita de tipo.

**Importante:** La conversión de tipo no funciona a la inversa, es decir, no se puede convertir un objeto de clase `A` a clase `B`.

**Nota adicional:** Aunque el tipo de datos define qué métodos se pueden usar, éstos siempre se ejecutan sobre el objeto. Es decir, un objeto de `ClaseB` al que se accede como un objeto de `ClaseA` sólo puede ejecutar los métodos definidos en `ClaseA`. Sin embargo, si se ejecuta un método sobrescrito, se ejecutará la versión de `ClaseB`.

## Conceptos importantes sobre métodos

### Sobrecarga de métodos

La **sobrecarga de métodos** significa que pueden existir diferentes definiciones del mismo método dentro de la misma clase. Para ello, los métodos deben diferir en el número o tipo de argumentos.

### Sobreescritura de métodos

- Al ejecutar un método, primero se busca en la clase en la que se pretende ejecutar. Si no está definido, se busca en las superclases.
- Esto permite heredar métodos de la clase padre sin necesidad de volverlos a definir.
- Para modificar el comportamiento de un método heredado, se puede sobrescribir en la subclase. Para ejecutar el método de la superclase, se utiliza:

```
super.metodo([args]);
```

## Clases y métodos abstractos

Cuando empleamos herencia, puede haber casos en los que sabemos que todas las clases hijas tendrán la misma funcionalidad (aunque con diferentes características). En este caso, podemos definir dicha funcionalidad en la clase padre como **abstracta**.

### Métodos abstractos

Un método abstracto se define con la palabra clave `abstract` y no tiene implementación en la clase padre:

```
[acceso] abstract retorno metodo([parámetros]);
```

### Clases abstractas

Una clase que contiene métodos abstractos debe ser declarada como `abstract`:

```
[acceso] abstract class Clase {
 ...
}
```

#### Notas:

- Una clase abstracta no puede ser instanciada (no se pueden crear objetos de la misma).
- Aunque una clase abstracta no puede ser instanciada, sí se pueden definir variables de su tipo y asignarles objetos de sus clases hijas (polimorfismo).

#### Ejemplo:

```
abstract class Empleado {
 abstract double calcularSueldo();
}

class EmpleadoFijo extends Empleado {
 double calcularSueldo() {
 return 2000.0;
 }
}

class EmpleadoPorHoras extends Empleado {
 double calcularSueldo() {
 return 15.0 * 40; // Ejemplo: 15€/hora por 40 horas
 }
}
```

## Clases y métodos finales

Una clase o método pueden ser etiquetados como `final`.

### Clases finales

Una clase declarada como `final` no puede ser extendida (no se pueden crear subclases):

```
final class Clase {
 ...
}
```

### Métodos finales

Un método declarado como `final` no puede ser sobrescrito en las subclases:

```
class Clase {
 final void metodo() {
 ...
 }
}
```

## 2.12. Actividad 12 - Herencia

### Instrucciones

[Github](#)

#### 1

En una empresa de alquiler de vehículos disponen de **coches** y **microbuses**. Se pide **diseñar una jerarquía de clases que permita gestionar el alquiler de estos vehículos**. A la hora de alquilar un **vehículo** se asumirá un **coste fijo por día de 40€**. Además, los **coches** pagarán un **coste adicional de 1,5 € / día y plaza**. Los **microbuses** son como coches pero tienen un **coste adicional diario que depende del microbús**.

Todos los vehículos dispondrán de los siguientes campos (en una clase Vehículo):

- `matricula`: El número de la matrícula (un String)
- `costeFijo`: Una constante entera con el coste fijo diario de cada vehículo. Dicho coste será de 40 € y se almacenará como public.

Para la clase `Vehículo` se definirán los siguientes métodos:

- un **constructor** con el número de `matricula`
- métodos `set/get` para el campo `matricula`

La clase `Coche` será **hija** de la clase `Vehículo`. Además de la `matricula` tendrá el siguiente campo:

- `plazas`: donde se indicará el **número de plazas** del vehículo.

Como métodos :

- un **constructor** con el número de `matricula` y el número de `plazas`. Este constructor **deberá llamar al constructor de la clase padre** (`super(matricula)`)
- métodos `set/get` para el número de `plazas`
- método `getCosteDia` que **retornará el coste diario del coche**. Será el coste fijo por vehículo más **1,5 € multiplicado** por el número de `plazas`.
- método `toString` que retornará la siguiente cadena

`matricula (plazas plazas)`

La clase `Microbus` es hija de la clase `Coche` y añadirá un nuevo campo:

- `costeTipo`: se almacenará el coste que **depende del tipo de autobús**.

Métodos :

- **constructor** que reciba todos los datos (`matricula, plazas y costeTipo`)
- métodos `set/get` para el `costeTipo`
- método `getCosteDia` que retornará el **coste diario** (coger el del `Coche`) más el `costeTipo`
- método `toString` que retornará la cadena

`matricula (plazas plazas) [Microbús]`

#### 2

Hacer un pequeño programa de prueba en el que :

- Se defina un coche y un microbús y se pidan sus datos.
- Se pregunte al usuario si quiere alquilar el coche o el microbús.
- En función de la respuesta se muestre el método `toString` del vehículo correspondiente y, en otra línea, su **coste diario**.

#### 3

En el ejemplo anterior, modificar la clase `Vehículo` para que disponga del método **abstracto** `getCosteDia`

OJO, que se tiene que llamar exactamente igual en todas las clases

Además añadir a la clase el siguiente campo y su método `is` (**NO queremos el método set**):

- `alquilado`: una variable **booleana** que indica si el vehículo está alquilado o no

Y los siguientes métodos :

- `alquilar`: este método retornará un valor double.
- Si el vehículo está alquilado (la variable `alquilado` es true) retornará un valor -1,
- Si no lo está, retornará el **coste diario** del vehículo y pondrá el campo `alquilado` a true
- `devolver`: este método **recibirá el número de días** que ha estado el coche alquilado y **retornará el coste total (coste diario por el número de días)**. Previamente pondrá el campo `alquilado` a false.

#### 4

Diseñar un programa que **defina un coche y un microbús** con sus datos. El programa **repetirá el siguiente proceso**:

- Se preguntará si se quiere **alquilar o devolver** (una A o una D)
- Tras ello se preguntará si se quiere **el coche o el microbús** (una C o una M)
- Si es el **coche** y se quiere **alquilar**
- Se llamará al método `alquilar` y se **guardará su resultado en una variable**:
  - Si es -1 se mostrará un **mensaje** al usuario diciendo que el coche **ya está alquilado**

- Si no, se mostrará un **mensaje** al usuario diciendo que alquila el coche e indicando el **coste diario** (lo que está guardado en la variable)
- Si es el **coche** y se quiere **devolver**
- Si el coche está **alquilado**
  - Se pedirá el **número de días** que ha estado el coche alquilado
  - Se llamará al método `devolver` con ese número de días y se guardará su resultado en una variable:
  - Se mostrará el **importe del alquiler** (el contenido de la variable anterior)
- Si es el **microbús** trabajaremos de manera similar.
- Tras acabar, se **preguntará** al usuario si desea alquilar/devolver **otro vehículo**

¿Sería posible hacer los dos procesos sin repetir las instrucciones?

### Adicional (especificar los métodos de modo que sean lo más genéricos posible)

- Definir un **método que permita comparar los costes diarios de dos vehículos cualesquiera**.
- Definir una **nueva clase Furgoneta** que tenga un campo `PMA` (peso máximo autorizado en kilos) y métodos `set/get` para el mismo, así como un **constructor** con todos los datos. El **coste diario** será el **coste fijo más 5€ por cada tonelada de PMA**.
- Tenemos que pagar un **impuesto sobre los alquileres de los vehículos** que es de un **12%** en el caso de los **coches** y de un **17%** en el caso de los **microbuses y furgonetas**. **Modificar** las clases de modo que podamos saber el **porcentaje de impuesto** y para que el **coste diario tenga en cuenta dicho impuesto**.
- Diseñar un pequeño programa que comprueba el funcionamiento de los nuevos métodos.

### 2.13. Interfaces

#### Interfaces en Java

Un interfaz especifica un protocolo de comportamiento que puede ser implementado por cualquier clase de la jerarquía de clases de la plataforma Java. En la práctica, es un conjunto de declaraciones de métodos (sin definición). También puede definir constantes, que son **implícitamente public, static y final**, y deben siempre inicializarse en la declaración. Todas las clases que implementan una determinada interfaz están obligadas a proporcionar una definición de los métodos de la misma.

Para declarar un interfaz, se emplea la siguiente sintaxis:

```
[public] interface Interfaz [extends Interf1[, Interf2 ...]] {
 // Lista de variables (final, public y static)
 // Declaración de métodos SIN IMPLEMENTAR
}
```

- Un interfaz puede ser `public` (implementado por cualquier clase); si no se indica nada, se asume `package` (sólo puede ser implementado por clases en el mismo paquete).
- Un interfaz puede derivar de otros interfaces, lo que obliga a la clase que lo implemente a definir todos los métodos del interfaz y de los interfaces de los que deriva.

En el cuerpo del interfaz, se pueden declarar propiedades y métodos. Las propiedades son **implícitamente final, static y public** y deben ser inicializadas **explicativamente**. Por su parte, los métodos son **implícitamente public y abstract**, por lo que hay que declararlos `public` a la hora de implementar el método en la clase que implementa el interfaz. De no hacerlo así, se generará un error en tiempo de compilación.

**Nota:** Es común emplear las interfaces para definir funcionalidades comunes que pueden tener varias clases. Por ello, es común que acaben en `able`, indicando que la clase es capaz de realizar dichas tareas (por ejemplo, `Cloneable`, `Serializable`, `Drawable`, etc.).

#### Implementación de interfaces

Una clase puede implementar una o varias interfaces. Para indicar que una clase implementa una o más interfaces, se ponen los nombres de las interfaces, separados por comas, detrás de la palabra `implements`. Por ejemplo:

```
public class Clase [extends SuperClase] implements Interfaz1[, Interfaz2...]
```

- Todas las clases que implementen el interfaz deben proporcionar una definición (aunque sea vacía) para todos los métodos declarados en el interfaz.

**Nota:** El nombre de una interfaz se puede utilizar como un nuevo tipo de referencia. En este sentido, el nombre de una interfaz puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la interfaz (polimorfismo).

## Múltiples Interfaces

Una clase en Java puede implementar múltiples interfaces, teniendo que definir los métodos de todos los interfaces que implemente. Esto puede causar problemas cuando no puedan coexistir los métodos de dichos interfaces en la misma clase.

Por ejemplo:

```
public interface Ifaz1 {
 void metodo1();
 int metodo2();
}

public interface Ifaz2 {
 int metodo1();
 int metodo3();
}
```

Si intentamos implementar ambos en la misma clase, tendremos un error, ya que `metodo1()` aparece en ambos interfaces pero de manera incompatible. Sin embargo, si los métodos son compatibles, la implementación será posible.

## Novedades en Java 8

### Métodos por Defecto

A partir de Java 8, los métodos de los interfaces pueden tener un cuerpo por defecto. Si no se implementa el método, se emplea el código definido en el cuerpo.

```
[public] default tipo metodo([args]) {
 cuerpo;
}
```

Por ejemplo:

```
public interface Emailable {
 default String de() {
 return "admin@zabalburu.org";
 }
 String a();
 String asunto();
 String texto();
}
```

## Métodos estáticos

Ahora se pueden definir métodos estáticos en los interfaces, pero es obligatorio asignarles un cuerpo:

```
static String getSMTPServer() {
 return "smtp.zabalburu.org";
}
```

**Nota:** Los métodos estáticos se pueden usar, pero no sobrescribir.

## Interfaces Funcionales y Expresiones Lambda

Un interfaz funcional es aquel que sólo tiene un método abstracto sin valor por defecto. Es decir, el interfaz puede tener múltiples métodos abstractos, pero todos excepto uno deberían tener un valor por defecto.

Los interfaces funcionales son importantes porque con ellos se pueden crear expresiones lambda. Las expresiones lambda proporcionan una manera compacta de crear un objeto de una clase que implemente un interfaz funcional.

**Ejemplo:**

```
public interface Operador {
 int operar(int uno, int dos);
}

public class Clase {
 public static void main(String[] args) {
 Operador suma = (a, b) -> a + b; // Retornamos el valor directamente
 Operador resta = (a, b) -> { // Segunda opción. Con un bloque de instrucciones
 return a - b;
 };

 System.out.println(suma.operar(12, 33)); // 45
 System.out.println(resta.operar(12, 33)); // -21
 }
}
```

## Interfaces Funcionales Predefinidos

Java proporciona una serie de interfaces funcionales predefinidos para los casos más habituales:

Interface	Parámetros	Resultado
Function<T,R>	1 entrada, 1 salida	Transforma un dato
BiFunction<T,U,R>	2 entradas, 1 salida	Combina dos datos
Consumer<T>	1 entrada, sin salida	Realiza una acción
BiConsumer<T,R>	2 entradas, sin salida	Realiza una acción
Predicate<T>	1 entrada, boolean	Valida un dato
BiPredicate<T,R>	2 entradas, boolean	Valida dos datos
Supplier<T>	Sin entradas, 1 salida	Proporciona un dato
UnaryOperator<T>	1 entrada, 1 salida	Modifica un dato
BinaryOperator<T>	2 entradas, 1 salida	Combina dos datos

**Ejemplo:**

```
Function<String, String> cursiva = (dato) -> "" + dato + "";
System.out.println(cursiva.apply("Hola a todo el mundo"));
```

**Salida:**

```
Hola a todo el mundo
```

### 2.14. Actividad 13 - Interfaces

## Actividad 13 - Interfaces

En una empresa quieren implementar un sistema de revisión para los diferentes documentos que se generan en la misma. Dado que los documentos pueden ser de múltiples tipos se ha decidido emplear interfaces para gestionar dicho proceso.

Definir las siguientes clases / interfaces;

### Interfaz Revisable

Debe proporcionar los siguientes métodos:

- String getRevisor()
- Date getFechaRevisión() : Debe tener un cuerpo por defecto que retorne la fecha de hoy
- int getValoracion()

En dicho interfaz, además se definirán **constantes enteras** para representar si el documento revisado ha sido **APROBADO (1)** o **RECHAZADO (2)**

### Clase Informe

Que implemente `Revisable` y que disponga de las siguientes **propiedades**

- String autor: El autor del informe
- Date fecha: La fecha de creación del informe
- String título: El título del informe
- String texto: El texto del informe
- String revisor: El revisor del informe
- int valoracion: La valoración de la revisión

Dicha clase, además, dispondrá de los siguientes **métodos**:

- Un constructor que reciba el `autor`, la `fecha`, el `título` y el `texto` del informe
- Métodos `set/get` para esos campos y los métodos `getRevisor` y `getValoracion` del interfaz que retornarán los valores de los campos correspondientes de la clase
- Un método `revisar` que recibirá el `revisor` y la `valoración` y los asignará a los campos correspondientes
- Un método `toString` que retorne el texto:

Informe `_título_ [_autor_]`. Fecha : `_fecha_`

### Clase Propuesta

Que implemente `Revisable` y que disponga de las siguientes **propiedades**

- String autor: El autor de la propuesta
- String título: El título de la propuesta
- String justificación: La justificación de la misma
- String revisor: El revisor de la propuesta
- Date fechaRevisión: La fecha de revisión de la propuesta
- int valoracion: La valoración de la misma

Dicha clase, además, dispondrá de los siguientes **métodos**:

- Un constructor que reciba el `autor`, el `título` y la `justificación` de la propuesta
- Métodos `set/get` para esos campos y los métodos `getRevisor`, `getFechaRevisión` y `getValoracion` del interfaz que retornarán los valores de los campos correspondientes de la clase
- Un método `revisar` que recibirá el `revisor`, la `fechaRevisión` y la `valoración` y los asignará a los campos correspondientes
- Un método `toString` que retorne el texto:

Propuesta `_título_ [_autor_]`.

### Clase Principal

Se definirá un método (recordad que debe ser **estático** para poderlo llamar desde el método `main`) `mostrarRevision` que recibirá un objeto `Revisable` y mostrará por pantalla:

- El resultado de llamar al método `toString` del objeto recibido
- El `texto`:

Revisor : `_revisor_`

Revisado el : `_fechaRevision_`

Valoración : `_APROBADO / RECHAZADO_`

Definir además un método `main` en el que:

- Se solicitará el `nombre` de la persona que va a revisar los documentos (va a ser una para todo el proceso)
- Se **repetirá el siguiente proceso** mientras el usuario quiera:
- Se **preguntará** si desea revisar un **informe** o una **propuesta** (I/P)
- En cualquiera de los casos se pedirá la información necesaria para crear el objeto y se creará el objeto con dicha información
- Se **pedirá** la información necesaria para llamar al método `revisar` del objeto. En el caso de la `valoración` el valor deberá ser un 1 o un 2.
- Se llamará al método `revisar` del objeto con la información obtenida

- Se llamará al método `mostrarRevisión` con dicho objeto
- Al final se mostrará cuántos documentos se han revisado y qué **porcentaje** ha sido **aprobado**

## 2.15. Testeando nuestra aplicación - TDD Test Driven Design

Cuando diseñamos una nueva clase es **importante comprobar que la funcionalidad de la misma es la adecuada** y para eso, la mejor opción pasa por diseñar un conjunto de **tests automatizados**. Aunque existen muchos frameworks para el diseño de Tests en Java, el más utilizado es `JUnit` que se integra perfectamente con los IDEs actuales. Un test en `JUnit` es simplemente **una clase que dispone de métodos anotados** con la anotación `@Test`. Dichos test pueden emplear una serie de comprobaciones (**aserciones**) que serán evaluadas para definir si se pasa o no el test. Las aserciones más empleadas son:

- `assertArrayEquals(arr1, arr2 [, mensaje])` : Comprueba si dos matrices contienen la misma información. Si no es así el test falla y se muestra el mensaje de error indicado
- `assertEquals(expr1, expr2 [,mensaje])` : Comprueba si dos expresiones son iguales empleando equals en el caso de objetos
- `assertFalse(expr1[, mensaje])` : Comprueba si la expresión es falsa
- `assertNotEquals(expr1, expr2 [, mensaje])` : Comprueba si dos expresiones son distintas
- `asssertNotNull(expr1[, mensaje])` : Comprueba que la expresión no sea nula
- `assertNotSame(expr1, expr2[, mensaje])` : Comprueba que las dos expresiones no retorne el mismo objeto (==)
- `asssertNull(expr1[, mensaje])` : Comprueba que la expresión sea nula
- `assertSame(expr1, expr2[, mensaje])` : Comprueba que las dos expresiones retornen el mismo objeto (==)
- `assertThrows(exc, función [, mensaje])` : Comprueba que la función lance la excepción indicada
- `assertTimeout(tiempo, función [, mensaje])` : Comprueba que la función se ejecute en el tiempo indicado
- `assertTrue(expr1[, mensaje])` : Comprueba si la expresión es cierta
- `fail(mensaje)` : Hace que el test falle con el mensaje indicado

Adicionalmente podemos etiquetar los métodos con alguna de las siguientes anotaciones:

- `@AfterAll`: El método se ejecuta después de pasar todos los tests
- `@AfterEach`: El método se ejecuta después de pasar cada uno de los tests
- `@BeforeAll`: El método se ejecuta antes de pasar todos los tests
- `@BeforeEach`: El método se ejecuta antes de pasar cada uno de los tests
- `@Disabled`: El test no debe pasarse
- `@DisplayName`: Permite especificar un nombre para el test
- `@Test`: Indica que el método es un test

NOTA: En NetBeans con art (el sistema de creación de aplicaciones que estamos empleando hasta ahora) no se puede usar JUnit 5. Para poder hacerlo hay que crear un proyecto Maven (emplearemos Maven para la creación de aplicaciones el próximo curso).

Veamos un ejemplo. Una vez creado un nuevo proyecto Maven vamos a diseñar una clase Producto con una serie de métodos que vamos luego a testear. La clase:

```
package org.zabalburu.tests;

public class Producto {
 private int prod_no;
 private String nombre;
 private int stock;
 private int stock_min;
 private double precioCoste;
 private double precioVenta;
 public Producto() {
 }
 public Producto(int prod_no, String nombre, int stock, int stock_min, double precioCoste, double precioVenta) {
 this.prod_no = prod_no;
 this.nombre = nombre;
 this.stock = stock;
 this.stock_min = stock_min;
 this.precioCoste = precioCoste;
 this.precioVenta = precioVenta;
 }
 public int getProd_no() {
 return prod_no;
 }

 public void setProd_no(int prod_no) {
 this.prod_no = prod_no;
 }

 ...
 @Override
 public boolean equals(Object obj) {
 if (this == obj) {
 return true;
 }
 }
}
```

```

 if (obj == null) {
 return false;
 }
 if (getClass() != obj.getClass()) {
 return false;
 }
 final Producto other = (Producto) obj;
 if (this.prod_no != other.prod_no) {
 return false;
 }
 return true;
 }

 @Override
 public String toString() {
 return "Producto(" + "prod_no=" + prod_no + ", nombre=" + nombre + ", stock=" + stock + ", stock_min=" + stock_min + ", precioCoste=" + precioCoste
 }

 public boolean bajoStock() {
 return this.stock < this.stock_min;
 }

 public double beneficio(){
 return this.precioVenta - this.precioCoste;
 }

 public double vender(int unidades){
 if (unidades > this.stock){
 throw new NoStockException(this.stock);
 }
 this.stock -= unidades;
 return unidades * this.precioVenta;
 }
}

```

Tenemos que crear la clase NoStockException :

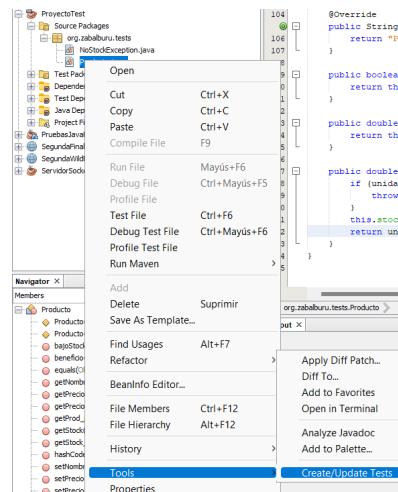
```

package org.zabalburu.tests;

public class NoStockException extends RuntimeException{
 public NoStockException(int stock){
 super("No hay suficiente stock en el almacén. Sólo quedan " + stock + " unidades.");
 }
}

```

NetBeans nos ayuda a la hora de crear los tests. Seleccionamos la opción Tools → Create / Update Tests del menú contextual de la clase que queremos testar:



y dejamos las opciones por defecto. Sólo vamos a testar algunos métodos. El código de la clase quedará de la siguiente forma:

```

package org.zabalburu.tests;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.DisplayName;

public class ProductoTest {

 @Test
 @DisplayName("Prueba del método equals")
 public void testEquals() {
 Producto producto = new Producto();
 Producto p2 = new Producto();
 assertEquals(producto, p2);
 producto.setProd_no(1);
 p2.setProd_no(2);
 assertNotEquals(producto, p2);
 p2.setProd_no(1);
 }
}

```

```

 assertEquals(producto, p2);
 }

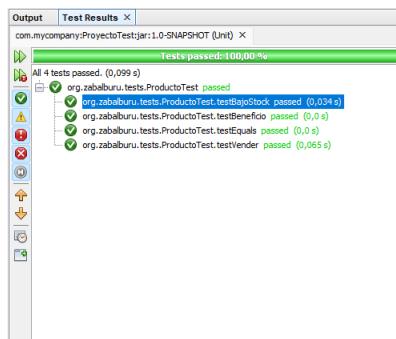
 @Test
 @DisplayName("Prueba del método bajoStock")
 public void testBajoStock() {
 Producto producto = new Producto();
 producto.setStock(10);
 producto.setStock_min(5);
 assertFalse(producto.bajoStock());
 producto.setStock(5);
 assertFalse(producto.bajoStock());
 producto.setStock(3);
 assertTrue(producto.bajoStock());
 }

 @Test
 @DisplayName("Prueba del método beneficio")
 public void testBeneficio() {
 Producto producto = new Producto();
 double pCoste = 120.35;
 double pVenta = 200.75;
 producto.setPrecioCoste(pCoste);
 producto.setPrecioVenta(pVenta);
 assertEquals(producto.beneficio(), pVenta - pCoste);
 }

 @Test
 @DisplayName("Prueba del método vender")
 public void testVender() {
 Producto producto = new Producto();
 int unidades = 5;
 int stock = 10;
 double pVenta = 200.75;
 producto.setPrecioVenta(pVenta);
 producto.setStock(stock);
 assertEquals(producto.vender(unidades), unidades * pVenta);
 assertEquals(producto.getStock(), stock - unidades);
 assertThrows(NoStockException.class, () -> producto.vender(100));
 }
}

```

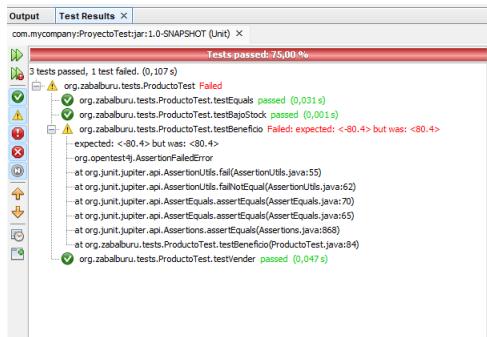
Ahora podemos ejecutar los tests con Run → Test file:



Si modificamos el código del método `beneficio()` simulando un error en la codificación:

```
public double beneficio(){
 return this.precioCoste - this.precioVenta;
}
```

Y volvemos a pasar los tests:



## Test Driven Design

Esta metodología propone **comenzar el diseño de una clase definiendo los tests (antes de crear la propia clase)**. Evidentemente todos los tests fallarán al principio (dado que ni siquiera tenemos la clase). Ahora el proceso es conseguir que pasen todos los tests. Una vez que pasan todos los tests se pasa a una nueva fase de refactorización en la que se trata de optimizar el código. Tras hacerlo se vuelve a comprobar que los tests siguen funcionando correctamente. Veamos un ejemplo. Creamos

(en el mismo proyecto) una clase de prueba para una clase Nota:

```
package org.zabalburu.tests;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

/**
 * @author ichue
 */
public class NotaTest {

 @Test
 public void testNotaATexto() {
 Nota n = new Nota(4);
 Assertions.assertEquals(n.getNotaTexto(),"Insuficiente");
 }

 @Test
 public void testNotaATextoCorto() {
 Nota n = new Nota(4);
 Assertions.assertEquals(n.getNotaTexto(),"IN");
 }

 @Test
 public void testAprobado() {
 Nota n = new Nota(4);
 Assertions.assertFalse(n.isAprobado());
 n.setNota(5);
 Assertions.assertTrue(n.isAprobado());
 }
}
```

Evidentemente el código ni compila. Creamos la clase Nota vacía y ahora podemos ir añadiendo los métodos (ayudándonos de NetBeans):

```
package org.zabalburu.tests;

/**
 * @author ichue
 */
public class Nota {

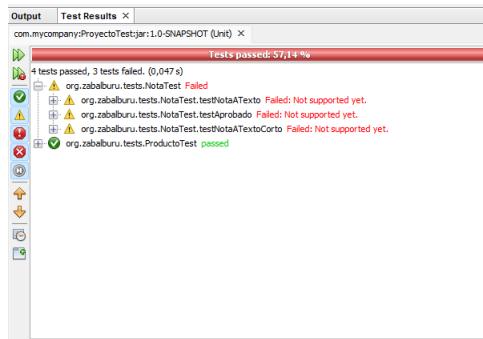
 Nota(int i) {
 throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.
 }

 Object getNotaTexto() {
 throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.
 }

 void setNota(int i) {
 throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.
 }

 boolean isAprobado() {
 throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.
 }
}
```

Ahora podemos compilar pero, evidentemente, los tests fallan:



Ahora intentamos añadir el código para solucionar el primer test:

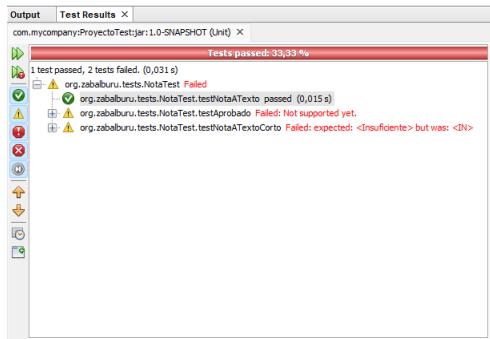
```
public String getNotaTexto() {
 switch (this.nota) {
 case 0:
 case 1:
 case 2:
 case 3:
 case 4:
 return "Insuficiente";
 case 5:
 return "Suficiente";
 }
}
```

```

 case 6:
 return "Bien";
 case 7:
 case 8:
 return "Notable";
 default:
 return "Sobresaliente";
 }
}

```

La salida ahora:



Hacemos lo mismo con el resto de los métodos:

```

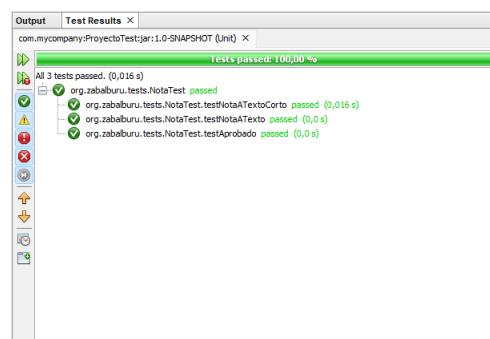
void setNota(int i) {
 this.nota = i;
}

boolean isAprobado() {
 if (this.nota < 5){
 return false;
 } else {
 return true;
 }
}

public String getNotaTextoCorto() {
 switch (this.nota){
 case 0:
 case 1:
 case 2:
 case 3:
 case 4:
 return "IN";
 case 5:
 return "SF";
 case 6:
 return "BN";
 case 7:
 case 8:
 return "NT";
 default:
 return "SB";
 }
}

```

Volvemos a pasar los tests:



Y ahora intentamos optimizar los métodos:

```

public String getNotaTexto() {
 if (this.nota < 5) {
 return "Insuficiente";
 } else if (this.nota == 5) {
 return "Suficiente";
 } else if (this.nota == 6) {
 return "Bien";
 } else if (this.nota < 9) {
 return "Notable";
 } else {
 return "Sobresaliente";
 }
}

```

```

 return "Notable";
 } else {
 return "Sobresaliente";
 }
}

public boolean isAprobado() {
 return this.nota >= 5;
}

public String getNotaTextoCorto() {
 if (this.nota < 5) {
 return "IN";
 } else if (this.nota == 5) {
 return "SE";
 } else if (this.nota == 6) {
 return "BN";
 } else if (this.nota < 9) {
 return "NT";
 } else {
 return "SB";
 }
}

```

Volvemos a pasar los tests para comprobar que los cambios hechos son correctos.

NOTA: En los tests se deberían comprobar todas las posibilidades de modo que sean lo más exhaustivos posible (algo que no hemos hecho en el ejemplo por ser repetitivo).

## 2.16. Actividad Adicional

### Instrucciones

Desarrollar una aplicación con información recopilada de Internet del consumo energético y coste aproximado debido al consumo en standby de una serie de equipos informáticos en una empresa. La aplicación solicitará el número de equipos de diferente tipo (ordenadores, monitores, tablets...) y mostrará el consumo y coste diario, mensual y anual. Todos los datos deberán aparecer correctamente formateados.

## 2.17. Github Classroom

### Guía de Usuario para GitHub Classroom

#### Introducción

GitHub Classroom es una plataforma que facilita la gestión y distribución de tareas y proyectos en un entorno educativo utilizando GitHub. Esta guía te ayudará a entender cómo utilizar GitHub Classroom para tus actividades académicas.

#### 1. Aceptar una Asignación (Assignment)

Tu profesor te proporcionará un enlace a una asignación en GitHub Classroom. Sigue estos pasos para aceptarla:

1. **Haz clic en el enlace de la asignación:** Este enlace te llevará a la página de la asignación en GitHub Classroom.
2. **Inicia sesión en GitHub:** Si aún no has iniciado sesión, se te pedirá que lo hagas con tu cuenta de GitHub.
3. **Acepta la asignación:** Busca un botón que diga "Accept the assignment" o similar y haz clic en él.
4. **Espera a que se cree tu repositorio:** GitHub Classroom creará un repositorio individual para ti, basado en una plantilla proporcionada por tu profesor. Este repositorio tendrá el formato `nombre-asignacion-tuusuari`.
5. **Accede a tu repositorio:** Una vez creado, verás un enlace a tu repositorio. Haz clic en él para acceder.

#### 2. Clonar el Repositorio

Para trabajar en el proyecto, necesitas clonar el repositorio a tu computadora local.

1. **Copia la URL del repositorio:** En la página principal de tu repositorio en GitHub, busca el botón verde que dice "Code". Haz clic en él y copia la URL HTTPS o SSH. Recomendamos usar HTTPS para principiantes.
2. **Abre tu terminal o Git Bash:** Navega hasta la carpeta donde deseas almacenar el proyecto.
3. **Ejecuta el comando `git clone <URL>`:** Reemplaza `<URL>` con la URL que copiaste en el paso 1. Por ejemplo:

```
bash git clone https://github.com/nombre-asignacion-tuusuari.git
```

1. **Ingresá tus credenciales de GitHub (Solo si usas HTTPS):**
2. **Nombre de usuario:** Se te pedirá tu nombre de usuario de GitHub. Ingrésalo y presiona Enter.
3. **Contraseña (o Token de Acceso Personal):**
  - **Contraseña:** Si la autenticación en dos pasos (2FA) no está activada en tu cuenta de GitHub, puedes ingresar tu contraseña directamente. Sin embargo, GitHub está eliminando gradualmente el uso de contraseñas.
  - **Token de Acceso Personal (Recomendado):** Si la autenticación en dos pasos está activada o prefieres una opción más segura, debes crear un Token de Acceso Personal (PAT) en GitHub. Puedes encontrar las instrucciones para crear un PAT en la [documentación de GitHub](#). Guarda tu PAT en un lugar seguro, ya que solo se mostrará una vez al crearlo. Cuando se te solicite la contraseña, pega el PAT en la terminal y presiona Enter.

**Nota:** Si estás usando SSH, necesitarás configurar una llave SSH en tu cuenta de GitHub y usarla para la autenticación. Esta opción es más avanzada y generalmente no es necesaria para las actividades de GitHub Classroom.

### 3. Trabajar en el Proyecto

Ahora tienes una copia local del repositorio y puedes empezar a trabajar en el proyecto.

1. **Realiza cambios en los archivos:** Abre los archivos del proyecto en tu editor de código favorito y realiza las modificaciones necesarias.

2. **Agrega los cambios al área de preparación (staging):** Usa el comando:

```
bash git add .
```

para agregar todos los archivos modificados, o:

```
bash git add <nombre-archivo>
```

para agregar archivos específicos.

1. **Confirma los cambios (commit):** Usa el comando:

```
bash git commit -m "Mensaje descriptivo de tus cambios"
```

para guardar tus cambios con un mensaje explicativo.

1. **Sube los cambios a GitHub (push):** Usa el comando:

```
bash git push
```

para subir tus cambios al repositorio remoto en GitHub. Si estás usando HTTPS, es posible que se te pidan tus credenciales nuevamente (nombre de usuario y PAT o contraseña, dependiendo de tu configuración).

---

### 4. Verificación Automática (Autograding) (Si Aplica)

Tu profesor puede haber configurado pruebas automáticas (autograding) para la asignación. Estas pruebas se ejecutarán cada vez que hagas un push a tu repositorio.

1. **Verifica el estado de las pruebas:** En la pestaña "Actions" de tu repositorio en GitHub, puedes ver el estado de las pruebas.

2. **Revisa los resultados de las pruebas:** Si las pruebas fallan, revisa los registros (logs) para identificar los errores y corregir tu código.

---

### 5. Entregar la Asignación

La entrega de la asignación generalmente se realiza simplemente haciendo push de tus cambios finales antes de la fecha límite. Tu profesor te indicará si hay algún paso adicional.

---

### 6. Obtener Ayuda

Si tienes problemas o preguntas, puedes:

- Consultar la documentación de GitHub: <https://docs.github.com/>
  - Buscar ayuda en línea: Stack Overflow y otros foros pueden ser útiles para resolver problemas comunes.
  - Contactar a tu profesor: Si tienes preguntas específicas sobre la asignación, contacta a tu profesor.
- 

#### Recuerda:

- Confirma tus cambios con frecuencia.
- Escribe mensajes de commit descriptivos.
- Mantén tu repositorio sincronizado con el remoto.
- Usa Tokens de Acceso Personal (PAT) en lugar de contraseñas para mayor seguridad.

### 3. Matrices y Colecciones en JAVA

#### 3.1. Introducción a las Matrices

Una matriz es una estructura que almacena elementos del mismo tipo. El tamaño (longitud) de una matriz se define en tiempo de ejecución cuando se crea la matriz y no puede ser modificada. Para almacenar datos de diferentes tipos o para estructuras de datos que precisen cambiar su tamaño se deben emplear clases derivadas de la clase `Collection` (algunas de estas clases se verán posteriormente).

##### Declaración, creación y uso básico de matrices

Como ya se ha indicado, una matriz es un conjunto de elementos del mismo tipo almacenados consecutivamente en memoria. Toda matriz tiene un nombre, y a los elementos de la matriz se accede a través de un índice que indica la posición relativa del elemento dentro de la matriz donde la primera posición es la cero. Por ejemplo, una matriz unidimensional de cinco nombres podría representarse así:

matriz	"Luis"	"Juan"	"Ana"	"María"	"Julio"
índice	0	1	2	3	4

Realmente la matriz no almacenaría directamente los nombres, dado que son objetos de tipo String, sino que guardaría las direcciones de memoria de los mismos. Pero por simplicidad, lo dejaremos así.

Los pasos para crear y trabajar con una matriz son similares a los pasos para crear un objeto :

- declarar la matriz : esto implica especificar el tipo de datos que va a contener (tipos básicos, objetos, e incluso otras matrices)

```
int[] numeros; // también vale int numeros[];
String[] nombres;
Empleado[] empleados;
```

- crear la matriz : aquí es donde especificamos el número de elementos que componen la matriz (el tamaño de la misma). Para especificar el número de elementos de la matriz se emplea el operador new

```
numeros = new int[10]; // Espacio para 10 números
nombres = new String[5]; // Espacio para 5 cadenas
empleados = new Empleado[20]; // Espacio para 20 empleados
// También se pueden hacer las dos cosas a la vez
Alumno alum[] = new Alumno[20]; // Espacio para 20 alumnos
```

- inicializar los elementos de la matriz: Con los pasos anteriores simplemente hemos especificado los datos que vamos a almacenar en la matriz, pero no hemos indicado que datos son. Para inicializar un elemento de la matriz, deberemos hacerlo a partir de su posición (índice) en la matriz. Dado que el primer elemento de la matriz está en la posición 0, el último índice válido será el número de elementos de la matriz menos 1.

```
numeros[0] = 20; // Almacena 20 como el primer número
nombres[4] = "Julio"; // Guarda Julio en la última posición de la matriz
empleado[0] = new Empleado("Julio","Martin"); // Se asigna un objeto empleado a la primera posición de empleados
```

Es posible (si conocemos los datos a priori) realizar estos tres procesos en un único paso mediante la **inicialización expresa de los elementos de la matriz**:

```
int[] numeros = { 10, 20, 30 }; // La matriz se declara con 3 elementos
String nombres[] = {"Luis", "Juan", "Ana", "Maria", "Julio"}; // 5 nombres
Empleado empleados[] = {
 new Empleado("Julio","Martin"),
 new Empleado("Ana","Marcos"),
 new Empleado("Pedro","Juárez"),
 new Empleado("Eva","Lesmas")
}; // 4 empleados
```

También es posible crear una matriz con valores sobre la marcha para utilizarla en cualquier sitio:

```
int[] matriz;
...
matriz = new int[]{4,5,7}; // Asigna una nueva matriz de tres elementos
ordenar(new String[]{"Juan","Ana","Luis"}); // Pasa una nueva matriz como parámetro a la función
```

#### Acceder a los elementos

Una vez creada e inicializada la matriz, podemos acceder a sus elementos mediante su índice correspondiente. Los elementos de la matriz pueden emplearse en cualquier sitio y pueden realizar cualquier operación que sea válida con los tipos de datos que representan. Para acceder a un elemento de una matriz hay que indicar el nombre de la matriz y, entre corchetes, la posición del elemento que queremos recuperar comenzando a contar desde 0, lo que implica que los valores posibles para el índice serán entre 0 y la longitud de la matriz menos 1. Por ejemplo, las siguientes instrucciones serían válidas con las matrices declaradas previamente:

```
numeros[0] = numeros[1] * numeros[2]; // Almacena 600 (20*30) en la posición 0
System.out.println(nombres[2].toUpperCase()); // Muestra ANA
empleados[3].setSueldo(1200); // Invoca el método setSueldo de la clase Empleado sobre Eva para modificar su sueldo
```

Es posible conocer el tamaño de una matriz, a través de la propiedad `length` de la matriz, por lo que podemos emplearlo para recorrer una matriz. El siguiente ejemplo mostraría todos los elementos de una matriz `numeros`:

```
int[] numeros = { 1,2,3,4,5,6,7,8,9,0 }, i;
for (int i = 0; i < numeros.length; i++) {
 System.out.println("numeros[" + i + "] = " + numeros[i]);
}
```

La salida

```
numeros[0] = 1
numeros[1] = 2
...
```

```
numeros[9] = 0
```

En caso de intentar acceder a un elemento fuera de los límites de la matriz se producirá una excepción `ArrayIndexOutOfBoundsException`.

```
public static void main(String[] args) {
 int[] numeros = { 1,2,3,4,5,6,7,8,9,0 };
 numeros[10] = 10;
}
```

La salida

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at matrices.app.Matrices.main(Matrices.java:19)
```

Como se puede ver, la salida nos indica el error, el índice que lo ha generado (10) y la línea del programa en la que se ha producido el error (línea 19, método main de la clase `matrices.app.Matrices` que está en el fichero `Matrices.java`).

### 3.2. Operaciones Básicas con Matrices Desordenadas Inserción eliminación búsqueda y listado

#### Inserción de elementos

A la hora de insertar elementos en una matriz desordenada, la forma más simple es insertarlos al final de la misma. Dado que la matriz no estará llena (si no, no podríamos añadir más elementos) necesitamos de un contador que indique cuántos elementos hay actualmente insertados en la matriz.

El proceso es muy sencillo:

- situar el elemento en la posición indicada por el contador de elementos (que será la primera posición libre en la matriz). Es decir, si actualmente la matriz tiene 5 elementos (`contador`), ocuparán las posiciones 0 a 4 por lo que la primera posición libre es la 5 (`contador`). Así que almacenaremos el nuevo elemento en la posición que nos indica el contador
- incrementar el contador en uno

Veamos un ejemplo (en este ejemplo, la inserción se hace en un método aparte):

```
public class InsercionAlFinal {
 static int contNombres=0;
 static String nombres[] = new String[10];

 public static void main(String args[]) {
 insertar("Juan");
 insertar("Ana");
 insertar("Eva");
 insertar("Luis");
 for (int i=0;i<contNombres;i++) {
 System.out.println(nombres[i]);
 }
 }

 public static boolean insertar(String elemento) {
 if (contNombres == nombres.length) {
 return false;
 } else {
 nombres[contNombres] =elemento;
 contNombres++;
 return true;
 }
 }
}
```

La salida sería

```
Juan
Ana
Eva
Luis
```

Evidentemente, los elementos en la matriz estarán en el orden en el que se han insertado

#### Búsqueda de un elemento

Cuando queremos localizar un elemento en una matriz desordenada, el proceso se basa en recorrer la matriz mientras no lleguemos al final (si la matriz está llena será lo que indique `length`, si no lo está, lo que indique el `contador`) y no encontramos lo que estamos buscando. Si al salir de la repetitiva estamos fuera de la matriz quiere decir que el elemento no existe. Si estamos dentro, la posición en la que nos hemos quedado es donde está el elemento buscado.

```
public class InsercionAlFinal {
 static int contNombres=0;
 static String nombres[] = new String[10];

 public static void main(String args[]) {
 ...
 System.out.println("Eva : " + buscar("Eva"));
 System.out.println("Carlos : " + buscar("Eva"));
 }

 public static boolean insertar(String elemento) {
 ...
 }

 public static int buscar(String nombre) {
 int pos;
 for (pos=0; pos<contNombres &&
```

```

 ! elemento.equalsIgnoreCase(nombres[pos]); pos++);
 if (pos<contNombres) {
 return pos;
 } else {
 return -1;
 }
}
}

```

La salida sería

```

...
Eva : 2
Carlos : -1

```

Es importante recalcar que, en la repetitiva de búsqueda, **después** del `for` sólo se pone un ; (instrucción vacía) dado que **no necesitamos repetir nada** (lo único que hay que hacer es sumar uno a pos y eso ya lo hacemos en el `for`).

### Eliminación de un elemento

La **eliminación** de un elemento normalmente pasa por dos fases:

- Encontrar el elemento a eliminar
- Eliminarlo

La **primera fase**, dado que la matriz está desordenada, la haremos con una **búsqueda desordenada**.

Para la **segunda fase** lo que hay que hacer es **poner en la posición del elemento a eliminar el siguiente, en la del siguiente el que está detrás de él y así sucesivamente hasta el anteúltimo** (posición a la que moveremos el último).

Una vez hecho esto, basta con **restar uno al contador** de elementos:

```

public class InsercionAlFinal {
 static int contNombres=0;
 static String nombres[] = new String[10];

 public static void main(String args[]) {
 ...
 for (int i=0;i<contNombres;i++) {
 System.out.println(nombres[i]);
 }
 ...
 eliminar("Luis");
 System.out.println("-----");
 for (int i=0;i<contNombres;i++) {
 System.out.println(nombres[i]);
 }
 }

 public static boolean insertar(String nombre) {
 ...
 }

 public static int buscar(String nombre) {
 ...
 }

 public static boolean eliminar(String nombre) {
 int pos = buscar(nombre);
 if (pos == -1) {
 return false;
 }
 for (int i=pos; i < contNombres - 1; i++) {
 nombres[i] = nombres[i+1];
 }
 contNombres--;
 return true;
 }
}

```

La salida sería

```

Ana
Eva
Juan
Luis
...

Ana
Eva
Luis

```

### Buscar elementos que cumplan una condición

En este caso, lo que queremos es **buscar todos los elementos de la colección que cumplan una condición** dada. Dado que la matriz no está ordenada, no nos queda más remedio que \*recorrer uno a uno todos los elementos y comprobar si cumplen o no la condición. Es posible que no haya ningún elemento que cumpla dicha condición así que, habitualmente, se intenta encontrar el primer elemento que cumpla la condición y, si se encuentra, entonces se procesa el resto de los elementos comprobando si cumplen o no. Para el siguiente ejemplo vamos a trabajar con una clase `Correo` que se usa para enviar mensajes de correo entre usuarios:

```
public class Correo {
```

```

private String de;
private String a;
private String asunto;
private String texto;

public Correo(){
}

public Correo(String de , String a , String asunto, String texto){
 this.de = de;
 this.a = a;
 this.asunto = asunto;
 this.texto = texto;
}

public String getDe(){
 return this.de;
}

public void setDe(String de){
 this.de = de;
}

...

public String toString(){
 return "De : " + this.de +
 "\nA : " + this.a +
 "\nAsunto : " + this.asunto +
 "\nMensaje : " + this.texto;
}
}

```

Vamos a crear un programa que busque todos los correos de un determinado usuario

```

public class Mensajes {
 private static Correo[] correos = {
 new Correo("juan","ana","Hola","Hola Ana. Tenemos que reunirnos"),
 new Correo("ana","juan","Re: Hola","Hola Juan. Yo puedo el jueves a las 18:00"),
 new Correo("juan","carlos","Reunión","Estoy intentando quedar con Ana. Te aviso cuando sepa algo"),
 new Correo("carlos","juan","Re: Reunión","Perfecto. ya me dirás"),
 new Correo("juan","ana","Re: Hola","Me viene Genial!. Aviso a carlos para que también esté"),
 new Correo("ana","juan","Re: Hola",";"),
 new Correo("juan","carlos","Re: Reunión","Hecho!. El jueves a las 6.")
 };

 public static void main(String[] args) {
 String nombre = JOptionPane.showInputDialog("Su nombre");
 int pos;
 for (pos = 0; pos < correos.length &&
 !nombre.equalsIgnoreCase(correos[pos].getA()));
 pos++);
 if (pos == correos.length){
 System.out.println("No tiene ningún correo");
 } else {
 System.out.println("Correos recibidos de " + nombre);
 int cont = 0;
 for(; pos<correos.length; pos++){
 if (nombre.equalsIgnoreCase(correos[pos].getA())){
 System.out.println("-----");
 System.out.println(correos[pos]);
 cont++;
 }
 }
 System.out.println("-----");
 System.out.println("Tiene " + cont + " correos");
 }
 }
}

```

La salida si introducimos Juan:

```

Correos recibidos de Juan

De : ana
A : juan
Asunto : Re: Hola
Mensaje : Hola Juan. Yo puedo el jueves a las 18:00

De : carlos
A : juan
Asunto : Re: Reunión
Mensaje : Perfecto. ya me dirás

De : ana
A : juan
Asunto : Re: Hola
Mensaje : ;)

Tiene 3 correos

```

La salida si introducimos Miguel:

## Patrón DAO (Objetos de Acceso a Datos)

El patrón de diseño DAO permite abstraer la aplicación de los datos a los que accede simplificando el cambio de los orígenes de datos. Es útil, por ejemplo, cuando una misma aplicación tiene que trabajar con un origen de datos u otro en función de alguna condición. Por ejemplo, cuando la aplicación está en producción (funcionando en la empresa) puede conectarse a una BBDD Oracle y cuando está en desarrollo (cuando se le añaden nuevas funcionalidades) conectarse a una BBDD más simple o incluso trabajar en memoria. El patrón DAO proporciona un mecanismo sencillo para cambiar de un entorno a otro sin necesidad de modificar la aplicación propiamente dicha. El patrón DAO incluye los siguientes elementos:

- **El dominio**: Son las clases en las que se va a almacenar la información (el **modelo**)
- **El interfaz de negocio**: Es un interfaz que expone todos los métodos de acceso a datos para el dominio (habitualmente consultas, modificaciones, inserciones y eliminaciones)
- **La implementación**: Son las clases que implementan el interfaz de negocio contra un origen de datos específico. Es donde se aparece el código de acceso a la información

Habitualmente se crean múltiples interfaces de negocio y su implementación en función de las diferentes clases del dominio. Es habitual añadir un **componente adicional** para simplificar el manejo de dichos elementos:

- La clase de **servicio**: Es la clase con la que se comunica la aplicación. Incluye los métodos necesarios de las **implementaciones** de negocio a las que, eventualmente, se pueden añadir reglas de negocio.

Veamos un ejemplo con una clase de dominio de `Producto` y empleando matrices como almacenamiento:

#### Modelo (Clase Producto)

```
package org.zabalburu.patrondao.modelo;

/**
 *
 * @author ichue
 */
public class Producto {
 private static int contIds = 1;

 private int id;
 private String nombre;
 private int stock;
 private int stockMinimo;
 private double precio;

 public Producto() {
 this.id = Producto.contIds;
 Producto.contIds++;
 }

 public int getStockMinimo() {
 return stockMinimo;
 }

 public void setStockMinimo(int stockMinimo) {
 this.stockMinimo = stockMinimo;
 }

 public Producto(String nombre, int stock, int stockMinimo, double precio) {
 this();
 this.nombre = nombre;
 this.stock = stock;
 this.stockMinimo = stockMinimo;
 this.precio = precio;
 }

 public int getId() {
 return id;
 }

 public void setId(int id) {
 this.id = id;
 }

 public String getNombre() {
 return nombre;
 }

 public void setNombre(String nombre) {
 this.nombre = nombre;
 }

 public int getStock() {
 return stock;
 }

 public void setStock(int stock) {
 this.stock = stock;
 }

 public double getPrecio() {
 return precio;
 }
}
```

```

 }

 public void setPrecio(double precio) {
 this.precio = precio;
 }

 public boolean bajoStock(){
 return this.stock < this.stockMinimo;
 }

 @Override
 public int hashCode() {
 int hash = 7;
 hash = 61 * hash + this.id;
 return hash;
 }

 @Override
 public boolean equals(Object obj) {
 if (this == obj) {
 return true;
 }
 if (obj == null) {
 return false;
 }
 if (getClass() != obj.getClass()) {
 return false;
 }
 final Producto other = (Producto) obj;
 return this.id == other.id;
 }

 @Override
 public String toString() {
 return "Producto{" + "id=" + id + ", nombre=" + nombre + ", stock=" + stock + ", stockMinimo=" + stockMinimo + ", precio=" + precio + '}';
 }
}

```

#### **Interfaz (ProductoDAO)**

```

import org.zabalburu.patrondao.modelo.Producto;

/**
 *
 * @author ichue
 */
public interface ProductoDAO {
 Producto nuevoProducto(Producto p);
 void eliminarProducto(int id);
 Producto getProducto(int id);
 Producto[] getProductos();
 Producto[] getProductosBajoStock();
}

```

#### **Implementacion (ProductoMatriz)**

```

package org.zabalburu.patrondao.dao;

import org.zabalburu.patrondao.modelo.Producto;

/**
 *
 * @author ichue
 */
public class ProductoMatriz implements ProductoDAO {

 private static Producto[] productos = new Producto[100];
 private static int contProductos = 0;

 @Override
 public Producto nuevoProducto(Producto p) {
 if (contProductos < productos.length){
 productos[contProductos] = p;
 contProductos++;
 }
 return p;
 }

 @Override
 public void eliminarProducto(int id) {
 int pos = getProductoPos(id);
 if (pos != -1){
 for(;pos < contProductos - 1; pos++){
 productos[pos] = productos[pos + 1];
 }
 contProductos--;
 }
 }

 @Override
 public Producto getProducto(int id) {
 int pos = getProductoPos(id);
 if (pos != -1){

```

```

 return productos[pos];
 }
 return null;
}

@Override
public int getProductoPos(int id) {
 int i;
 for(i=0; i<contProductos && id != productos[i].getId(); i++);
 if (i < contProductos){
 return i;
 } else {
 return -1;
 }
}

@Override
public Producto[] getProductos() {
 // Devolvemos una matriz llena
 Producto[] temp = new Producto[contProductos];
 for (int i = 0; i < contProductos; i++) {
 temp[i] = productos[i];
 }
 return temp;
}

@Override
public Producto[] getProductosBajoStock() {
 Producto[] encontrados = new Producto[productos.length];
 int num = 0;
 for(int i=0; i< contProductos; i++){
 if (productos[i].bajoStock()){
 encontrados[num] = productos[i];
 num++;
 }
 }
 Producto[] temp = new Producto[num];
 for(int i=0; i<num; i++){
 temp[i] = productos[i];
 }
 return temp;
}
}

```

#### **Clase servicio (opcional)**

```

package org.zabalburu.patrondao.servicio;

import org.zabalburu.patrondao.dao.ProductoDAO;
import org.zabalburu.patrondao.dao.ProductoMatriz;
import org.zabalburu.patrondao.modelo.Producto;

/**
 *
 * @author ichue
 */
public class ProductoServicio {
 private static ProductoDAO dao = new ProductoMatriz();

 public Producto nuevoProducto(Producto p){
 return dao.nuevoProducto(p);
 }

 public void eliminarProducto(int id){
 dao.eliminarProducto(id);
 }

 public Producto getProducto(int id){
 return dao.getProducto(id);
 }

 public int getProductoPos(int id){
 return dao.getProductoPos(id);
 }

 public Producto[] getProductos(){
 return dao.getProductos();
 }

 public Producto[] getProductosBajoStock(){
 return dao.getProductosBajoStock();
 }
}

```

#### **Aplicación Principal**

```

package org.zabalburu.patrondao;

import org.zabalburu.patrondao.modelo.Producto;
import org.zabalburu.patrondao.servicio.ProductoServicio;

```

```

/**
 *
 * @author ichue
 */
public class PatronDAO {
 private static ProductoServicio servicio = new ProductoServicio();

 public static void main(String[] args) {
 servicio.nuevoProducto(new Producto("Producto Uno", 10, 20, 75.23));
 servicio.nuevoProducto(new Producto("Producto Dos", 24, 10, 125.3));
 servicio.nuevoProducto(new Producto("Producto Tres", 15, 15, 97.75));
 servicio.nuevoProducto(new Producto("Producto Cuatro", 10, 15, 42.50));
 System.out.println("---- Tras Insertar ----");
 for(Producto p : servicio.getProductos()){
 System.out.println(p);
 }
 System.out.println("");

 System.out.println("---- Búsqueda ----");
 System.out.println("Posición Empleado id 3 : " + servicio.getProductoPos(3));
 System.out.println("Posición Empleado id 30 : " + servicio.getProductoPos(30));
 System.out.println("");
 System.out.println("Empleado id 3 : " + servicio.getProducto(3));
 System.out.println("Empleado id 30 : " + servicio.getProducto(30));
 System.out.println("");

 servicio.eliminarProducto(3);
 servicio.eliminarProducto(30); // NO Existe
 System.out.println("---- Tras Eliminar ----");
 for(Producto p : servicio.getProductos()){
 System.out.println(p);
 }
 System.out.println("");
 }
}

```

#### La salida

```

---- Tras Insertar ----
Producto{id=1, nombre=Producto Uno, stock=10, stockMinimo=20, precio=75.23}
Producto{id=2, nombre=Producto Dos, stock=24, stockMinimo=10, precio=125.3}
Producto{id=3, nombre=Producto Tres, stock=15, stockMinimo=15, precio=97.75}
Producto{id=4, nombre=Producto Cuatro, stock=10, stockMinimo=15, precio=42.5}

---- Búsqueda ----
Posición Empleado id 3 : 2
Posición Empleado id 30 : -1

Empleado id 3 : Producto{id=3, nombre=Producto Tres, stock=15, stockMinimo=15, precio=97.75}
Empleado id 30 : null

---- Tras Eliminar ----
Producto{id=1, nombre=Producto Uno, stock=10, stockMinimo=20, precio=75.23}
Producto{id=2, nombre=Producto Dos, stock=24, stockMinimo=10, precio=125.3}
Producto{id=4, nombre=Producto Cuatro, stock=10, stockMinimo=15, precio=42.5}

---- Productos Bajo Stock ----
Producto{id=1, nombre=Producto Uno, stock=10, stockMinimo=20, precio=75.23}
Producto{id=2, nombre=Producto Dos, stock=24, stockMinimo=10, precio=125.3}

```

#### 3.3. Actividad 14 - Matrices desordenadas

#### Instrucciones

[Enlace a GitHub](#)

[Actividad Adicional](#) [Actividad Adicional II](#) En una librería disponen de los siguientes libros en venta :

Título	Tema	Precio	Unidades
Aprenda C Ya	Programación	75,12	5
Microsoft Office	Ofimática	58,6	12
Windows 10	Sistemas Operativos	45	8
C Avanzado	Programación	90	3
Word Básico	Ofimática	64,6	10
Windows 2015 Server	Sistemas Operativos	52,3	7
Access 2015	Ofimática	32,45	5
Diseño de Algoritmos	Programación	90,15	10

Título	Tema	Precio	Unidades
Excel 2015	Ofimática	52,58	4

Se desea **realizar un programa que gestione las ventas de los diferentes libros de la librería**. Para ello se deberá diseñar una clase, denominada `Libro` que permita gestionar los datos de un libro. La clase deberá incluir:

- Las **propiedades** para cada uno de los datos de cada libro (título, tema, precio y unidades)
- Una **propiedad adicional**, denominada `ventas` que almacenará las unidades vendidas del libro y que, **en principio, estará a 0**.

Y los métodos:

- Un **constructor** que reciba el `título` y el `tema` del libro y otro que, **además**, incluya el `precio` y las `unidades`
- Métodos `get / set` para las **propiedades** de la clase (**excepto** para `ventas` que sólo dispondrá de método `get`).
- Un método, denominado `vender` que reciba el número de `libros` a vender y **retorne** el `importe` correspondiente a la venta. Este método **deberá descontar el número de libros vendidos de las unidades existentes**. Además se **acumularán** el número de `libros` vendidos al campo `ventas`. Si no hay suficientes unidades, se **retornará un 0 y no se hará el proceso anterior**.

Se deberán **definir**, además, el **DAO** (donde se creará la matriz de libros) y el **Servicio** con los siguientes métodos:

- `Libro buscarLibro(String título)`: Buscará de manera **desordenada** el libro con ese `título`
- `Libro[] getLibros()`: Retornará la matriz de libros
- `Libro[] getLibrosTema(String tema)`: Este método realizará las siguientes tareas:
- **Creará** una matriz de libros **vacía** (`librosTema`) con tantas posiciones como libros hay
- **Recorrerá toda la matriz de libros** y si el libro **es del tema** lo **añadirá** a la matriz `librosTema`
- Retornará la matriz

Tras definir la clase, se creará una **clase ejecutable** denominada `Librería` que, en el método `main`, definirá crear una propiedad estática para almacenar el servicio. Tras ello, el programa mostrará el siguiente menú al usuario mediante un `JOptionPane`:

1. Vender Libro

2. Buscar Tema

3. Salir

Opción [1-3]

### Opción 1

- Se pedirá el `título` del libro que se desea vender y se **buscará el libro en la matriz de libros**:
- Si el libro **no se encuentra** se dará un mensaje de **error**
- Si el libro **se encuentra** se mostrará su `tema, precio, las unidades disponibles` y se preguntará qué cantidad de libros se desea **comprar**:
  - Si la cantidad es **mayor** que las unidades existentes se dará un mensaje de **error**
  - Si la cantidad es **menor o igual** se ejecutará el método `vender` pasándole la cantidad a vender y se mostrará el `importe` (lo que devuelve el método) por pantalla
- Se **preguntará** al usuario si se desean **vender más** libros.
- Si responde **negativamente** se volverá al menú principal. En caso contrario se **repetirá** el proceso

### Opción 2

- Se preguntará el `tema` a buscar y se mostrará el siguiente listado:

Tema : nombre

Título Precio Unidades  
Título Precio Unidades  
....

Se han encontrado n libros del tema.

Si no hay libros del tema se dará el error correspondiente

- Al finalizar el proceso se **preguntará si se quiere seguir buscando libros de otros temas**.
- Si responde negativamente se volverá al menú principal. En caso contrario se repetirá el proceso

### Opción 3

Al salir se mostrará el siguiente resumen

Resumen de ventas

Título	Tema	Precio	Unidades	Ventas	Importe
Título	Tema	Precio	Unidades	Ventas	Importe
...					
		Suma	Suma	Suma	

Todos los **imports** deberán aparecer correctamente formateados con formato monetario.

### 3.4. Actividad 15 - Operaciones con matrices desordenadas

<https://classroom.github.com/a/2CpDVb-V> Partir de una copia del ejercicio anterior. En este caso, la matriz se deberá definir para almacenar hasta 50 libros diferentes y se creará el contador correspondiente (dado que, ahora, la matriz no va a estar llena). Los libros de la tabla se añadirán en el constructor y se actualizará el contador con el valor adecuado. Si ejecutamos ahora la aplicación, fallará. ¿Por qué? Corregir el programa para que vuelva a funcionar. Añadir al menú principal las siguientes opciones (y cambiar Salir a la opción 6): 3. Añadir un Nuevo Libro 4. Incrementar las Unidades de un Libro 5. Eliminar un Libro 6. Salir

#### Opción 3

- Se pedirá el título del libro a añadir y se buscará en la matriz de manera desordenada
- Si se encuentra se dará un mensaje de error indicando que el libro ya existe
- Si no se encuentra

Se pedirán el resto de los datos del libro (tema, precio y unidades). NOTA: El precio se pedirá en castellano Con todos los datos se creará un objeto Libro y se añadirá al final de la matriz (actualizando el contador)

- Se preguntará al usuario a ver si quiere añadir un nuevo libro y se actuará en consecuencia

#### Opción 4

- Se pedirá el título del libro a modificar y se buscará en la matriz de manera desordenada
- Si no se encuentra se dará un mensaje de error indicando que el libro no existe
- Si se encuentra

Se mostrarán sus datos y se pedirán las unidades recibidas que se acumularán sobre las unidades disponibles actualmente del libro

- Se preguntará al usuario si quiere repetir el proceso y se actuará en consecuencia

#### Opción 5

- Se pedirá el título del libro a eliminar y se buscará en la matriz de manera desordenada
- Si no se encuentra se dará un mensaje de error indicando que el libro no existe
- Si se encuentra

Se mostrarán sus datos y se pedirá confirmación para eliminar el libro Si se confirma

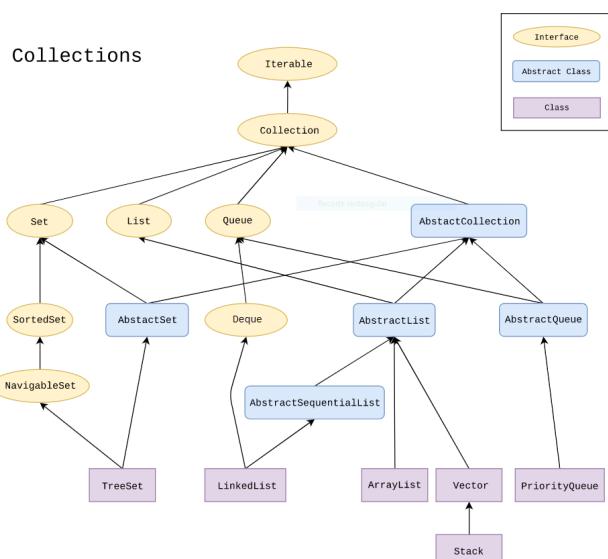
Se eliminará el libro de la matriz

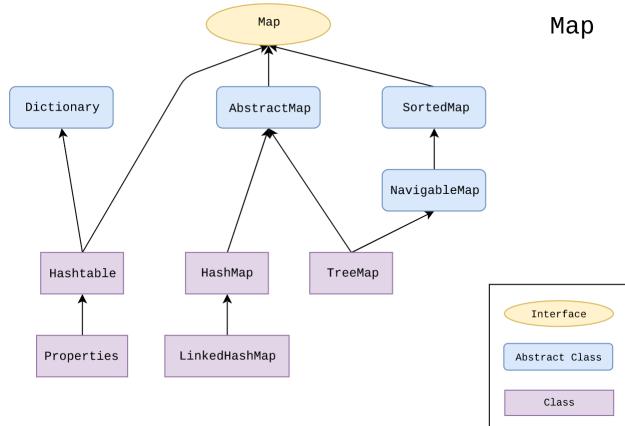
- Se preguntará al usuario si quiere repetir el proceso y se actuará en consecuencia

IMPORTANTE: En las tres opciones hay que buscar el libro en base al título del mismo. En lugar de repetir el código definir un método buscarLibro(título) que retorne la posición en la que se encuentra el libro o un valor -1 si no existe ningún libro con ese título. Emplear ese método en las búsquedas.

### 3.5. API de Colecciones

Las colecciones en Java proporcionan diversas clases e interfaces especializados en el almacenamiento de información y que amplían las posibilidades de las matrices. Hay gran cantidad de colecciones en la plataforma Java. Desde la primera versión de Java se dispone de las clases Vector y HashTable y del interfaz Enumeration. En la versión 1.2 del JDK se introdujo el Java Framework Collections o "estructura de colecciones de Java" (en adelante JCF). Se trata de un conjunto de clases e interfaces que mejoran notablemente las capacidades del lenguaje respecto a estructuras de datos. La estructura del JFC podemos dividirla en dos grandes bloques: colecciones y mapas:





Fuente: Wikipedia

Como se puede ver disponemos de una serie de interfaces que implementan clases abstractas de las que se crean clases estándar que podemos emplear en nuestras aplicaciones. Si necesitamos crear nuestra propia implementación de colecciones podríamos partir de un interfaz o de una clase abstracta. NOTA: Todas las colecciones de la JFC admiten genéricos

#### Interfaces de la JCF

Constituyen el elemento central de la JCF.

- Iterable : especifica métodos para obtener iteradores
- Collection: define métodos para tratar una colección genérica de elementos
- Set: colección que no admite elementos repetidos
- SortedSet: set cuyos elementos se mantienen ordenados según el criterio establecido (NOTA: aunque en la imagen aparece como clase abstracta, realmente es un interfaz)
- NavigableSet : set ordenado que se puede recorrer en ambos sentidos
- List: admite elementos repetidos y mantiene un orden inicial
- Map: conjunto de pares clave/valor, sin repetición de claves
- SortedMap: map cuyos elementos se mantienen ordenados según el criterio establecido
- Queue : permite gestionar colas de objetos (FIFO)
- Deque : similar pero permite el acceso por ambos extremos de la colección (FIFO y LIFO)

#### Interfaces de soporte:

- Iterator: sustituye a la interface Enumeration. Dispone de métodos para recorrer una colección y para borrar elementos.
- ListIterator: deriva de Iterator y permite recorrer listas en ambos sentidos.
- Comparable: declara el método compareTo() que permite ordenar las distintas colecciones según un orden natural (String, Date, Integer, Double, ...).
- Comparator: declara el método compare() y se utiliza en lugar de Comparable cuando se desea ordenar objetos no estándar o sustituir a dicha interface.

#### Clases de propósito general

Son las implementaciones directas de las interfaces de la JFC.

- HashSet: Interface Set implementada mediante una hash table.
- TreeSet: Interface SortedSet implementada mediante un árbol binario ordenado.
- Vector : Interface Collection : actualmente implementa otros interfaces por lo que es similar a ArrayList
- ArrayList: Interface List implementada mediante un array.
- LinkedList: Interface Queue implementada mediante una lista vinculada.
- HashMap: Interface Map implementada mediante una hash table.
- TreeMap: Interface SortedMap implementada mediante un árbol binario
- HashTable : Es la clase original para emplear conjuntos de elemento de tipo clave/valor

Además hay otras clases como Stack (que es un Vector para trabajar con pilas), LinkedHashMap (que es un HashMap cuyos elementos están doblemente enlazados) o PriorityQueue (cola con prioridades).

#### Collections (java.util.Collections)

La clase Collections (no confundir con la interface Collection, en singular) es una clase que define un buen número de métodos static con diversas finalidades. Los más

interesantes son los siguientes:

- `public static void sort(List)` : Ordena la lista
- `public static void sort(List, Comparator)` : ordena en función del objeto Comparator
- `public static void shuffle(List)` : Desordena la lista de modo aleatorio
- `public static void shuffle(List, Random)` : Idem empleando de base un número aleatorio
- `public static void reverse(List)` : Da la vuelta a la lista
- `public static int binarySearch(List, Object)` : Devuelve la posición del objeto en la lista empleando una búsqueda binaria
- `public static int binarySearch(List, Object, Comparator)` : Idem, pero basándose en el objeto Comparator
- `public static void copy(List, List)` : Copia una lista en otra
- `public static void fill(List, Object)` : Rellena una lista con el objeto especificado
- `public static Object max(Collection)` : Retorna el mayor elemento de la colección (emplea compareTo)
- `public static Object max(Collection, Comparator)` : Idem empleando el objeto Comparator
- `public static Object min(Collection)` : Retorna el menor objeto de la colección
- `public static Object min(Collection, Comparator)` : Idem empleando el objeto Comparator
- `static void swap(List,int,int)` : Intercambia los objetos de las posiciones indicadas

### 3.6. Interfaces Collection y List e implementaciones Vector ArrayList

#### Interface Collection (java.util.Collection)

Todos los interfaces y clases de la JFC implementan este interfaz y permite almacenar un grupo de objetos denominados elementos. Algunas colecciones permiten elementos duplicados y otras no. Algunas mantienen los elementos ordenados y otras no. Los métodos más importantes son:

- `public abstract boolean add(Object)` : Añade un objeto a la colección. Opcional, es decir, no está disponible en las clases que no permiten modificar sus objetos. En este caso devuelven una excepción UnsupportedOperationException
- `public abstract boolean addAll(Collection)` : Añade una colección a la actual. Opcional.
- `public abstract void clear()` : Elimina los objetos de la colección. Opcional
- `public abstract boolean contains(Object)`: true si el objeto está en la colección. Se emplea el método equals para hacer la comprobación.
- `public abstract boolean containsAll(Collection)` : true si todos los objetos están en la colección.
- `public abstract boolean isEmpty()` : true si la colección está vacía
- `public abstract java.util.Iterator iterator()` : Devuelve un objeto Iterator para recorrer los elementos de la colección
- `public abstract boolean remove(Object)` : Elimina el objeto especificado (en la búsqueda se emplea el método equals). True si se ha modificado la colección. Opcional
- `public abstract boolean removeAll(Collection)` : Elimina todos los objetos indicados. True si ha tenido éxito. Opcional.
- `public abstract boolean retainAll(Collection)` : Mantiene sólo los elementos especificados eliminando el resto. opcional
- `public abstract int size()` : Retorna el tamaño de la colección
- `public abstract Object[] toArray()` : Retorna una matriz de objetos con los elementos de la colección

#### Interfaz Iterator (java.util.Iterator)

La interface Iterator sustituye a Enumeration, utilizada en versiones anteriores del JDK. Se diferencia de dicha interfaz en que el interfaz Iterator permite eliminar elementos de la colección mientras se va recorriendo

- `public abstract boolean hasNext()` : true si hay otro elemento
- `public abstract Object next()` : Retorna el elemento siguiente
- `public abstract void remove()` : Elimina el último elemento accedido

#### Interfaz ListIterator (java.util.ListIterator)

La interfaz ListIterator desciende de Iterator y define mecanismos para recorrer una lista en ambas direcciones, modificarla durante el recorrido y obtener la posición actual.

- `public abstract void add(Object)`: Añade un nuevo objeto a la lista
- `public abstract boolean hasNext()` : true si hay un elemento siguiente
- `public abstract boolean hasPrevious()` : true si hay un elemento anterior
- `public abstract Object next()` : Retorna el siguiente elemento
- `public abstract int nextIndex()`: Retorna el índice (empezando por cero) del siguiente elemento de la lista
- `public abstract Object previous()` : Retorna el elemento anterior
- `public abstract int previousIndex()`: Retorna el índice del elemento anterior en la lista

- public abstract void remove() : Elimina el último elemento recuperado con un next() o un previous(). Opcional
- public abstract void set(Object) : Reemplaza el último elemento recuperado con un next() o un previous() con el objeto especificado. Opcional

#### Interfaz List (java.util.List)

La interface List define métodos para operar con colecciones accesibles a través de un índice y que pueden tener elementos repetidos. Por ello, dicha interface declara métodos adicionales que tienen que ver con el orden y con el acceso a elementos o rangos de elementos. Además de los métodos de Collection, la interface List declara los métodos siguientes:

- public abstract void add(int, Object) : Inserta el objeto en la posición indicada
- public abstract boolean addAll(int, Collection) : Inserta la colección de objetos en la posición indicada
- public abstract Object get(int) : Retorna el objeto en la posición dada
- public abstract int indexOf(Object) : Retorna la posición del objeto en la lista. Si el elemento no se encuentra se retorna un -1. Emplea el método equals para la búsqueda
- public abstract int lastIndexOf(Object) : Idem pero buscando desde el final
- public abstract ListIterator listIterator() : Retorna un objeto ListIterator para recorrer la lista
- public abstract ListIterator listIterator(int) : Idem pero desde la posición indicada
- public abstract Object remove(int) : Elimina el elemento en la posición indicada
- public abstract Object set(int, Object) : Modifica el elemento en la posición indicada, asignándole el objeto
- public abstract List subList(int, int) : Retorna una sublist entre las posiciones indicadas, ambas incluidas. Los cambios en la sublist afectan a la lista original

Existen dos implementaciones de la interface List, que son las clases ArrayList y LinkedList. La diferencia está en que la primera almacena los elementos de la colección en una matriz de Objects, mientras que la segunda los almacena en una lista vinculada. Además de estas clases, la clase Vector también implementa este interfaz. En el siguiente ejemplo se emplean varios de los métodos disponibles en el interfaz List. Para ello se emplea la implementación ArrayList:

NOTA: Dado que la clase Vector también implementa List, el programa también funcionaría con : Vector empleados = new Vector<>();

- Editar
- 

#### 3.7. Actividad 16 - Uso de Colecciones

##### Instrucciones

<https://classroom.github.com/a/vjXJM2LD> [Actividad 16-II \(Gestor de Eventos\)](#) Se desea diseñar una aplicación que permita gestionar los gastos de los comerciales de una empresa. Para gestionar esa información se va a diseñar una clase Gasto con los siguientes campos:

- num : un número que identifica al gasto y que deberá ser único (emplear una propiedad estática)
- concepto : el concepto del gasto
- fecha : la fecha del mismo
- importe : el importe del gasto

La clase dispondrá de un constructor con todos los argumentos (excepto num que se calculará automáticamente), y métodos set / get y toString. Además se dispondrá de una clase Comercial con los siguientes campos:

- codigo : El código del comercial (deberá ser único)
- nombre : El nombre
- apellidos : Los apellidos
- gastos : Un ArrayList donde almacenaremos los gastos del comercial. En principio, estará vacío

La clase tendrá los siguientes métodos:

- constructor con todos los campos excepto los gastos. Deberá asignar automáticamente un código diferente a cada comercial.
- set / get : De todos los campos
- toString
- equals : Basado en el código del comercial
- getTotalImporteGastos : este método retornará la suma de los importes de todos los gastos del comercial. Para ello

Inicializará una variable totalGastos a 0 Recorrerá la lista de gastos

Sumará el importe del gasto a totalGastos

Retornará totalGastos

Adicionalmente definiremos un interfaz GastosDAO con los siguientes métodos:

- void nuevoComercial(Comercial nuevo)

- Comercial getComercial(int codigo)
- List getComerciales();
- void nuevoGasto(int codigo, Gasto nuevo)
- void eliminarGasto(int codigo, int num)

Definiremos una implementación del interfaz en la clase GastosList que:

- Definirá una propiedad de tipo List llamada comerciales para guardar los datos de todos los comerciales de la empresa. Se inicializará con un ArrayList vacío
- Definirá los siguientes métodos:

void nuevoComercial(Comercial nuevo) : Añadirá el comercial nuevo a la lista de comerciales void eliminarComercial(int codigo) : Eliminará el comercial con el código indicado de la lista de comerciales (si existe, si no, no haremos nada) Comercial getComercial(int codigo) : Retornará el comercial con el código indicado si existe en la lista de comerciales. Si no, retornará null List getComerciales() : Retornará la lista de comerciales void nuevoGasto(int codigo, Gasto nuevo) : Buscará el comercial con el código indicado. Si existe, se añadirá el gasto nuevo a su lista de gastos (que cogeremos con el método getGastos() del comercial). void eliminarGasto(int codigo, int num) : Realizará las siguientes tareas:

Buscará el comercial con el código indicado Si existe

Eliminará de sus gastos (getGastos()) el que tenga el num indicado

Definirá un constructor sin argumentos en el que:

Creará 5 comerciales con datos inventados y los añadirá a la lista de comerciales con el método nuevoComercial

Si queréis (no es obligatorio) definir una clase de servicio (GastosServicio) con los mismos métodos de GastosList. El programa definirá una propiedad para almacenar el DAO (o el servicio si lo habéis creado) y mostrará el siguiente menú:

1. Nuevo Gasto
2. Eliminar Gasto
3. Listar Gastos de un Comercial
4. Listar Gastos Totales
5. Salir

Opción [1-6] : Para el programa se definirá una propiedad estática de tipo DateFormat con formato SHORT y otra de tipo NumberFormat que permita almacenar decimales. Además se definirá un método llamado pedirCodigo (recordad que debe ser estático) que no reciba nada y retorne un entero. El método

- Recuperará todos los comerciales del DAO (o del servicio)
- Pedirá el código del comercial mostrando los datos de los comerciales. Por ejemplo:

Comerciales =====

Código	Nombre	Apellidos
cod1	nombre	apellidos
cod2	nombre	apellidos
...		

Código del comercial :

- Retornará el código introducido por el usuario

### Nuevo Gasto

En este apartado se pedirá en primer lugar el código del comercial que ha hecho el gasto con el método anterior y se recuperará el comercial del DAO con ese código. Si no existe, se dará un mensaje de error. Si existe :

- Pediremos el concepto asociado al gasto
- Pediremos la fecha del gasto en formato corto. Emplearemos la propiedad de tipo DateFormat definida anteriormente para obtenerla
- Pediremos el importe del gasto empleando la propiedad de tipo NumberFormat
- Con los datos obtenidos, crearemos un nuevo gasto y lo añadiremos al comercial mediante el método correspondiente del DAO

### Eliminar Gasto

Al igual que en el caso anterior pediremos el código del comercial del que se quiere eliminar el gasto y lo buscaremos mediante el método anterior:

- Si no se encuentra se dará un mensaje de error
- Si se encuentra, se mostrarán los datos del comercial y sus gastos y se pedirá el número del que se desea eliminar:

### Eliminar Gasto de nombre apellidos

| Número | Concepto | Fecha | Importe || n1 | concepto | dd/mm/aa | importe || n2 | concepto | dd/mm/aa | importe || ... |||

Número de Gasto a Eliminar:

- Se llamará al método correspondiente del DAO para eliminar el gasto

### Listar Gastos de un Comercial

Se pedirá el código del comercial con la función creada al principio

- Si no se encuentra, se dará un mensaje de error
- Si se encuentra se mostrará el siguiente listado:

Gastos de nombre apellido (código) Fecha Concepto Importe fecha concepto importe ... Importe Total : getImporteGastos()

### Resumen de Gastos

En este caso se mostrarán todos los gastos de todos los comerciales: Comercial Importe Gastos nombre apellido getImporteGastos() ... Gastos Totales : Suma(getImporteGastos()) NOTA: En los listados se mostrará la fecha y los importes correctamente formateados

### 3.8. Operaciones Básicas con Matrices Colecciones Ordenadas

Una matriz / colección ordenada es una matriz / colección cuyos elementos están ordenados siguiendo un determinado criterio en ascendente o en descendente.

## Búsquedas

A la hora de buscar en una matriz ordenada tenemos dos opciones:

### Búsqueda Ordenada

La búsqueda ordenada es una búsqueda secuencial en la que aprovechamos el hecho de que la matriz esté ordenada. En este caso la búsqueda se basa en recorrer la matriz mientras no lleguemos al final (si la matriz está llena será lo que indique length, si no lo está, lo que indique el contador) y lo que estamos buscando sea mayor (si la matriz está ordenada en ascendente) o menor (si está ordenada en descendente) que el valor actual. Si al salir de la búsqueda hemos llegado al final o no hemos encontrado lo que buscamos, la búsqueda no ha tenido éxito y nos habremos parado en la posición en la que debería estar el elemento. Si no hemos llegado al final y hemos encontrado lo que buscábamos, la búsqueda ha concluido con éxito y tendremos la posición del elemento buscado en la matriz. Evidentemente, para poder emplear búsquedas ordenadas es imprescindible que la matriz esté ordenada por el mismo criterio por el que vamos a buscar. De no ser así, tendríamos que hacer una búsqueda desordenada. Otra opción es buscar todos los elementos de una colección que cumplan una condición si la colección está ordenada por ese criterio. Por ejemplo, buscar los alumnos de una clase si la colección está ordenada en base a dicho campo. El proceso sería el siguiente:

- Buscar el primer elemento que cumpla la condición (el primer alumno de la clase). Dado que la matriz está ordenada emplearemos búsqueda ordenada
- Si existe, todos los elementos que cumplen la condición deben estar seguidos (todos los alumnos de la clase irán seguidos) por lo que habrá que repetir el proceso para cada elemento mientras no lleguemos al final de la colección Y se siga cumpliendo la condición

El siguiente ejemplo, muestra ambos casos con matrices y con listas:

### Búsqueda Binaria

La búsqueda binaria se basa en dividir la zona de búsqueda en dos en cada paso. Para ello se compara el valor a buscar con el valor que se encuentra en el medio de la matriz. Si está ahí, hemos acabado. Si no está debaremos continuar la búsqueda en la parte superior o inferior en función de la relación entre el elemento medio y el valor buscado (y en si la matriz está ordenada en ascendente o en descendente). Para realizar el proceso deberemos controlar las posiciones menor y mayor en las que estamos buscando en cada momento. En caso de buscar en la parte superior cambiaremos el valor menor al medio más 1, en caso de buscar en la parte inferior cambiaremos el mayor por medio menos 1. En cualquier caso habrá que recalcular el nuevo punto medio. ¿Qué ocurre si no está? Llegará un momento en el que el elemento menor y el mayor coincidan y, en el siguiente paso, llegaremos a que el elemento menor será mayor que el mayor lo cual no es posible. De modo que sabremos que no lo hemos encontrado en esta circunstancia. Evidentemente, deberemos incluir en la repetitiva esta condición. Modificamos el ejemplo anterior para que emplee búsqueda binaria:

### Complejidad de los algoritmos

Al definir algoritmos hay un término que debemos comprender y es la complejidad de un algoritmo( $O()$ ). La complejidad se podría definir como el número de pasos que tendríamos que dar en el peor de los casos para que nuestro algoritmo obtenga el resultado deseado. En el caso de la búsqueda desordenada y la búsqueda ordenada y, asumiendo en casos, la complejidad de los algoritmos sería  $O(n)$  dado que, en el peor de los casos, necesitamos comprobar todos los elementos de la colección. Es decir, para localizar un elemento entre 2048 necesitaríamos (en el peor de los casos) 2048 búsquedas. En el mismo caso en la búsqueda binaria la complejidad sería  $O(\log(n))$  donde  $\log$  es  $\log_2$ . En este caso, para localizar un elemento entre 2048 necesitaríamos en el peor de los casos  $\log_2(2048)=11$  búsquedas. Evidentemente un algoritmo de complejidad  $O(n)$  es peor que otro de complejidad  $O(\log(n))$ . NOTA: La complejidad no es lo único que deberemos mirar a la hora de decidir emplear un algoritmo u otro. La búsqueda binaria es mucho más eficiente que la secuencial pero el algoritmo es más complejo (más lento) y para usarla necesitamos que la colección esté ordenada lo que puede implicar la necesidad de ordenarla. Estos condicionantes pueden hacer que sea mejor emplear búsqueda secuencial en determinados escenarios. La complejidad no sólo se aplica a las búsquedas. ¿Cuál es la complejidad de coger un elemento de una matriz en base a su índice?  $O(1)$  ¿La de añadir un elemento?  $O(n)$ . ¿Y la de eliminarlo?  $O(n)$

### Inserción Ordenada

Para añadir elementos a una matriz ordenada tenemos que dar los siguientes pasos:

- Localizar la posición en la que debiera ir el elemento (búsqueda ordenada)
- "Hacer hueco" lo que implica mover todos los elementos desde el último hasta el que está en la posición obtenida una posición a la derecha
- Situar el elemento en la posición
- Incrementar el número de elementos.

El código:

### Eliminación

La eliminación es exactamente igual que en el caso de matrices desordenadas. La única diferencia es que la búsqueda del elemento a eliminar la podemos hacer mediante búsqueda ordinaria o búsqueda binaria.

### Listado Agrupado (Roturas de Control)

Si tenemos una colección ordenada por uno o más criterios, podemos recorrer dicha colección de manera agrupada (por ejemplo, listar todos los empleados agrupados por departamento). Para ello el proceso es simple:

- Realizamos las tareas que hay que hacer una vez para el programa (cabeceras, inicializar totales finales...)
- Repetimos el proceso mientras queden datos

Realizamos las tareas que hay que hacer una vez por cada agrupación (cabeceras de grupo, inicializar totales de grupo...) Almacenamos en una variable el campo por el que queremos agrupar. Repetimos el proceso mientras queden datos y el elemento actual pertenezca al grupo

Realizamos las tareas por cada elemento (mostrar, incrementar totales...)

Realizamos las tareas que hay que hacer cada vez que acaba un grupo (mostrar totales de grupo, incrementar totales finales con los finales del grupo...)

- Realizamos las tareas que hay que hacer al final del programa (mostrar totales finales...)

Evidentemente, si queremos varios niveles de agrupamiento habría que repetir el proceso (guardar el campo de agrupación y la repetitiva) por cada uno de ellos. Veamos un ejemplo en el que vamos a mostrar los datos de los empleados por Departamento (incluyendo el número de empleados y el sueldo medio por departamento y en total). Queda claro que es imprescindible que los empleados estén ordenados por departamento: La salida:

#### 3.9. Actividad 17 - Matrices Ordenadas Búsqueda

##### Instrucciones

<https://classroom.github.com/a/gNkQN5Xi> Se desea diseñar una aplicación que permita gestionar las valoraciones que hacen los usuarios sobre las series de una plataforma. De cada serie se almacenará la siguiente información:

- título : El título de la serie (único)
- sinopsis : La sinopsis de la misma (un pequeño resumen)
- votos : Cuántos usuarios han valorado la serie
- valoraciónTotal : La suma de todas las valoraciones de los usuarios

La clase dispondrá de métodos get/set para los campos título y sinopsis y un método get para los votos. Además dispondrá de un constructor que reciba el título y la sinopsis y los siguientes métodos:

- votar(int valoracion) : Incrementará el número de votos en 1 y la valoraciónTotal en lo indicado en valoración
- getValoracion() : Retornará la valoración media de la serie.

Adicionalmente se definirá un interfaz DAO (SeriesDAO) con los siguientes métodos:

- Serie getSerie(String título)
- Serie getSerieBinario(String título)
- List getSeries()
- List buscarSeries(String texto)

Además se crearán las clases SeriesMatriz y SeriesList que implementarán dicho interface con matrices y listas. La clase SeriesMatriz asumirá que puede haber un total de 100 series. Introducir datos de prueba en ambas cases asumiendo que la lista o matriz deberá estar ordenada por título. En las implementaciones:

- El método getSerie retornará la serie con el título indicado empleando búsqueda ordenada. Si la serie no existe retornará null.
- El método getSerieBinario hará lo mismo pero mediante búsqueda binaria
- El método getSeries retornará todas las series.
- El método buscarSeries retornará un lista de series con todas las series cuyo título empiece (startsWith) con el texto indicado. En la búsqueda

Se definirá una variable de tipo List para almacenar las series encontradas con un ArrayList vacío. Se buscará la primera serie que comiencen por dicho título mediante búsqueda ordenada

Si la hay se añadirá dicha serie a la lista mientras no lleguemos al final y se siga cumpliendo la condición (comiencen por el texto)

Se retornará la lista con las series encontradas (que estará vacía si no cumple ninguna)

Se pide diseñar una aplicación que mostrará el siguiente menú:

1. Votar Serie Buscar Series Salir

Opción [1-3]:

#### Votar Serie

En esta opción:

- Se pedirá el título de la serie y se buscará mediante el dao mediante búsqueda binaria.

Si la serie no existe, se dará un mensaje de error adecuado. Si existe

se mostrarán sus datos (título, sinopsis, votos y valoración media) y se pedirá la valoración del usuario se votará la serie con el método correspondiente se mostrará un mensaje agradeciendo el voto

- el proceso podrá repetirse a petición del usuario

### Buscar Series

En esta opción se van a buscar todas las series que empiecen por un determinado texto. Por ejemplo, si el usuario escribe Juego deberán mostrarse todas las series que comiencen por Juego. El proceso a realizar será el siguiente:

- Se pedirá el texto a buscar
- Se obtendrán las series que comienzan por ese texto mediante el DAO
- Si hay alguna

Se listarán todos sus datos

- Se mostrará cuántas series se han encontrado y la valoración media de todas ellas.
- El proceso podrá repetirse a petición del usuario

Al finalizar la aplicación se mostrará por consola un listado con toda la información de todas las series.

### 3.10. Actividad 17-2 - Matrices Ordenadas Inserción y Eliminación

#### Instrucciones

Modificar la Actividad anterior para añadir las siguientes opciones al DAO y la implementación para añadir las opciones en matrices y listas.

- void nuevaSerie(Serie s) : Insertará la serie en la posición que le corresponda en función del título de modo que la colección siga ordenada por ese campo
- void eliminarSerie(String título) : Eliminará la serie con el título indicado

Además se modificará la aplicación para añadir dos opciones más:

1. Votar Serie Buscar Series Añadir Serie Eliminar Serie Salir

Opción [1-5]:

### Añadir Serie

En esta opción:

- Se pedirá el nombre de la serie
- Se comprobará que no haya ninguna serie en la matriz con el mismo nombre
- Si la hay se dará un mensaje de error
- Si no la hay

Se pedirán el resto de los datos y se añadirá mediante el DAO. Se dará un mensaje indicando que se ha añadido la serie correctamente

- El proceso podrá repetirse a petición del usuario

### Eliminar Serie

En esta opción:

- Se pedirá el título de la serie a eliminar
- Se buscará la serie (DAO) mediante búsqueda binaria
- Si no está se dará un mensaje de error
- Si está

Se mostrarán el resto de los datos y se pedirá confirmación para eliminarla. Si se confirma

Se eliminará la serie. Se dará un mensaje indicando esta circunstancia

- El proceso podrá repetirse a petición del usuario

### 3.11. Ejemplo Con Expresión Lambda Avanzado

Una expresión lambda es una forma compacta de crear un objeto de una clase que disponga de un único método. Para conseguir que sólo haya un método se emplea un interfaz funcional. Veamos un ejemplo simple. Supongamos que queremos realizar operaciones genéricas sobre un par de números (sumar, restar...) empleando diferentes objetos, pero queremos poder emplear cualquiera de esos objetos indistintamente en otros métodos. Una solución pasaría por crear un interfaz como el siguiente: public interface Operar { public int operar(int n1, int n2); } El siguiente paso sería crear las clases que implementan el interfaz: public class Suma implements Operar { public int operar(int n1, int n2){ return n1+n2; } } public class Resta implements Operar { public int operar(int n1, int n2){ return n1-n2; } } ... Ahora podríamos crear objetos de ambas clases y ejecutar el código sobre ellos. Además podríamos definir métodos genéricos que puedan trabajar sobre dichos objetos public class Ejemplo { public static void ejecutarMetodo(Operar oper, int n1, int n2) { System.out.println("El resultado de la operación es: " + oper.operar(n1,n2)); } public static void main(String[] args){ Operar suma = new Suma(); Operar resta = new Resta(); ejecutarMetodo(suma,10,7); ejecutarMetodo(resta,10,7); } } El código: Evidentemente, podríamos añadir nuevas clases que realizaran cualquier operación y nuestro método sería capaz de utilizarlas.

## Clases Anónimas

En la aplicación, las clases Suma y Resta sólo se definen para crear un único objeto. Podemos simplificar su uso, creando a la vez la clase y el objeto. La idea es pasar el código de la clase a la hora de crear el objeto. Por ejemplo, el objeto suma ahora: Operar suma = new Operar(){ public int operar(int n1, int n2){ return n1+n2; }}; Como se ve, a la hora de crear el objeto se pone como clase el Interfaz que debe cumplir nuestra nueva clase (también se podría poner una clase padre de la que extienda) por lo que nuestra clase no puede tener nombre (es anónima). Tras ello, se pone el código de la clase. Ejemplo:

NOTA: Podemos ver que se generan las clases anónimas Main\$1.class y Main\$2.class.

## Expresiones Lambda

El uso de objetos de clases anónimas que sólo tienen un método (implementan un interfaz funcional) es muy habitual. Las expresiones lambda permiten precisamente crear estos objetos de una forma muy simple. En dicha expresión se va a indicar únicamente el código de dicha función. La expresión tiene los siguientes formatos: (listaArgs) -> valorRetorno; (listaArgs) -> { código; [return valorRetorno]; } Clase|objeto::método donde :

- listaArgs : Son los argumentos del método definido en el interfaz (en nuestro caso n1 y n2). Si sólo hay un argumento no es necesario poner paréntesis
- valorRetorno : Es el valor que retorna la función.
- Clase|objeto::método : Permite simplificar la expresión cuando queremos el valor que retorna un método de una clase cuyos parámetros coinciden con los del método del interfaz (veremos ejemplos más adelante)

Por tanto, ahora podríamos crear el objeto suma a través de las siguientes expresiones lambda: Operar suma = (n1,n2) -> n1+n2; Operar suma = (n1,n2) -> { return n1+n2; } El código ahora:

## Uso de expresiones Lambdas para búsqueda avanzada

Uno de los problemas que podemos tener a la hora de buscar información es que, dado que la búsqueda siempre es igual, buscar por diferentes criterios nos obliga a reescribir el mismo código. Si pensamos un poco en una búsqueda, para buscar lo único que necesitamos es poder comparar dos objetos para saber si son iguales o no. Aprovechándonos de eso podemos definir un interfaz funcional para comparar dos objetos cualesquiera empleando genéricos. Como ya se indicó un genérico permite poner como tipo de un objeto una especie de variable (genérico) que se definirá cuando se emplee dicho genérico. Definimos el interfaz funcional con una clase genérica (T): public interface Buscar { public boolean cumple(T); } En este caso definimos un único método que debería retornar un valor true si el objeto recibido cumple la condición y false en caso contrario. A partir de este interfaz ya podemos definir un método que permita buscar todos los objetos que cumplan una condición en una lista de objetos: Vamos a añadir a los DAOs del ejercicio anterior un método genérico para buscar empleados que reciba una expresión lambda con la búsqueda a realizar. El método sera e mismo en ambos casos: public List buscar(Buscar búsqueda) { List empleados = new ArrayList<>(); for(Emppleado e : empleados){ if (búsqueda.cumple(e)){ empleados.add(e); } } return empleados;} Ahora podemos emplear el método con diferentes métodos de búsqueda. El código:

### 3.12. Ordenación

Los algoritmos de ordenación nos permite ordenar matrices (colecciones) en base a algún criterio. Hay múltiples algoritmos de ordenación y veremos algunos de ellos.

#### Ordenación por Inserción

La idea del algoritmo es ir comparando cada elemento con el siguiente e intercambiando si no están correctamente ordenados. Al finalizar la primera pasada tendremos en la primera posición el menor (ascendente) o mayor (descendente) elemento de la matriz. Si repetimos el algoritmo desde el segundo elemento hasta el final ordenaremos este segundo elemento. Al repetir este proceso para todos los elementos (hasta el anteúltimo) conseguiremos tener la matriz ordenada.

#### Ordenación por Selección

La idea del algoritmo es localizar, en la primera pasada, la posición del menor (ascendente) o mayor (descendente) elemento desde el elemento inicial al último. Si ese elemento no está ya en la primera posición, se intercambian. El proceso se repite para el resto de las posiciones

#### Ordenación por el método de la Burbuja

La idea del algoritmo es ir comparando los elementos de dos en dos y poniéndolos en el orden correcto. En el primer paso el último elemento será el más bajo (descendente) / alto (ascendente) de la matriz. En el segundo paso el anteúltimo será el más alto / bajo siguiente, etc. Se llama método de la burbuja porque los elementos más pesados (en ascendente) tienden a ir al final de la matriz.

#### Ordenación QuickSort

Con optimizaciones es el método más rápido. Se basa en elegir un elemento de la matriz, denominado pivote y ordenar los restantes elementos de la matriz respecto al mismo, es decir, dejando los elementos más bajos a la izquierda del pivote y los más altos a la derecha (en ascendente). Para ello se recorre la matriz de izquierda a derecha y de derecha a izquierda comprobando si los elementos están bien ordenados hasta que los índices se crucen. Los elementos que no estén ordenados se intercambian y, al final, se sitúa el pivote en el punto de cruce. Ahora realizamos el mismo proceso con las dos listas de la izquierda y de la derecha. El proceso se repite recursivamente hasta que sólo queda un elemento a ordenar. La forma más simple de utilizarlo es emplear como pivote el último elemento de la matriz a ordenar

#### Complejidad

La complejidad de los algoritmos de selección, inserción y burbuja es  $O(n^2)$  dado que debemos recorrer la matriz completa dos veces. En el caso del algoritmo quicksort es algo más complicado dado que, en el peor caso (al dividir la matriz en dos en cada caso queda una vacía) sería  $O(n^2)$  y en el mejor (siempre dividimos la matriz por la mitad) sería  $O(n \log(n))$ . En todas las pasadas del algoritmo recorremos todos los elementos pero en el primer caso la división se hace tantas veces como elementos (dado que siempre nos quedamos con una de las matrices vacías) mientras que en el segundo se hace la mitad de veces. Si elegimos como pivote un elemento aleatorio de los disponibles, el algoritmo se acerca siempre a  $O(n \log(n))$ . La solución más simple sería elegir una posición al azar entre las disponibles y luego intercambiárla con el mayor elemento. De este modo nos valdría el algoritmo anterior.

- Editar
- 

### 3.13. Actividad 22 - Ordenación de matrices

## Instrucciones

Partiendo de la clase Libro creada en la Actividad 14 vamos a crear una clase ejecutable llamada Librería que disponga de la misma matriz de libros que en dicho ejercicio. El programa mostrará el siguiente menú al usuario mediante un JOptionPane:

1. Listar Libros Alfabéticamente Mostrar Libros de un Tema Listar Libros por Temas Listar Libros por Precio Salir

Opción [1-5]

### Mostrar Libros Alfabéticamente

- Se ordenará la matriz por el método de Inserción en base al título del libro.
- Se realizará un listado similar al siguiente:

Listado de Libros Título Tema Precio Unidades Título Tema Precio Unidades ... Suma Suma

### Mostrar Libros de un Tema

- Se ordenará la matriz en base al tema por el método de selección
- Se pedirá el tema a listar y se buscará mediante búsqueda ordenada
- Si no se encuentra se dará un mensaje de error
- Si se encuentra se mostrará el siguiente listado :

Tema : nombre Título Precio Unidades Título Precio Unidades .... Se han encontrado n libros del tema. NOTA : Tener en cuenta que todos los libros del tema estarán seguidos, así que la repetitiva será mientras no lleguemos al final de la matriz Y el libro sea del tema

### Listar Libros por Temas

- Se ordenará la matriz en base al tema por el método de la Burbuja
- Se mostrará un listado similar al siguiente

Tema :tema Título Unidades Precio Título Unidades Precio ... Tema :tema Título Unidades Precio Título Unidades Precio ...

- El proceso a realizar será el siguiente:
- Se inicializará una variable pos con el valor 0
- Mientras pos no llegue al final de la matriz

Se guardará en un String llamado tema el tema del libro que está en pos. Se escribirá el tema y las cabeceras

- Mientras pos no llegue al final de la matriz Y la variable tema y el tema del libro que está en pos SEAN IGUALES

Mostraremos los datos del libro

- Sumaremos 1 a pos

### Listar Libros por Precio

- Se ordenará la matriz por el método quicksort al precio y en descendente
- Se pedirá el precio máximo que se quiere listar
- Se mostrará el siguiente listado con los libros cuyo precio sea MENOR o IGUAL al introducido :

Listado de Libros Título Tema Unidades Título Tema Unidades ... Se han encontrado n libros NOTA: Si no hay ningún libro que cumpla la condición se deberá mostrar un mensaje de error indicando cuál es el precio máximo de los libros de la matriz

### 3.14. Ejemplo Examen

En una tienda de juegos quieren una aplicación para gestionar los préstamos de los mismos. Para ello se desea diseñar una clase denominada Juego con los siguientes campos:

- titulo : El título del juego
- tipo : De qué tipo de juego se trata : RPG, FPS, Estrategia...
- cliente : El nombre de la persona que tiene alquilado el juego (si está alquilado o un valor null si el juego está disponible)
- coste : El coste del alquiler

La clase deberá disponer de un constructor con todos los datos y de métodos set/get apropiados. Se deberá definir un interfaz (JuegosDAO) con los siguientes métodos.

- Juego buscarJuego(String título)
- void nuevoJuego(Juego nuevo)
- void eliminarJuego(String título)
- List getAlquileresCliente(String cliente)
- List getJuegosPorTipo()

La implementación (JuegosImpl) se hará mediante matrices. Partiremos de una matriz de hasta 30 juegos:

- Juego buscarJuego(String título) : En este método:

Se buscará el juego con el título indicado mediante búsqueda desordenada Si existe se retornará el juego Si no existe se retornará null

- void nuevoJuego(Juego nuevo) : Este método añadirá el juego al final de la matriz
- void eliminarJuego(String título) : En este método:

Se ordenará la matriz por el método de selección Se buscará el juego con el título indicado mediante búsqueda binaria Si existe, se eliminará

- List getAlquileresCliente(String cliente) : Este método

Definirá un ArrayList Buscará el primer juego del cliente mediante búsqueda desordenada

Si no existe retornará la lista Si existe

Añadirá todos los juegos del cliente a la lista (recordad que la matriz está desordenada) Retornará la lista

- List getJuegosPorTipo() : Este método:

Ordenará la matriz por tipo mediante el método de la burbuja Creará un ArrayList Añadirá todos los juegos a dicha lista La retornará

La aplicación a desarrollar se gestionará mediante el siguiente menú: Alquileres de Juegos

1. Nuevo Juego Eliminar Juego Nuevo Alquiler Ver Alquileres de un Cliente Ver Juegos por Tipo Salir

Opción [1-6]

## Nuevo Juego

En esta opción:

- Se pedirá el título del juego y se comprobará si existe (getJuego)

Si existe, se dará un mensaje de error Si no existe

se pedirán el resto de los datos se creará un juego con esos datos se añadirá a la colección (nuevoJuego)

## Eliminar Juego

En esta opción:

- Se pedirá el título del juego y se comprobará si existe (getJuego)

Si no existe, se dará un mensaje de error Si existe y está alquilado (el cliente no es null) se dará un mensaje de error indicando que no se puede eliminar un juego alquilado Si no

Se mostrarán sus datos y se pedirá confirmación para eliminarlo Si se confirma, se eliminará de la matriz (eliminarJuego)

## Nuevo Alquiler

En esta opción

- Se pedirá el título a alquilar y se comprobará que existe (getJuego)

Si existe, se comprobará si el juego está disponible (cliente es null):

Si lo está se pedirá el nombre del cliente, se almacenará en el campo cliente y se mostrará el coste del alquiler Si no lo está, se dará un mensaje indicando que el juego ya está alquilado

Si no existe, se dará un mensaje de error indicándolo

- El proceso podrá repetirse mientras el usuario lo quiera

## Ver Alquileres de un Cliente

En esta opción

- Se pedirá el cliente y se recuperarán los alquileres del cliente
- Si no hay alquileres se dará un mensaje indicándolo
- Si los hay, se listarán según el siguiente formato:

Cliente : cliente Juego Tipo Coste Título Tipo Coste ... Total :  $\sum$ coste

## Ver Juegos por Tipo

En este proceso:

- Se ordenará la matriz en base al tipo (getJuegosPorTipo)
- Se listarán los alquileres agrupados por tipo. Para ello

Mientras no lleguemos al final

Se almacenará el tipo

Mientras haya juegos y sean de ese tipo Se listará el juego

Se pasará al siguiente juego

- El listado tendrá el siguiente aspecto:

Tipo : tipo Titulo Cliente Coste titulo cliente coste ... Suma(coste)

Tipo : tipo

Titulo Cliente Coste

titulo cliente coste

\* \* \*

Suma(coste)

... NOTA: La suma de los costes sólo se hará de los juegos alquilados. NOTA: Todos los campos numéricos se mostrarán y pedirán en castellano (no es necesario validar la información)

3.15. Matrices Multidimensionales Matrices Anidadas y clase Arrays

Matrices Multidimensionales (Matrices Anidadas)

Los lenguajes de programación disponen de la posibilidad de definir matrices de múltiples dimensiones. En el caso de Java conseguimos algo similar definiendo matrices de matrices (matrices anidadas). Dado que una matriz puede contener elementos de cualquier tipo, dichos elementos pueden ser, a su vez, matrices creando efectivamente matrices de dos dimensiones. Esas segundas matrices pueden estar formadas a su vez por matrices con lo que conseguiríamos matrices de 3 dimensiones, etc. Definimos una matriz cuadrada de enteros de 10x2 y una de cadenas de 3 dimensiones; int[][] matriz = new int[10][2]; String[][] cadenas = new String[4][3][2]; También podemos crear matrices multidimensionales asignándole los valores (en este caso las matrices no tienen por qué ser cuadradas: int[] mat = { {1,2,3}, {4,5}, {6,7,8,9} }; String[][] = { {"Juan", "Ana"}, {"Carlos", "Luisa", "Javier"}, { {"Eneko"}, {"Marta", "Eva", "Sonia"}, {"Miguel", "Enrique"} }; Veamos un ejemplo de cómo recorrer estas matrices:

```
public class MatricesMultidimensionales {
```

```

public static void main(String[] args) {
 int[][] mat = {
 {1,2,3},
 {4,5},
 {6,7,8,9}
 };
 String[][][] nombres = {
 { {"Juan","Ana"}, {"Carlos","Luisa","Javier"} },
 { {"Eneko"}, {"Marta","Eva","Sonia"}, {"Miguel","Enrique"} }
 };
 for(int i=0;i<mat.length;i++){
 for(int j=0; j<mat[i].length; j++){
 System.out.print("["+i+", "+j+"] = " + mat[i][j] + " ");
 }
 System.out.println("");
 }
 for(String[][] dim1:nombres){
 for(String[] dim2:dim1){
 System.out.print("[");
 for(String nombre:dim2){
 System.out.print(nombre+" ");
 }
 System.out.print("] ");
 }
 System.out.println("");
 }
}

```

La salida:  
[0,0]=1 [0,1]=2 [0,2]=3 [1,0]=4 [1,1]=5 [2,0]=6 [2,1]=7 [2,2]=8 [2,3]=9 [ Juan Ana ] [ Carlos Luisa Javier ] [ Eneko ] [ Marta Eva Sonia ] [ Miguel Enrique ]

Como vemos hemos empleado en un caso el for normal y en el otro el for de colecciones.

NOTA: Las matrices de múltiples elementos no son fáciles de gestionar y no es probable que se necesiten excepto cuando se trata de matrices de dos dimensiones.

## ## Clase Arrays

En Java se dispone de la clase `java.util.Arrays` que nos permite realizar diferentes operaciones con matrices. Entre ellos destacan los métodos:

- `binarySearch` : Múltiples métodos para emplear la búsqueda binaria en matrices ordenadas. Se puede buscar en todos los tipos básicos y en objetos (lo veremos en el próximo apartado)
- `copyOf / copyOfRange` : Retorna una copia de una matriz / parte de una matriz en
- `equals / deepEquals` : Retorna true si dos matrices tienen los mismos elementos. El método `deepEquals` se debe utilizar en matrices anidadas
- `fill` : Rellena todo o parte de una matriz con un valor dado
- `sort` : Ordena la matriz en ascendente. Se puede buscar en todos los tipos básicos y en objetos (lo veremos en el próximo apartado)
- `toString / deepToString` : Retorna una cadena con los elementos de la matriz (`deepToString` se debe usar con matrices anidadas)

Vemos un ejemplo con nombres:

```

``` java
import java.util.Arrays;

public class EjemploClaseArrays {

    public static void main(String[] args) {
        String[] nombres = {
            "Juan", "Ana", "Carlos", "Luisa", "Javier", "Eneko",
            "Marta", "Eva", "Sonia", "Miguel", "Enrique"};
        System.out.println(Arrays.toString(nombres));
        System.out.println(Arrays.toString(Arrays.copyOf(nombres, 4)));
        System.out.println(Arrays.toString(Arrays.copyOf(nombres, 15)));
        System.out.println(Arrays.toString(Arrays.copyOfRange(nombres, 2, 4)));
        System.out.println(Arrays.equals(Arrays.copyOfRange(nombres, 2, 4),
            new String[]{"Carlos", "Luisa"}));
        System.out.println(Arrays.equals(Arrays.copyOfRange(nombres, 2, 4),
            new String[]{"Carlos", "Luisa", "Juan"}));
        Arrays.sort(nombres);
        System.out.println(Arrays.toString(nombres));
        System.out.println(Arrays.binarySearch(nombres, "Eneko"));
        System.out.println(Arrays.binarySearch(nombres, "Belén"));
        Arrays.fill(nombres, 8, nombres.length, "Borrado/a");
        System.out.println(Arrays.toString(nombres));
    }
}

```

La salida: [Juan, Ana, Carlos, Luisa, Javier, Eneko, Marta, Eva, Sonia, Miguel, Enrique][Juan, Ana, Carlos, Luisa][Juan, Ana, Carlos, Luisa, Javier, Eneko, Marta, Eva, Sonia, Miguel, Enrique, null, null, null, null][Carlos, Luisa]truefalse[Ana, Carlos, Eneko, Enrique, Eva, Javier, Juan, Luisa, Marta, Miguel, Sonia]2-2[Ana, Carlos, Eneko, Enrique, Eva, Javier, Juan, Luisa, Borrado/a, Borrado/a, Borrado/a]

3.16. Clase Arrays Ordenación de Objetos Interfaces Comparable Comparator y Expresiones Lambda

Hemos visto que la clase Arrays es capaz de ordenar matrices de tipos básicos pero ¿qué pasa con las matrices de objetos? ¿cómo se puede definir un método para trabajar con objetos que se desconocen? Si nos planteamos el tema de la ordenación, vemos que para ordenar cualquier colección de elementos lo único que necesitamos saber es si un elemento es mayor, menor o igual a otro. Si supiéramos que una clase tiene un método que nos dé esa información y supiéramos cómo se llama ese método podríamos ordenar los elementos de dicha clase. Para conseguir esto en POO disponemos de los interfaces:

Interface Comparable

Como ya hemos visto un interface es una colección de métodos que deberemos implementar. Dado que para ordenar elementos sólo necesitamos poder compararlos, si tenemos un interface con un método que permita comparar dos objetos podremos ordenar una colección de elementos de cualquier clase que cumpla el interfaz. Precisamente el interface Comparable dispone de un único método compareTo que se utiliza para eso. Vamos a ver un ejemplo de cómo implementar un método que ordene cualquier matriz que implemente Comparable:

Como se puede ver podemos ordenar cualquier cosa que implemente Comparable. Y eso es exactamente lo que hace el método Arrays.sort. Así que podemos cambiar nuestro método ordenar por Arrays.sort y el programa funcionaría exactamente igual.

Comparator

¿Qué ocurre si, ahora que tenemos la clase Empleado preparada para ordenar por apellidos y nombre queremos otra ordenación para algún otro proceso?. La clase Arrays dispone de otra serie de métodos que permiten ordenar una clase en base a un objeto de otra clase (o la misma) que implemente el interfaz Comparator. El proceso sería el siguiente:

- Definir una clase que implemente Comparable
- Redefinir el método compare(T uno, T otro) para comparar dos objetos (de tipo genérico) que nos pasará la clase Arrays
- Crear un objeto de dicha clase y pasárselo al método Arrays.sort

Podríamos hacerlo de tres formas:

- Creando una clase y un objeto de la misma

```
class OrdenarEmplDep implements Comparator<Empleado> { public int compare(Empleado e1, Empleado e2){ return
e1.getDepartamento().compareTo(e2.getDepartamento()); } public static void main(String[] args){ Arrays.sort(new OrdenarEmplDep()); ... }}
```

- Creando un objeto de una clase anónima

```
public static void main(String[] args){ Arrays.sort(new Comparator<Empleado>(){ public int compare(Empleado e1, Empleado e2){ return
e1.getDepartamento().compareTo(e2.getDepartamento()); }
}); ... }
```

- Dado que es un interfaz funcional (sólo tiene un método) podemos emplear una expresión lambda:

```
public static void main(String[] args){ Arrays.sort((e1,e2) -> e1.getDepartamento().compareTo(e2.getDepartamento())); ... } Veamos un ejemplo:
```

3.17. Interfaz Set y SortedSet e implementación con clases HashSet y TreeSet

Set (java.util.Set)

La interface Set sirve para acceder a una colección sin elementos repetidos. La colección puede estar o no ordenada (con un orden natural o definido por el usuario, se entiende). La interface Set no declara ningún método adicional a los de Collection. Como un Set no admite elementos repetidos es importante saber cuándo dos objetos son considerados iguales. Para ello se dispone de los métodos equals() y hashCode(), que el usuario puede redefinir si lo desea. NOTA: Una colección que implemente Set no garantiza que los elementos se recuperen en el mismo orden en el que se han insertado. Simplemente garantiza que no hay dos elementos iguales (equals). Veamos un ejemplo de HashSet:

Interfaz SortedSet (java.util.SortedSet)

La interface SortedSet extiende la interface Set y gestiona una colección de elementos ordenada. Se añaden al interfaz Collection los siguientes métodos:

- public abstract Comparator comparator() : Retorna un objeto para realizar comparaciones. Si no se ha implementado el interfaz Comparable, este método retorna null.
- public abstract Object first() _ Retorna el primer objeto del set
- public abstract SortedSet headSet(Object) : Subconjunto hasta el objeto
- public abstract Object last() : Último elemento del conjunto
- public abstract SortedSet subSet(Object, Object) : Subconjunto entre ambos objetos
- public abstract SortedSet tailSet(Object) : Subconjunto desde el final hasta el objeto

El mismo ejemplo (TreeSet) asumiendo que la clase Empleado tiene un método compareTo basado en el numEmp:

3.18. Interfaz Map y SortedMap e implementación con clases HashMap y TreeMap

Interfaz Map (java.util.Map)

Un mapa es una estructura de datos agrupados en parejas clave/valor. Pueden ser considerados como una tabla de dos columnas. La clave debe ser única y se utiliza para acceder al valor. Aunque la interface Map no deriva de Collection, es posible acceder a los mapas como colecciones de claves, de valores o de parejas clave/valor. A continuación se muestran los métodos del interfaz :

- public abstract void clear() : Elimina los objetos del mapa
- public abstract boolean containsKey(Object) : true si contiene la clave indicada
- public abstract boolean containsValue(Object) : true si contiene el valor indicado
- public abstract Set entrySet() : Devuelve una vista del objeto Map como un Set. En esta vista se pueden modificar y eliminar elementos pero no añadir nuevos
- public abstract boolean equals(Object) : true si el objeto es igual al mapa
- public abstract Object get(Object) : Obtiene el valor asociado a la clave especificada
- public abstract int hashCode() : Retorna un código único para el mapa
- public abstract boolean isEmpty() : true si el mapa no contiene elementos
- public abstract Set keySet() : Devuelve una vista de las claves como un conjunto.
- public abstract Object put(Object, Object) : Añade una nueva pareja clave / valor
- public abstract void putAll(Map) : Añade todos los elementos del mapa
- public abstract Object remove(Object) : Elimina la primera aparición del objeto indicado
- public abstract int size() : Retorna el número de pares clave / valor del mapa
- public abstract Collection values() : Retorna una colección con los valores del mapa

La interfaz Map dispone de una subinterfaz denominada Entry que representa una pareja clave / valor y que dispone de los siguientes métodos:

- public abstract boolean equals(Object) : True si el objeto es igual a la entrada
- public abstract Object getKey() : Retorna la clave asociada a la entrada
- public abstract Object getValue() : Retorna el valor
- public abstract int hashCode() : Retorna el código
- public abstract Object setValue(Object) : Modifica el valor de la entrada. Opcional

Veamos un ejemplo con la clase HashMap que implementa Map:

Interfaz SortedMap (java.util.SortedMap)

Este interfaz deriva de Map y permite implementar mapas ordenados en función de la clave. Añade los siguientes métodos:

- public abstract Comparator comparator() : Retorna un objeto Comparator para realizar las comparaciones a menos que se haya implementado el interfaz Comparable, en cuyo caso retorna null.
- public abstract Object firstKey() : Retorna la primera clave del mapa
- public abstract SortedMap headMap(Object) : Retorna un mapa con los elementos que se encuentren hasta el objeto
- public abstract Object lastKey() : Retorna la última clave del mapa
- public abstract SortedMap subMap(Object, Object) : Retorna un mapa con los objetos que se encuentran entre los indicados
- public abstract java.util.SortedMap tailMap(Object) : Retorna un mapa con los objetos desde el indicado hasta el final

Modificamos el ejemplo para emplear un TreeMap: SortedMap empleados = new TreeMap<>(); Ordena por la clave (dni) y, dado que es de tipo String y que String implementa Comparable, la salida estará ordenada en base al dni:

Podemos modificar el comportamiento pasando un Comparable al constructor:

```
SortedMap empleados = new TreeMap<>((a,b) -> -a.compareTo(b));
```

Ahora la salida estará ordenada en descendente:

NOTA: La ordenación sólo se tiene en cuenta cuando accedemos a los elementos por la clave (métodos entrySet y keySet) pero no por los valores (values). Por eso, los listados que emplean este último método no aparecen ordenados.

3.19. Actividad 23 - Colecciones

Instrucciones

<https://classroom.github.com/a/xH5JXEEf>

Vamos a hacer un ejemplo utilizando alguna de las clases de colecciones vistas hasta ahora. Vamos a realizar una aplicación que permita gestionar las notas de unos alumnos de 1º DAW en una evaluación dada: Los pasos a dar son los siguientes:

- Se creará una clase Alumno con los siguientes campos (id se deberá incrementar para cada alumno):

id, apellidos, nombre

- También se creará la clase Modulo (el código es un String, por ejemplo PRO, IT, SI...)

codigo, nombre, profesor

- Y, por último, la clase Nota

id, idAlumno, codigoModulo, nota Las clases dispondrán de métodos set/get para cada propiedad, un constructor sin argumentos y otro que reciba todos los datos, un método toString y un método equals basado en id, código e id respectivamente. En el caso de Alumno y Nota los constructores deberán asignar un id correlativo. Además, se definirá una clase ejecutable llamada GestiónNotas que :

- Definirá una propiedad TreeSet para almacenar los alumnos ordenados por apellido
- Definirá una propiedad de tipo HashMap para almacenar todos los módulos. La clave será el código del módulo
- Definirá un ArrayList para almacenar todas las notas

NOTA: Todas estas propiedades deben ser estáticas para poder acceder a ellas desde main

- En el método main

Se llamará a un método cargarModulos que añadirá los datos de los módulos del curso al HashMap (poner los módulos de este año) Se llamará a un método cargarAlumnos que añadirá los datos de los alumnos al TreeSet. Si queréis podéis definir un método cargarNotas que añada algunas notas de ejemplo al ArrayList. Se mostrará el siguiente menú:

Gestión de Notas

1. Añadir Notas Cambiar Nota Ver Notas Alumno Ver Notas Módulo Salir

Opción [1-5]:

Añadir Notas

- Se pedirá el código del módulo del que se quieren añadir las notas
- Se comprobará si existe dicho módulo

De no ser así se dará un mensaje de error Si existe

Se recorrerán los alumnos uno por uno Para cada uno de ellos

Se pedirá la nota Se creará un objeto Nota con los datos necesarios Se añadirá el objeto a la lista de notas

- El proceso podrá repetirse a petición del usuario

Cambiar Notas

- Se pedirá el id del alumno y el código del módulo de la nota que se quiere modificar
- Se buscará la nota con dicha información

Si no existe, se dará un mensaje de error Si existe, se mostrará la nota actual, se pedirá la nueva nota y se almacenará

Ver Notas Alumno

- Se pedirá el id del alumno y se buscará el alumno en el set
- Si no existe se dará un mensaje de error
- Si existe

Se comprobará que tenga alguna nota mediante búsqueda desordenada Si no tiene notas se dará un mensaje informativo Si tiene notas se mostrará el siguiente listado:

Notas de apellidos, nombre Módulo Profesor Nota código-descripción profesor nota ... Nota Media : media

Ver Notas Módulo

- Se pedirá el código del modulo y se buscará en el mapa

- Si no existe se dará un mensaje de error
- Si existe

Se comprobará que tenga alguna nota mediante búsqueda desordenada. Si no tiene notas se dará un mensaje informativo. Si tiene notas se mostrará el siguiente listado (al final se mostrará el número y porcentaje de aprobados del curso):

Notas de código-descripción : profesor Alumno Nota apellidos, nombre nota ... N° Aprobados : cuenta (porc%)

Adicional

- A partir de los datos introducidos obtener un resumen de los resultados por módulo:

Resumen por Módulo Modulo Aprobados %Aprobados código-descripción cuenta porc% ...

% Media Aprobados : porc%

- Mostrar un listado con los alumnos que promocionarán. Ver los criterios de promoción y las horas por asignatura en la agenda.

3.20. Introducción a Streams Java 8

El API Streams

En la versión 8 de Java se introdujo el mayor cambio en el sistema de colecciones con la aparición del **API Streams** que contiene una serie de clases que permiten procesar secuencias de elementos a través de interfaces funcionales (y, por tanto, expresiones lambda). La clase principal es `Stream<T>` que dispone de métodos para buscar, contar, filtrar... elementos entre un conjunto de datos (stream).

Creación de un Stream

Para poder trabajar con corrientes de datos a partir de colecciones (también se pueden obtener dichas corrientes de otros sitios) necesitamos partir de un objeto `Stream`. Podemos crear un objeto de este tipo:

- Desde una matriz podemos emplear `Arrays.stream(matrix)`
- Desde un objeto de una clase que implemente `Collection<List, Map, SortedMap>` podemos emplear directamente `stream()`
- Si la clase es un `Map` o `HashMap` podemos convertir a stream las entradas, las claves o los valores (dado que todos ellos retornan colecciones)
- A partir de un conjunto de elementos podemos emplear `Stream.of(elementos)`

Operaciones intermedias y terminales

Sobre un stream podemos aplicar **dos tipos de operaciones**

- **intermedias:** realizan una operación sobre el stream y retornan un `Stream` del mismo tipo por lo que se pueden encadenar
- **terminales:** finalizan el stream y lo retornan como un objeto

Operaciones Terminales

Vamos a ver ejemplos de cada caso basados en la clase `Empleado`:

```
import java.util.Date;
import java.util.Objects;

/**
 * @author ichue
 */
public class Empleado {
    private String dni;
    private String nombre;
    private String apellidos;
    private String departamento;
    private double sueldo;
    private Date fechaAlta;

    public Empleado(String dni, String nombre, String apellidos, String departamento, double sueldo, Date fechaAlta) {
        this.dni = dni;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.departamento = departamento;
        this.sueldo = sueldo;
        this.fechaAlta = fechaAlta;
    }

    public Empleado() {
    }

    public String getDni() {
        return dni;
    }

    public void setDni(String dni) {
        this.dni = dni;
    }

    public String getNombre() {
        return nombre;
    }
```

```

    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getApellidos() {
        return apellidos;
    }

    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }

    public String getDepartamento() {
        return departamento;
    }

    public void setDepartamento(String departamento) {
        this.departamento = departamento;
    }

    public double getSueldo() {
        return sueldo;
    }

    public void setSueldo(double sueldo) {
        this.sueldo = sueldo;
    }

    public Date getFechaAlta() {
        return fechaAlta;
    }

    public void setFechaAlta(Date fechaAlta) {
        this.fechaAlta = fechaAlta;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 97 * hash + Objects.hashCode(this.dni);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Empleado other = (Empleado) obj;
        return Objects.equals(this.dni, other.dni);
    }

    @Override
    public String toString() {
        return "Empleado{" +
            "dni=" + dni + ", nombre=" + nombre + ", apellidos=" + apellidos + ", departamento=" + departamento + ", sueldo=" + sueldo + ", "
    }
}

```

Y la siguiente lista de empleados:

```

List<Empleado> empleados = new ArrayList<>();
empleados.add(new Empleado("55555555A", "Miguel", "Andrés", "Compras", 1234.45, new GregorianCalendar(2001, Calendar.JANUARY, 4).getTime()));
empleados.add(new Empleado("11111111A", "Juan", "López", "Compras", 1234.45, new GregorianCalendar(2000, Calendar.JANUARY, 15).getTime()));
empleados.add(new Empleado("44444444A", "Ana", "Sanz", "Administración", 1350.78, new GregorianCalendar(2000, Calendar.MARCH, 12).getTime()));
empleados.add(new Empleado("22222222A", "Carlos", "Ginés", "Producción", 1985.5, new GregorianCalendar(2000, Calendar.APRIL, 1).getTime()));
empleados.add(new Empleado("77777777A", "Luisa", "Jiménez", "Administración", 1350.78, new GregorianCalendar(2001, Calendar.JUNE, 4).getTime()));

```

- **count()** : Retorna el **número de elementos**

```
System.out.println(empleados.stream().count());
```

5

- **anyMatch(lambda)** : Retorna **true** si **alguno de los elementos cumple la condición**

```
System.out.println(empleados.stream().anyMatch(e -> e.getDepartamento().equals("Compras")));
```

true

- **allMatch(lambda)** : Retorna **true** si **todos los elementos cumplen la condición**

```
System.out.println(empleados.stream().anyMatch(e -> e.getDepartamento().equals("Compras")));
```

false

- noneMatch(lambda) : Retorna true si ningún elemento cumple la condición

```
System.out.println(empleados.stream().anyMatch(e -> e.getDepartamento().equals("Compras")));
false
```

- collect(Collector) : **Retorna un resultado a partir del stream.** Podemos emplear alguno de los métodos de la clase de utilidad Collectors:

Método	Retorna
Collectors.toList()	List
Collectors.toSet()	Set
Collectors.toMap(lambda)	Map : la expresión lambda tiene dos partes: la primera debe retornar la clave y la segunda el valor
Collectors.averagingDouble(lambda)	Double : el promedio de lo que retorna la expresión lambda
Collectors.averagingInt(lambda)	Integer : el promedio de lo que retorna la expresión lambda sin decimales
Collectors.averagingLong(lambda)	Long : el promedio de lo que retorna la expresión lambda sin decimales
Collectors.counting()	Integer : retorna el número de elementos del Stream
Collectors.summingDouble(lambda)	Double : la suma de lo que retorna la expresión lambda
Collectors.summingInt(lambda)	Integer : la suma de lo que retorna la expresión lambda sin decimales
Collectors.summarizingDouble(lambda)	DoubleSummaryStatistics: un objeto que incluye estadísticas sobre lo que retorna la expresión lambda
Collectors.summarizingInt(lambda)	IntSummaryStatistics: un objeto que incluye estadísticas sobre lo que retorna la expresión lambda
Collectors.summarizingLong(lambda)	LongSummaryStatistics: un objeto que incluye estadísticas sobre lo que retorna la expresión lambda
Collectors.maxBy(lambda)	Optional<Empleado> : Retorna el máximo elemento de la colección teniendo en cuenta la expresión lambda (comparador)
Collectors.minBy(lambda)	Optional<Empleado> : Retorna el mínimo elemento de la colección teniendo en cuenta la expresión lambda (comparador)
Collectors.groupingBy(lambda[, lambda2])	Map<Object, Object> : Retorna un Map cuya clave es el campo retornado por la expresión lambda y cuyo valor es una lista con todos los elementos que tienen dicho campo. Opcionalmente se le puede pasar una función de totalización (counting(), averaging(lambda) ...)
Collectors.mapping(lambda, collector)	Object : Convierte los datos a otros en función de la expresión lambda antes de aplicar el collector
Collectors.joining([sep])	String : Retorna una cadena con la concatenación de todos los elementos separados por el separador indicado. La colección debe ser de caracteres o de cadenas
Collector.partitioning(lambda)	Map<Boolean, Empleado> : Retorna un mapa con dos claves, los que cumplen y los que no cumplen la expresión lambda. El valor es una lista de objetos
Collector.reducing(lambda)	Optional<Object> : Retorna un único elemento combinado después de aplicar una función lambda a todos los elementos de la colección. Dicha función tendrá dos parámetros : el elemento combinado y el elemento de la colección (ambos del tipo de la colección). En cada paso se asignará al elemento combinado el resultado retornado por dicha función

```
System.out.println("-- toList");
System.out.println(empleados.stream().collect(Collectors.toList()).get(0));
System.out.println("-- toSet");
System.out.println(empleados.stream().collect(Collectors.toSet()).iterator().next());
System.out.println("-- toMap");
System.out.println(empleados.stream().collect(Collectors.toMap(e -> e.getDni(), e -> e.getApellidos() + ", " + e.getNombre())));
System.out.println("-- averagingDouble");
System.out.println(empleados.stream().collect(Collectors.averagingDouble(e -> e.getSueldo())));
System.out.println("-- counting");
System.out.println(empleados.stream().collect(Collectors.counting()));
System.out.println("-- summingInt");
System.out.println(empleados.stream().collect(Collectors.summingInt(e -> (int) (e.getSueldo()))));
System.out.println("-- summarizingDouble");
System.out.println(empleados.stream().collect(Collectors.summarizingDouble(e -> (int) (e.getSueldo()))));
DoubleSummaryStatistics stats = empleados.stream().collect(Collectors.summarizingDouble(e -> (int) (e.getSueldo())));
System.out.println(stats.getAverage());
System.out.println("-- maxBy minBy");
System.out.println(empleados.stream().collect(Collectors.maxBy((e1, e2) -> (int) (e1.getSueldo() - e2.getSueldo())).get()));
System.out.println(empleados.stream().collect(Collectors.minBy((e1, e2) -> (int) (e1.getSueldo() - e2.getSueldo())).get()));
System.out.println(empleados.stream().collect(Collectors.maxBy((e1, e2) -> e1.getDni().compareTo(e2.getDni()))).get());
System.out.println(empleados.stream().collect(Collectors.minBy((e1, e2) -> e1.getDni().compareTo(e2.getDni()))).get());
System.out.println("-- groupingBy");
Map<String, List<Empleado>> resultados = empleados.stream().collect(Collectors.groupingBy(e -> e.getDepartamento()));
System.out.println(resultados);
System.out.println(resultados.get("Compras"));
System.out.println(resultados.get("Administración").stream().collect(Collectors.summarizingDouble(e -> e.getSueldo())));
System.out.println("-- groupingBy con totales");
Map<String, DoubleSummaryStatistics> deps = empleados.stream()
    .collect(Collectors.groupingBy(Empleado::getDepartamento, Collectors.summarizingDouble(e -> e.getSueldo())));
deps.forEach( (k,v) -> {
    System.out.println(k+":"+v);
});
System.out.println("-- mapping");
System.out.println(empleados.stream().collect(Collectors.mapping(e -> e.getApellidos()+" "+e.getNombre(), Collectors.toList())));
```

```

System.out.println("-- joining");
System.out.println(Stream.of("Juan","Luis","Ana","Carlos").collect(Collectors.joining()));
System.out.println(Stream.of("Juan","Luis","Ana","Carlos").collect(Collectors.joining("*--*")));
Map<Boolean, List<Empleado>> mapa = empleados.stream().collect(Collectors.partitioningBy(e -> e.getSueldo()>1500));
System.out.println(mapa.get(true).stream().count());
System.out.println(mapa.get(false).stream().count());
System.out.println("-- reducing");
System.out.println(empleados.stream().map(e->e.getSueldo()).collect(Collectors.reducing((total,s) -> s+=total)));
-- toList
Empleado{dni=55555555A, nombre=Miguel, apellidos=Andrés, departamento=Compras, sueldo=1234.45, fechaAlta=Thu Jan 04 00:00:00 CET 2001}
-- toSet
Empleado{dni=55555555A, nombre=Miguel, apellidos=Andrés, departamento=Compras, sueldo=1234.45, fechaAlta=Thu Jan 04 00:00:00 CET 2001}
-- toMap
{77777777A=Jiménez, Luisa, 55555555A=Andrés, Miguel, 4444444A=Sanz, Ana, 11111111A=López, Juan, 22222222A=Ginés, Carlos}
-- averagingDouble
1431.192
-- counting
5
-- summingInt
7153
-- summarizingDouble
DoubleSummaryStatistics{count=5, sum=7153,000000, min=1234,000000, average=1430,600000, max=1985,000000}
1430.6
-- maxBy minBy
Empleado{dni=22222222A, nombre=Carlos, apellidos=Ginés, departamento=Producción, sueldo=1985.5, fechaAlta=Sat Apr 01 00:00:00 CEST 2000}
Empleado{dni=55555555A, nombre=Miguel, apellidos=Andrés, departamento=Compras, sueldo=1234.45, fechaAlta=Thu Jan 04 00:00:00 CET 2001}
Empleado{dni=77777777A, nombre=Luisa, apellidos=Jiménez, departamento=Administración, sueldo=1350.78, fechaAlta=Mon Jun 04 00:00:00 CEST 2001}
Empleado{dni=11111111A, nombre=Juan, apellidos=López, departamento=Compras, sueldo=1234.45, fechaAlta=Sat Jan 15 00:00:00 CET 2000}
-- groupingBy
{Producción=[Empleado{dni=22222222A, nombre=Carlos, apellidos=Ginés, departamento=Producción, sueldo=1985.5, fechaAlta=Sat Apr 01 00:00:00 CEST 2000}], Compras=[Empleado{dni=55555555A, nombre=Miguel, apellidos=Andrés, departamento=Compras, sueldo=1234.45, fechaAlta=Thu Jan 04 00:00:00 CET 2001}, Empleado{dni=11111111A, nombre=Juan, apellidos=López, departamento=Compras, sueldo=1234.45, fechaAlta=Sat Jan 15 00:00:00 CET 2000}]
-- groupingBy con totales
Producción:DoubleSummaryStatistics{count=1, sum=1985,500000, min=1985,500000, average=1985,500000, max=1985,500000}
Compras:DoubleSummaryStatistics{count=2, sum=2468,900000, min=1234,450000, average=1234,450000, max=1234,450000}
Administración:DoubleSummaryStatistics{count=2, sum=2701,560000, min=1350,780000, average=1350,780000, max=1350,780000}
-- mapping
[Andrés, Miguel, López, Juan, Sanz, Ana, Ginés, Carlos, Jiménez, Luisa]
-- joining
JuanLuisAnaCarlos
Juan---*Luis---*Ana---*Carlos
-- partitioningBy
1
4
-- reducing
Optional[7155.96]

• findAny() : Retorna un elemento del stream como un Optional

System.out.println(empleados.stream().findAny().get());

Empleado{dni=55555555A, apellidos=Miguel, nombre=Andrés, departamento=Compras, sueldo=1234.45, fechaAlta=Thu Jan 04 00:00:00 CET 2001}

• findFirst() : Retorna el primer elemento del stream

System.out.println(empleados.stream().findFirst().get());

Empleado{dni=55555555A, apellidos=Miguel, nombre=Andrés, departamento=Compras, sueldo=1234.45, fechaAlta=Thu Jan 04 00:00:00 CET 2001}

• 'forEach(lambda) : Realiza una tarea con cada elemento del stream.

A partir de Java 8 también se incluye en el interfaz Iterable, así que está en todas las colecciones.

empleados.stream().forEach(System.out::println);

Empleado{dni=55555555A, apellidos=Miguel, nombre=Andrés, departamento=Compras, sueldo=1234.45, fechaAlta=Thu Jan 04 00:00:00 CET 2001}
Empleado{dni=11111111A, apellidos=Juan, nombre=López, departamento=Compras, sueldo=1234.45, fechaAlta=Sat Jan 15 00:00:00 CET 2000}
Empleado{dni=4444444A, apellidos=Ana, nombre=Sanz, departamento=Administración, sueldo=1350.78, fechaAlta=Sun Mar 12 00:00:00 CET 2000}
Empleado{dni=22222222A, apellidos=Carlos, nombre=Ginés, departamento=Producción, sueldo=1985.5, fechaAlta=Sat Apr 01 00:00:00 CEST 2000}
Empleado{dni=77777777A, apellidos=Luisa, nombre=Jiménez, departamento=Administración, sueldo=1350.78, fechaAlta=Mon Jun 04 00:00:00 CEST 2001}

• min(lambda) : Retorna el menor elemento de la colección en base al comparador definido

System.out.println(empleados.stream().min((e1,e2) -> (int) (e1.getSueldo()-e2.getSueldo())));

Optional[Empleado{dni=55555555A, apellidos=Miguel, nombre=Andrés, departamento=Compras, sueldo=1234.45, fechaAlta=Thu Jan 04 00:00:00 CET 2001}]

• max(lambda) : Retorna el mayor elemento de la colección en base al comparador definido

System.out.println(empleados.stream().max((e1,e2) -> (int) (e1.getSueldo()-e2.getSueldo())));

Optional[Empleado{dni=22222222A, apellidos=Ginés, nombre=Carlos, departamento=Producción, sueldo=1985.5, fechaAlta=Sat Apr 01 00:00:00 CEST 2000}]

• reduce(lambda) : Lo mismo que Collections.reducing:

System.out.println(empleados.stream().map(e->e.getSueldo()).reduce((total,s) -> s+=total));

Optional[7155.96]

• toArray() : Retorna el stream como una matriz

System.out.println(empleados.stream().map(Empleado::getApellidos).toArray()[0]);
Miguel

```

Operaciones No Terminales

- `filter(lambda)` : Retorna un **stream con los elementos que cumplen la condición**

```
System.out.println(empleados.stream().filter(e -> e.getSueldo()<1500).count());  
4
```

- `map(lambda)` : Retorna un **stream al que se le ha aplicado la transformación indicada en cada elemento**

```
System.out.println(empleados.stream().filter(e -> e.getSueldo()<1500).map(Empleado::getDni).collect(Collectors.toList()));  
[55555555A, 11111111A, 44444444A, 77777777A]
```

- `distinct()` : Retorna un **stream sin elementos repetidos (equals)**

```
System.out.println(empleados.stream().map(Empleado::getDepartamento).distinct().collect(Collectors.toList()));  
[Compras, Administración, Producción]
```

- `limit(n)` : Retorna un **stream con los n primeros elementos**

```
System.out.println(empleados.stream().limit(2).map(e -> e.getApellidos()+" "+e.getNombre()).collect(Collectors.toList()));  
[Andrés Miguel, López Juan]
```

- `skip(n)` : **Salta n elementos**

```
System.out.println(empleados.stream().skip(2).map(e -> e.getApellidos()+" "+e.getNombre()).collect(Collectors.toList()));  
[Sanz Ana, Ginés Carlos, Jiménez Luisa]
```

- `peek(lambda)` : **Permite ejecutar una función lambda sobre el elemento actualmente procesado sin quitarlo del stream.** Es muy útil para depurar el funcionamiento de las operaciones realizadas

```
System.out.println(empleados.stream()  
.peek(e -> { System.out.println(e); })  
.filter(e -> e.getDepartamento().equals("Administración"))  
.peek(e -> { System.out.println("F: " + e); })  
.map(Empleado::getApellidos)  
.peek(e -> { System.out.println("M: " + e); })  
.collect(Collectors.toList());  
  
Empleado{dni=55555555A, nombre=Miguel, apellidos=Andrés, departamento=Compras, sueldo=1234.45, fechaAlta=Thu Jan 04 00:00:00 CET 2001}  
Empleado{dni=11111111A, nombre=Juan, apellidos=López, departamento=Compras, sueldo=1234.45, fechaAlta=Sat Jan 15 00:00:00 CET 2000}  
Empleado{dni=44444444A, nombre=Ana, apellidos=Sanz, departamento=Administración, sueldo=1350.78, fechaAlta=Sun Mar 12 00:00:00 CET 2000}  
M: Sanz  
Empleado{dni=22222222A, nombre=Carlos, apellidos=Ginés, departamento=Producción, sueldo=1985.5, fechaAlta=Sat Apr 01 00:00:00 CEST 2000}  
Empleado{dni=77777777A, nombre=Luisa, apellidos=Jiménez, departamento=Administración, sueldo=1350.78, fechaAlta=Mon Jun 04 00:00:00 CEST 2001}  
M: Jiménez  
[Sanz, Jiménez]
```

- `sorted(lambda)` : Retorna el **stream ordenado en base al comparador**

```
System.out.println(empleados.stream().sorted((e1,e2)-> (int)(e2.getSueldo()-e1.getSueldo())).map(e->e.getApellidos()+" "+e.getSueldo()).collect(Collectors.toList()));  
[Ginés:1985.5, Sanz:1350.78, Jiménez:1350.78, Andrés:1234.45, López:1234.45]
```

Como ya hemos visto, las operaciones no terminales se pueden encadenar.

3.21. Actividad 24 - Streams

Instrucciones

[Enlace al Classroom](#)

Se dispone de **información sobre los alquileres de inmuebles en España en el año 2022**. Esta información se almacenará en objetos de una clase `Alquiler` con los siguientes campos:

- `private int id;`
- `private String provincia;`
- `private String ccaa;`
- `private String descripcion;`
- `private double precio;`
- `private int habitaciones;`
- `private int metros;`

Se proporciona el proyecto inicial en el que ya está definido un DAO que permite acceder a la información de los alquileres. A partir de este proyecto se pide responder a las siguientes preguntas empleando Streams:

- ¿ Cuántos alquileres hay en la lista ?
- ¿ Hay algún alquiler en Vizcaya ?
- ¿ Cuál es el precio medio del alquiler en España ?
- ¿ Cuál es el alquiler más bajo ? ¿ Y el más caro ?

- ¿ Y en Vizcaya ?
- ¿ Cuántos alquileres hay de pisos de menos de 2 habitaciones ?
- Obtener una lista con todas las provincias que hay en la lista
- ¿ Cuántas comunidades autónomas hay ?
- Obtener un mapa que tenga como clave la Comunidad Autónoma y como valores, los totales relativos a las habitaciones de los alquileres en esa comunidad.
- ¿ Cuántos alquileres hay de pisos con más de 2 habitaciones, más de 80 metros y precio inferior a 800€ ? ¿ Cuáles de ellos están en Vizcaya ?
- ¿ Cuáles son los 5 alquileres más caros de Vizcaya ? ¿ Y los 5 más baratos?

4. Corrientes de Datos

4.1. Introducción

En múltiples ocasiones un programa necesita enviar información a o recibir información de un destino externo. Esta información puede estar en cualquier sitio: en un fichero, en una red, en memoria o en otro programa. Además esta información puede ser de múltiples tipos : objetos, caracteres, imágenes o sonidos. **Para manipular la transferencia de información desde o hacia un programa Java se emplea lo que se denomina corrientes de datos.** El proceso para **leer datos** desde un origen de datos (un fichero, memoria, red...) sería el siguiente:

- Se **abre** en el objeto origen **una corriente de datos** de entrada (`input stream`)
- Mientras **haya más datos**
- Se **lee** la información y se procesa
- Se **cierra** la corriente de datos

El proceso para **escribir datos** en un destino de datos:

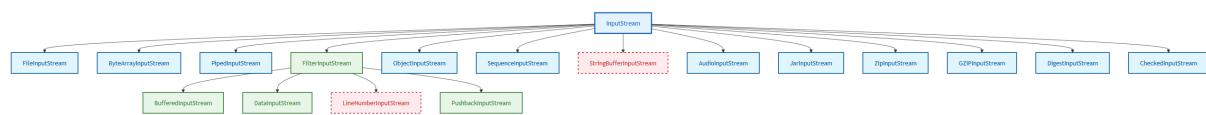
- Se **abre** una corriente de datos de **salida** (`output stream`)
- Mientras **haya más datos que escribir**
- Se **escribe** la información
- Se **cierra** la corriente de datos

Todas las clases que manipulan corrientes de datos se encuentran en el paquete `java.io`. Estas clases se agrupan en **dos grandes bloques**:

- clases que **manipulan octetos**
- clases que **manipulan caracteres** (en Java los caracteres son Unicode y ocupan dos octetos)

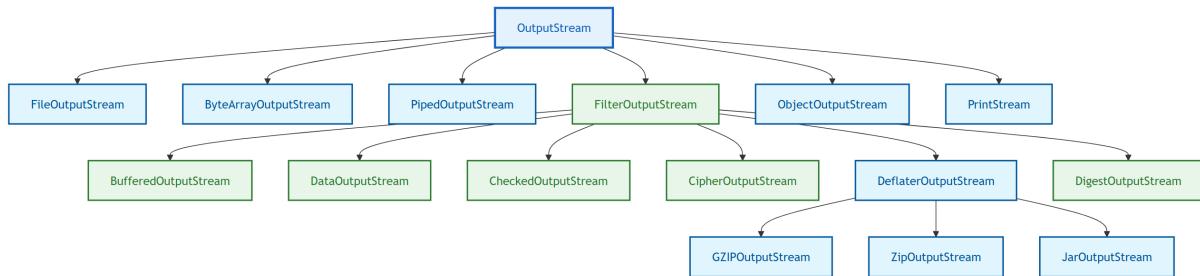
Corrientes de Datos de Octetos

Las corrientes de datos a nivel de octeto son útiles para **transferir información binaria**, como por ejemplo objetos, imágenes, sonidos, etc. Todas las clases derivan de las clases **abstractas** `InputStream` (para **lectura** de datos) y `OutputStream` (para **escritura**). Las **subclases** de estas clases realizan tareas más específicas. El conjunto de clases es el siguiente :



Aquí tienes una tabla con las principales clases de la jerarquía de `InputStream` y una breve descripción de su uso:

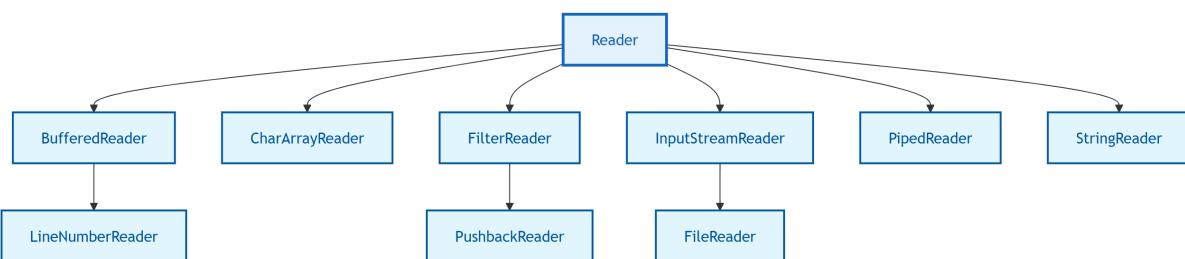
Clase	Descripción
<code>InputStream</code>	Clase abstracta base para todos los flujos de entrada de bytes
<code>FileInputStream</code>	Lee bytes desde un archivo en el sistema de archivos
<code>ByteArrayInputStream</code>	Lee bytes desde un array de bytes en memoria
<code>PipedInputStream</code>	Lee datos de un <code>PipedOutputStream</code> , útil para comunicación entre hilos
<code>FilterInputStream</code>	Clase base para filtros que procesan otros flujos de entrada
<code>BufferedInputStream</code>	Añade buffer a otro <code>InputStream</code> para mejorar el rendimiento
<code>DataInputStream</code>	Lee tipos de datos primitivos de Java desde un flujo de entrada
<code>LineNumberInputStream</code>	(Obsoleta) Mantiene el seguimiento de números de línea
<code>PushbackInputStream</code>	Permite "devolver" bytes al flujo para ser releídos
<code>ObjectInputStream</code>	Lee objetos Java serializados desde un flujo
<code>SequenceInputStream</code>	Concatena múltiples flujos de entrada en uno solo
<code>StringBufferInputStream</code>	(Obsoleta) Lee bytes desde un <code>String</code>
<code>AudioInputStream</code>	Lee datos de audio desde un flujo
<code>JarInputStream</code>	Lee archivos JAR
<code>ZipInputStream</code>	Lee archivos ZIP
<code>GZIPInputStream</code>	Lee archivos comprimidos con formato GZIP
<code>DigestInputStream</code>	Calcula un resumen de mensaje (<code>hash</code>) mientras lee datos
<code>CheckedInputStream</code>	Calcula un checksum mientras lee datos



Clase	Descripción
OutputStream	Clase abstracta base para todos los flujos de salida de bytes
FileOutputStream	Escribe bytes a un archivo en el sistema de archivos
ByteArrayOutputStream	Escribe datos a un array de bytes en memoria
PipedOutputStream	Escribe datos a un <code>PipedInputStream</code> , útil para comunicación entre hilos
FilterOutputStream	Clase base para filtros que procesan otros flujos de salida
BufferedOutputStream	Añade buffer a otro <code>OutputStream</code> para mejorar el rendimiento
DataOutputStream	Escribe tipos de datos primitivos de Java a un flujo de salida
CheckedOutputStream	Calcula un checksum mientras escribe datos
CipherOutputStream	Cifra datos mientras son escritos al flujo
DeflaterOutputStream	Comprime datos mientras son escritos al flujo
DigestOutputStream	Calcula un resumen de mensaje (<code>hash</code>) mientras escribe datos
GZIPOutputStream	Escribe datos comprimidos en formato GZIP
ZipOutputStream	Escribe datos en formato ZIP
JarOutputStream	Escribe datos en formato JAR (extensión de ZIP)
ObjectOutputStream	Serializa objetos Java a un flujo de salida
PrintStream	Escribe representaciones textuales de objetos Java (usado por <code>System.out</code>)

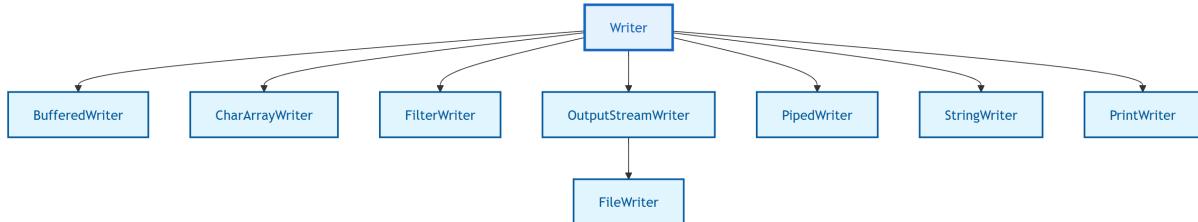
Corrientes de Datos de Caracteres

Las corrientes de datos a nivel de carácter permiten transferir información textual. Derivan de las clases abstractas Reader (para corrientes de entrada) y Writer (para corrientes de salida). Las subclases son las siguientes :



Clase	Descripción
Reader	Clase abstracta base para todos los lectores de caracteres
BufferedReader	Añade buffer a otro Reader para mejorar el rendimiento de lectura
LineNumberReader	Extiende <code>BufferedReader</code> para mantener un seguimiento de números de línea
CharArrayReader	Lee caracteres desde un array de caracteres en memoria
FilterReader	Clase base para lectores que filtran otros lectores
PushbackReader	Permite "devolver" caracteres al flujo para ser releídos
InputStreamReader	Puente entre flujos de bytes y flujos de caracteres (convierte bytes a caracteres)

Clase	Descripción
FileReader	Lee caracteres desde archivos usando la codificación predeterminada del sistema
PipedReader	Lee caracteres desde un <code>PipedWriter</code> , útil para comunicación entre hilos
StringReader	Lee caracteres desde un <code>String</code> en memoria



Clase	Descripción
Writer	Clase abstracta base para todos los escritores de caracteres
BufferedWriter	Añade buffer a otro <code>Writer</code> para mejorar el rendimiento de escritura
CharArrayWriter	Escribe caracteres a un array de caracteres en memoria
FilterWriter	Clase base para escritores que filtran otros escritores
OutputStreamWriter	Puente entre flujos de caracteres y flujos de bytes (convierte caracteres a bytes)
FileWriter	Escribe caracteres a archivos usando la codificación predeterminada del sistema
PipedWriter	Escribe caracteres a un <code>PipedReader</code> , útil para comunicación entre hilos
StringWriter	Escribe caracteres a un <code>StringBuffer</code> en memoria
PrintWriter	Proporciona métodos para imprimir representaciones formateadas de objetos en un flujo de texto

abstract class InputStream ([java.io.InputStream](#))

Dado que esta es la clase abstracta de la que derivan todas las clases de lectura de octetos desde una corriente de datos vamos a ver cuáles son sus principales métodos.

- `int available()` : Número de octetos que se pueden leer antes de la siguiente lectura de datos
- `void close()` : Cierra la corriente de datos. Puede lanzar `IOException`
- `void mark(int marca)` : Pone una marca en el fichero para retornar a ella posteriormente (ver `reset()`). Puede lanzar una `IOException` si no se puede hacer la operación.
- `boolean markSupported()` : Indica si se soporta el uso de marcas
- `abstract int read()` : Lee un octeto y retorna su código. Si estamos al final del fichero retorna un `-1`. Si no se puede leer el octeto por otras razones, se lanza una excepción `IOException`.
- `int read(byte[] b)` : Lee octetos de la corriente y los almacena en la matriz `b`. Se retorna el número de octetos leídos o un `-1` si se ha alcanzado el final de la corriente de datos. Se lanza una `IOException` si no se pueden leer los datos.
- `int read(byte[] b, int pos, int num)` : Lee hasta `num` octetos de la corriente de datos y los almacena a partir de la posición `pos` de la matriz `b`. Se retorna el número de octetos leídos o un `-1`. Se lanza una `IOException` si no se pueden leer los datos.
- `void reset()` : Se restituye la corriente de datos en la última marca establecida
- `long skip(long n)` : Se saltan el número de octetos especificados.

abstract class Reader ([java.io.Reader](#))

Clase abstracta para la lectura de caracteres.

- `abstract void close()` : Cierra la corriente de datos. Puede lanzar `IOException`
- `void mark(int adelante)` : Pone una marca en la posición actual. Esta marca es válida durante el número de octetos indicados en adelante. Puede lanzar una `IOException`.
- `boolean markSupported()` : Indica si se soporta el uso de marcas
- `int read()` : Lee un carácter y retorna su código. Si estamos al final del fichero retorna un `-1`. Si no se puede leer el carácter por otras razones, se lanza una excepción `IOException`.
- `int read(char[] c)` : Lee caracteres de la corriente y los almacena en la matriz `c`. Se retorna el número de caracteres leídos o un `-1`. Se lanza una `IOException` si no se pueden leer los datos.
- `abstract int read(char[] c, int pos, int num)` : Lee hasta `num` caracteres de la corriente de datos y los almacena a partir de la posición `pos` de la matriz `c`. Se

retorna el número de caracteres leídos o un -1. Se lanza una `IOException` si no se pueden leer los datos.

- `boolean ready()` : Indica si la corriente está preparada para leer datos
- `void reset()` : Se restituye la corriente de datos en la última marca establecida
- `long skip(long n)` : Se saltan el número de caracteres especificados.

abstract class OutputStream (java.io.OutputStream)

Clase abstracta para la escritura de octetos

- `void close()` : Cierra la corriente de datos
- `void flush()` : Escribe todos los octetos pendientes. Es importante ejecutarla antes de cerrar la corriente para asegurarnos de que no quedan datos pendientes de guardar (sobre todo cuando se emplean buffers)
- `void write(byte[] b)` : Escribe los octetos de la matriz `b` en la corriente
- `void write(byte[] b, int pos, int num)` : Escribe `num` octetos desde la posición `pos` de la matriz `b` en la corriente
- `void write(int b)` : Escribe el octeto `b` en la corriente

Todas las operaciones pueden lanzar una `IOException`.

abstract class Writer (java.io.Writer)

Clase abstracta para la escritura de caracteres

- `void close()` : Cierra la corriente de datos
- `void flush()` : Escribe todos los caracteres pendientes
- `void write(char[] c)` : Escribe los caracteres de la matriz `c` en la corriente
- `void write(char[] c, int pos, int num)` : Escribe `num` caracteres desde la posición `pos` de la matriz `c` en la corriente
- `void write(int c)` : Escribe el carácter `c` en la corriente
- `void write(String s)` : Escribe los caracteres de la cadena `s` en la corriente
- `void write(String s, int pos, int num)` : Escribe `num` caracteres desde la posición `pos` de la cadena `s` en la corriente

Todas las operaciones pueden lanzar una `IOException`.

4.2. Corrientes de Datos con Ficheros FileStreams Clases File y JFileChooser

Clase File

La clase `java.io.File` representa un fichero (o directorio). Dispone de los siguientes métodos:

Método	Descripción
<code>File(String) / File(File, String) / File(String, String) / File(URI)</code>	Crea un nuevo objeto que representa el fichero/directorio especificado como un camino absoluto, relativo al padre o como un URI
<code>boolean canRead() / boolean canWrite()</code>	Devuelve <code>true</code> si el fichero/directorio puede ser leído/escrito
<code>boolean createNewFile()</code>	Crea un nuevo fichero si no existe ya uno con el mismo nombre
<code>File createTempFile(String prefijo, String sufijo)</code>	Crea un fichero temporal con el prefijo y sufijo indicado asegurándose que el fichero no exista
<code>boolean delete()</code>	Elimina el fichero o directorio al que representa el objeto
<code>void deleteOnExit()</code>	El fichero/directorio será eliminado al finalizar la ejecución de la máquina virtual de Java
<code>boolean exists()</code>	Devuelve <code>true</code> si el fichero existe
<code>File getAbsoluteFile()</code>	Retorna la trayectoria absoluta al fichero como un objeto <code>File</code>
<code>String getAbsolutePath()</code>	Retorna la trayectoria absoluta al fichero como una cadena
<code>File getCanonicalFile() / String getCanonicalPath()</code>	Retorna el nombre canónico como un objeto o como una cadena (el nombre canónico representa una trayectoria única al fichero)
<code>String getName()</code>	Retorna el nombre del fichero/directorio
<code>String getParent() / File getParentFile()</code>	Retorna el padre del fichero/directorio como una cadena o como un objeto
<code>String getPath()</code>	Retorna una cadena con el camino al fichero/directorio
<code>boolean isAbsolute()</code>	Devuelve <code>true</code> si el objeto contiene una referencia absoluta
<code>boolean isDirectory() / boolean isFile()</code>	Devuelve <code>true</code> si es un fichero/directorio

Método	Descripción
boolean isHidden()	Devuelve true si es un fichero oculto
long lastModified() / boolean setLastModified(long)	Retorna/establece la fecha de última modificación como un long
long length()	Tamaño del fichero (0 en directorios)
String[] list() / String[] list(FilenameFilter) / File[] listFiles() / File[] listFiles(FilenameFilter) / File[] listFiles(FileFilter)	Lista de ficheros y directorios contenidos en el directorio representado por el objeto. Adicionalmente se puede especificar un filtro (ver FileFilter)
static File[] listRoots()	Retorna todas las raíces del sistema de ficheros
boolean mkdir() / boolean mkdirs()	Crea el directorio indicado (en el segundo caso creando cualquier directorio adicional necesario). Retorna true si la creación ha tenido éxito
boolean renameTo(File nuevo)	Renombra el fichero al nuevo
boolean setReadOnly()	Convierte el fichero a sólo lectura retornando true si la operación ha tenido éxito
String toString()	Retorna la trayectoria absoluta al objeto
URI toURI() / URL toURL()	Retorna la trayectoria al fichero/directorio como un URI o un URL

Vemos la información obtenida sobre la forma de acceder a un fichero (directorio):

```
package org.zabalburu.ficheros;

import java.io.File;
import java.io.IOException;

public class Ficheros {

    public static void main(String[] args) throws IOException {
        File f = new File("Fichero.java");
        System.out.printf("%-20s%-40s%n", "getAbsoluteFile()", f.getAbsoluteFile());
        System.out.printf("%-20s%-40s%n", "getAbsolutePath()", f.getAbsolutePath());
        System.out.printf("%-20s%-40s%n", "getCanonicalFile()", f.getCanonicalFile());
        System.out.printf("%-20s%-40s%n", "getCanonicalPath()", f.getCanonicalPath());
        System.out.printf("%-20s%-40s%n", "getName()", f.getName());
        System.out.printf("%-20s%-40s%n", "getPath()", f.getPath());
    }
}

getAbsoluteFile()    D:\ichueca\Ficheros\Fichero.java
getAbsolutePath()   D:\ichueca\Ficheros\Fichero.java
getCanonicalFile()  D:\ichueca\Ficheros\Fichero.java
getCanonicalPath() D:\ichueca\Ficheros\Fichero.java
getName()          Fichero.java
getPath()          Fichero.java
```

Veamos otro ejemplo que nos permita mostrar toda la estructura de ficheros y directorios de una carpeta de nuestro equipo:

```
package org.zabalburu.ficheros;

import java.io.File;
import java.io.IOException;
import java.util.Date;

public class Ficheros {

    public static void main(String[] args) throws IOException {
        File directorio = new File("../ficheros");
        System.out.println(directorio.getAbsoluteFile());
        int nivel = 0;
        listar(directorio, nivel);
    }

    private static String getEspacios(int nivel){
        String espacios = "";
        for(int i=0;i<nivel;i++){
            espacios += " ";
        }
        espacios = espacios + ((nivel>0)?"-":":");
        return espacios;
    }

    private static void listar(File directorio, int nivel) {

        System.out.println(getEspacios(nivel) + directorio.getName());
        File[] contenido = directorio.listFiles();
        if (contenido != null){
            for(File f : contenido){
                if (f.isDirectory()){
                    if (!f.isHidden()){
                        nivel++;
                        listar(f,nivel);
                        nivel--;
                    }
                }
            }
        }
    }
}
```

La salida:

```
ficheros
 |--pom.xml                               03/07/25 18:24:16    762
 |--src
   |--main
     |--java
       |--org
         |--zabalburu
           |--ficheros
             |--Fichero.java                03/07/25 18:30:11    925
             |--Ficheros.java              03/07/25 18:27:30    1.424
   |--test
     |--java
   |--target
     |--classes
       |--.netbeans_automatic_build    03/07/25 18:30:40    0
     |--org
       |--zabalburu
         |--ficheros
           |--Fichero.class            03/07/25 18:30:11    1.275
           |--Ficheros.class          03/07/25 18:28:45    2.516
   |--Ficheros-1.0-SNAPSHOT.jar            03/07/25 18:28:45    3.393
   |--generated-sources
     |--annotations
   |--maven-archiver
     |--pom.properties               03/07/25 18:28:45    118
   |--maven-status
     |--maven-compiler-plugin
       |--compile
         |--default-compile
           |--createdFiles.lst        03/07/25 18:28:45    38
           |--inputFiles.lst          03/07/25 18:28:45    123
       |--testCompile
         |--default-testCompile
           |--inputFiles.lst          03/07/25 18:28:45    0
   |--test-classes
     |--.netbeans_automatic_build    03/07/25 18:30:40    0
```

Clase JFileChooser

La clase `JFileChooser` permite mostrar un cuadro de diálogo para que el usuario **seleccione ficheros y/o directorios**.

Método	Descripción
<code>JFileChooser() / JFileChooser(File) / JFileChooser(String)</code>	Crea un nuevo selector de ficheros en el que opcionalmente se indica el directorio inicial
<code>boolean accept(File)</code>	Devuelve <code>true</code> si el fichero puede ser aceptado
<code>void addActionListener(ActionListener) / void removeActionListener(ActionListener)</code>	Añade/elimina receptores de eventos
<code>void changeToParentDirectory()</code>	Pasa al directorio padre del actual
<code>FileFilter getAcceptAllFileFilter()</code>	Retorna un filtro para aceptar todo tipo de ficheros
<code>FileFilter[] getChoosableFileFilters() / void setChoosableFileFilters(FileFilter[])</code>	Retorna/establece la lista de ficheros seleccionables por el usuario
<code>File getCurrentDirectory() / void setCurrentDirectory(File)</code>	Retorna/establece el directorio actual
<code>String getDescription(File)</code>	Retorna la descripción del fichero
<code>String getDialogTitle() / void setDialogTitle(String)</code>	Retorna/establece el título del cuadro de diálogo
<code>FileFilter getFileFilter() / void setFileFilter(FileFilter)</code>	Retorna/establece qué ficheros van a ser mostrados
<code>int getFileSelectionMode() / void setFileSelectionMode(int)</code>	Retorna/establece el modo de selección de ficheros: <code>JFileChooser.FILES_ONLY</code> , <code>JFileChooser.DIRECTORIES_ONLY</code> , <code>JFileChooser.FILES_AND_DIRECTORIES</code>
<code>FileView getFileView() / void setFileView(FileView)</code>	Retorna/establece el modo en que se van a visualizar ficheros y directorios
<code>Icon getIcon(File)</code>	Retorna el ícono asociado al fichero
<code>String getName(File)</code>	Retorna el nombre del fichero

Método	Descripción
File getSelectedFile() / File[] getSelectedFiles() / void setSelectedFile(File) / void setSelectedFiles(File[])	Retorna/establece el/los fichero(s) seleccionado(s)
String getTypeDescription(File)	Retorna una descripción del tipo de fichero
boolean isAcceptAllFileFilterUsed()	Devuelve true si se aceptan todos los ficheros
boolean isDirectorySelectionEnabled()	Devuelve true si se permite la selección de directorios
boolean isFileHidingEnabled() / void setFileHidingEnabled(boolean)	Retorna/establece si se deben mostrar los ficheros ocultos
boolean isFileSelectionEnabled()	Devuelve true si se permite seleccionar ficheros
boolean isMultiSelectionEnabled() / void setMultiSelectionEnabled(boolean)	Establece/indica si se permite seleccionar múltiples ficheros a la vez
boolean isTraversable(File)	Devuelve true si se puede acceder al directorio
void rescanCurrentDirectory()	Vuelve a obtener los ficheros del directorio actual
void setAcceptAllFileFilterUsed(boolean)	Establece si el filtro para aceptar todos los ficheros debe ser mostrado
int showDialog(Component, String)	Muestra el diálogo con el texto indicado en el botón de aceptar
int showOpenDialog(Component)	Muestra un diálogo para abrir un fichero
int showSaveDialog(Component)	Muestra un diálogo para guardar fichero

El proceso es simple, se crea un nuevo objeto `JFileChooser`, se modifican sus **propiedades** y se muestra con `showOpenDialog()`, `showSaveDialog()` o `showDialog()`. El valor devuelto puede ser:

- `JFileChooser.CANCEL_OPTION`: Se ha pulsado el botón **cancelar**
- `JFileChooser.APPROVE_OPTION`: Se ha pulsado el botón **aceptar**
- `JFileChooser.ERROR_OPTION`: Se ha producido un **error** o se ha cerrado el cuadro de diálogo

Tras esta instrucción podemos acceder a las propiedades del fichero/directorio seleccionado mediante el método `getSelectedFile()` que retorna un objeto de tipo `File`. En el caso de que permitamos **selecciones múltiples** de ficheros y/o directorios emplearemos `getSelectedFiles()` que retorna una matriz con todos los ficheros/directorios seleccionados.

abstract class / interface FileFilter (`javax.swing.filechooser.FileFilter`)

Para establecer un **filtro** de modo que sólo aparezcan determinados ficheros deberemos crear una clase que implemente el interfaz `FileFilter` o que derive de la clase del mismo nombre y redefinir los siguientes métodos:

- `boolean accept(File)`: Retornaremos `true` si el fichero debe ser **mostrado**, `false` en caso contrario (es importante retornar `true` si se trata de un directorio para permitir al usuario moverse por la estructura)
- `String getDescription()`: La **descripción** para el filtro que aparecerá en la lista desplegable de tipos de ficheros

Emplaremos un objeto de dicha clase junto con el método `setFileFilter()`.

abstract class FileView (`javax.swing.filechooser.FileView`)

Si lo que queremos es modificar la **forma en la que se muestran** ficheros y directorios deberemos crear una clase de vista, que derive de la clase abstracta `FileView` y que redefina los siguientes métodos:

Método	Descripción
String getDescription(File)	Una descripción del fichero pasado
Icon getIcon(File)	Una imagen para el fichero (puede ser un objeto <code>ImageIcon</code>)
String getName(File)	El nombre del fichero
String getTypeDescription(File)	La descripción del tipo de fichero
boolean isTraversable(File)	Si es un directorio, si debe poder atravesarlo o no

Veamos un ejemplo que permita únicamente seleccionar ficheros de imágenes:

```
package org.zabalburu.ficheros;

import com.formdev.flatlaf.FlatLightLaf;
import java.io.File;
import java.io.IOException;
import java.util.stream.Stream;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFileChooser;
import javax.swing.UIManager;
```

```

import javax.swing.filechooser.FileFilter;
import javax.swing.filechooser.FileView;

public class ImageFileChooser {
    public static void main(String[] args) throws IOException {
        try {
            UIManager.setLookAndFeel(new FlatLightLaf());
        } catch (Exception ex) {
            System.err.println("Error al inicializar el Look & Feel");
        }

        JFileChooser jfc = new JFileChooser("E:\\Iiigo\\Imagenes");
        FileFilter ft = jfc.getAcceptAllFileFilter();
        jfc.removeChoosableFileFilter(ft);
        jfc.setFileFilter(new FileFilter() {
            @Override
            public boolean accept(File f) {
                if (f.isDirectory()) {
                    return true;
                }
                if (f.getName().toLowerCase().matches(".*(gif|jpg|png|jpeg|bmp)")) {
                    return true;
                } else {
                    return false;
                }
            }
        });

        @Override
        public String getDescription() {
            return "Ficheros de imágenes (*.gif|*.jpg|*.png|*.bmp)";
        }
    });

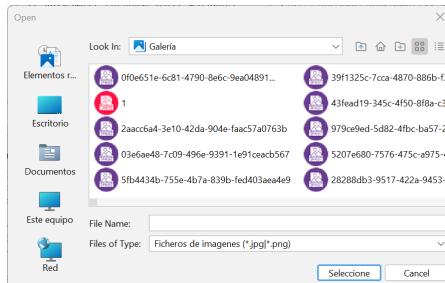
    jfc.setFileView(new FileView() {
        public Icon getIcon(File f) {
            if (f.isDirectory()) {
                return super.getIcon(f);
            }
            if (f.getName().toLowerCase().endsWith(".gif")) {
                return (new ImageIcon("gif.png"));
            }
            if (f.getName().toLowerCase().matches(".*(jpg|jpeg)")) {
                return (new ImageIcon("jpg.png"));
            }
            if (f.getName().toLowerCase().endsWith(".png")) {
                return (new ImageIcon("png.png"));
            }
            if (f.getName().toLowerCase().endsWith(".bmp")) {
                return (new ImageIcon("bmp.png"));
            }
            return super.getIcon(f);
        }
    });

    jfc.setApproveButtonText("Seleccione");
    jfc.setFileSelectionMode(JFileChooser.FILES_ONLY);
    jfc.setMultiSelectionEnabled(true);

    if (jfc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
        Stream.of(jfc.getSelectedFiles())
            .sorted((f1, f2) -> f1.getName().compareTo(f2.getName()))
            .forEach(System.out::println);
    }
}
}

```

La salida:



Al seleccionar:

```

C:\Users\ichue\OneDrive\Fotos\Typedown\03e6ae48-7c09-496e-9391-1e91ceacb567.png
C:\Users\ichue\OneDrive\Fotos\Typedown\0f0e651e-6c81-4790-8e6c-9ea048911946.png
C:\Users\ichue\OneDrive\Fotos\Acreditación\1.jpg

```

En este ejemplo se ha usado un **Look And Feel personalizado (FlatLaf)** para que tenga un aspecto más actual. Hay que añadir la dependencia:

```

<dependency>
    <groupId>com.formdev</groupId>

```

```

<artifactId>flatlaf</artifactId>
<version>3.5.4</version>
<type>jar</type>
</dependency>

```

4.3. Corrientes de Datos con Ficheros FileStreams DataStreams

class FileInputStream (java.io.FileInputStream)

Esta clase permite acceder a un fichero para **leer su contenido octeto a octeto**. Constructores:

- `FileInputStream(File fich) / FileInputStream(String camino)`: Crea y abre una **corriente de datos de entrada** desde un fichero.

El resto de los métodos son versiones redefinidas de los métodos de la clase `InputStream`.

class FileOutputStream (java.io.FileOutputStream)

Con esta clase podemos **escribir en un fichero octeto a octeto**. Constructores:

- `FileOutputStream(File fich[, boolean añadir]) / FileOutputStream(String camino[, boolean añadir])`: Crea y abre un fichero para añadir datos. Si el valor `añadir` es `true` las escrituras se hacen al **final del fichero**. Si es `false`, se hace al **principio** (eliminando el contenido previo del fichero).

El resto de los métodos son versiones redefinidas de los métodos de la clase `OutputStream`. Veamos cómo podemos copiar un fichero:

```

package ficheros;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JFileChooser;

public class Ficheros {
    public static void main(String[] args) {
        JFileChooser jfcFichero = new JFileChooser();
        jfcFichero.setMultiSelectionEnabled(false);
        jfcFichero.setFileSelectionMode(JFileChooser.FILES_ONLY);
        jfcFichero.setApproveButtonText("Copiar");

        if (jfcFichero.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            JFileChooser jfcDirectorio = new JFileChooser();
            jfcDirectorio.setMultiSelectionEnabled(false);
            jfcDirectorio.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);

            if (jfcDirectorio.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
                try {
                    FileInputStream fis = new FileInputStream(jfcFichero.getSelectedFile().getAbsolutePath());
                    String destino = new File(jfcDirectorio.getSelectedFile().getAbsolutePath());
                    System.out.println("Copiando a " + destino + "...");
                    FileOutputStream fos = new FileOutputStream(destino);

                    int b;
                    while ((b = fis.read()) != -1) {
                        fos.write(b);
                    }

                    fis.close();
                    fos.close();
                } catch (FileNotFoundException ex) {
                    Logger.getLogger(Ficheros.class.getName()).log(Level.SEVERE, null, ex);
                } catch (IOException ex) {
                    Logger.getLogger(Ficheros.class.getName()).log(Level.SEVERE, null, ex);
                }
            }
        }
    }
}

```

La salida:

```
Copiando a C:\Users\Iñigo\Documents\DocumentaciónGreenfoot.pdf...
```

En el ejemplo se muestra como es necesario **capturar las excepciones** `FileNotFoundException` que se puede lanzar al crear la corriente de datos si el fichero no existe y la excepción `IOException` que se puede dar en las lecturas o escrituras de datos.

class FileReader (java.io.FileReader)

Esta clase permite acceder a un fichero para **leer su contenido carácter a carácter**. Constructores:

- `FileReader(File fich) / FileReader(String camino)`: Crea y abre una corriente de datos de entrada desde un fichero.

El resto de los métodos son versiones redefinidas de los métodos de la clase `InputStreamReader`.

class FileWriter (java.io.FileWriter)

Con esta clase podemos **escribir en un fichero carácter a carácter**. Constructores:

- `FileWriter(File fich[], boolean añadir) / FileWriter(String camino[], boolean añadir)`: Crea y abre un fichero para añadir datos. Si el valor `añadir` es true las escrituras se hacen al final del fichero. Si es false, se hace al principio (eliminando el contenido previo del fichero).

El resto de los métodos son versiones redefinidas de los métodos de la clase `InputStreamWriter`.

Lectura / Escritura de Datos Básicos en Binario

Evidentemente, intentar trabajar con los datos a nivel de octeto o a nivel de carácter no tiene mucha utilidad dado que el acceso a los datos se hace a muy bajo nivel. En Java tenemos la posibilidad de **anidar una corriente de datos dentro de otra**:

```
CorrienteInterna interno = new CorrienteInterna(...); CorrienteExterna externo = new CorrienteExterna(interno);
```

En este caso y suponiendo corrientes de entrada, los datos serían leídos por la **corriente interna** pasados a la **corriente externa** que los puede procesar (para agruparlos, para convertirlos en otros tipos de datos, etc). De este modo, el programador trabaja a nivel de la corriente externa, empleando sus métodos, y es la corriente externa la que pide a la interna que lea los datos. A este tipo de operaciones se le suele denominar **filtrado de corrientes de datos** y ya hay predefinidas varias clases que nos facilitan este tipo de tareas.

class DataInputStream (java.io.DataInputStream)

La clase `DataInputStream` permite **convertir datos básicos (octetos)** en **los tipos de datos básicos de Java** (`int`, `byte`, etc). Por ello funciona en base a una corriente de datos que es la que realmente lee la información. En lo que a nosotros más nos interesa la corriente de datos interna puede ser un `FileInputStream`. Los métodos más importantes de la clase son:

- `DataInputStream(InputStream interno)`: Como se ve necesita un objeto que implemente `InputStream` (por ejemplo `FileInputStream`)
- `int read(byte[] b[, int pos[, int long]])`: Lee los octetos especificados y los almacena en la matriz b. Si se ha llegado al final del fichero retorna un valor -1.
- `boolean readBoolean()`: Lee un valor booleano. Lanza una excepción `EOFException` si se llega al final del fichero.
- `byte readByte()`: Lee un octeto. Lanza una excepción `EOFException` si se llega al final del fichero.
- `char readChar()`: Lee un carácter. Lanza una excepción `EOFException` si se llega al final del fichero.
- `double readDouble()`: Lee un valor de doble precisión. Lanza una excepción `EOFException` si se llega al final del fichero.
- `float readFloat()`: Lee un valor de simple precisión. Lanza una excepción `EOFException` si se llega al final del fichero.
- `int readInt()`: Lee un entero. Lanza una excepción `EOFException` si se llega al final del fichero.
- `long readLong()`: Lee un entero largo. Lanza una excepción `EOFException` si se llega al final del fichero.
- `short readShort()`: Lee un entero corto. Lanza una excepción `EOFException` si se llega al final del fichero.
- `String readUTF()`: Lee un conjunto de caracteres en formato UTF (Unicode) como una cadena. Lanza una excepción `EOFException` si se llega al final del fichero.
- `int skip(int octetos)`: Salta el número de octetos especificado. Lanza una excepción `EOFException` si se llega al final del fichero.

class DataOutputStream (java.io.DataOutputStream)

La clase `DataOutputStream` permite **escribir datos básicos a una corriente de datos**.

- `DataOutputStream(OutputStream interno)`: Crea el objeto
- `void flush()`: Fuerza la escritura de todos los datos pendientes
- `int size()`: Retorna el número de octetos escritos hasta el momento
- `void write(byte[] b[, int pos[, int long]])`: Escribe los octetos especificados en la matriz b en la corriente de datos
- `void writeBoolean(boolean b)`: Escribe el valor booleano especificado
- `void writeByte(byte b)`: Escribe el octeto
- `void writeBytes(String cad)`: Escribe la cadena como una secuencia de octetos
- `void writeChar(char c)`: Escribe el carácter
- `void writeChars(String cad)`: Escribe la cadena como una secuencia de caracteres
- `void writeDouble(double d)`: Escribe el valor de doble precisión dado
- `void writeFloat(float f)`: Escribe el valor de simple precisión dado
- `void writeInt(int i)`: Escribe el entero
- `void writeLong(long l)`: Escribe el entero largo
- `void writeShort(short s)`: Escribe el entero corto
- `void writeUTF(String cad)`: Escribe la cadena dada como un conjunto de caracteres UTF-8 (Unicode)

Ejemplo: Vamos a ver un pequeño programa que permite **almacenar los datos de una serie de productos**. De cada producto se guarda su id, nombre, precio y si está bajo stock:

```
package org.zabalburu.ficheros;
```

```

import java.io.*;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class DataStreams {
    public static void main(String[] args) {
        FileInputStream fis = null;
        FileOutputStream fos;
        DataInputStream dis;
        DataOutputStream dos;
        List<Producto> productos = new ArrayList<>();
        NumberFormat nf = NumberFormat.getInstance();
        NumberFormat nfMoneda = NumberFormat.getCurrencyInstance();
        int id;
        String nombre;
        double precio;
        boolean bajoStock;

        try {
            fis = new FileInputStream("productos.dat");
            dis = new DataInputStream(fis);
            while (true) {
                id = dis.readInt();
                nombre = dis.readUTF();
                precio = dis.readDouble();
                bajoStock = dis.readBoolean();
                productos.add(new Producto(id, nombre, precio, bajoStock));
            }
        } catch (FileNotFoundException ex) {
            // OK si no existe el fichero
        } catch (EOFException ex) {
            try {
                dis.close();
                fis.close();
            } catch (IOException ex1) {
                ex1.printStackTrace();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }

        id = 1;
        if (!productos.isEmpty()) {
            id = productos.get(productos.size() - 1).getId() + 1;
        }

        if (JOptionPane.showConfirmDialog(null, "Añadir Productos", "Pregunta",
                JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION) {
            do {
                nombre = JOptionPane.showInputDialog("Nombre Producto");
                String resp = JOptionPane.showInputDialog("Precio");
                try {
                    precio = nf.parse(resp).doubleValue();
                } catch (ParseException ex) {
                    precio = 0;
                }

                if (JOptionPane.showConfirmDialog(null, "Está Bajo Stock", "Pregunta",
                        JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION)
                    bajoStock = true;
                else
                    bajoStock = false;

                productos.add(new Producto(id, nombre, precio, bajoStock));
                id++;
            } while (JOptionPane.showConfirmDialog(null, "Añadir Otro Producto", "Pregunta",
                JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION);
        }

        productos.stream().forEach(System.out::println);

        try {
            fos = new FileOutputStream("productos.dat");
            dos = new DataOutputStream(fos);
            for (Producto p : productos) {
                dos.writeInt(p.getId());
                dos.writeUTF(p.getNombre());
                dos.writeDouble(p.getPrecio());
                dos.writeBoolean(p.isBajoStock());
            }
            dos.flush();
            dos.close();
            fos.close();
        } catch (FileNotFoundException ex) {
            Logger.getLogger(DataStreams.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
    }
}

```

```

        Logger.getLogger(DataStreams.class.getName()).log(Level.SEVERE, null, ex);
    }
}

class Producto {
    private int id;
    private String nombre;
    private double precio;
    private boolean bajoStock;

    public Producto() {}

    public Producto(int id, String nombre, double precio, boolean bajoStock) {
        this.id = id;
        this.nombre = nombre;
        this.precio = precio;
        this.bajoStock = bajoStock;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    public boolean isBajoStock() {
        return bajoStock;
    }

    public void setBajoStock(boolean bajoStock) {
        this.bajoStock = bajoStock;
    }

    @Override
    public String toString() {
        return "Producto {" + "id=" + id + ", nombre='" + nombre + "', precio=" + precio +
               ", bajoStock=" + bajoStock + '}';
    }
}

```

La salida:

```

Producto{id=1, nombre=Producto 1, precio=123.45, bajoStock=false}
Producto{id=2, nombre=Producto 2, precio=234.87, bajoStock=true}

```

4.4. Corrientes de Datos con Ficheros Buffers PrintWriter RandomAccess

Uso de Buffers

Las clases vistas hasta ahora para leer y escribir, leen y escriben los datos **directamente en la corriente de datos**. Especialmente con ficheros, este proceso puede consumir mucho tiempo. Para acelerar los procesos de lectura/escritura se pueden emplear clases que **almacenén la información temporalmente** antes de escribirla y que lean más allá de la información solicitada al objeto de reducir las operaciones de lectura/escritura sobre la corriente de datos que, especialmente en el caso de ficheros, son las que más tiempo consumen. Existen clases para el uso de buffers con corrientes de datos orientadas a octetos y orientadas a caracteres:

- **BufferedInputStream:** Trabaja sobre un `InputStream`
- **BufferedOutputStream:** Trabaja sobre un `OutputStream`
- **BufferedReader:** Trabaja sobre un objeto `Reader`
- **BufferedWriter:** Trabaja sobre un objeto `Writer`

Modificamos el primer ejemplo (copiar un fichero) para que trabaje **con y sin buffer** controlando el tiempo empleado con un fichero de varios megas. Para controlar el tiempo empleado utilizamos el método `System.currentTimeMillis()` que devuelve el número de milisegundos.

```
package ficheros;
```

```

import java.io.BufferedReader;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

```

```

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JFileChooser;

public class Ficheros {
    public static void main(String[] args) {
        JFileChooser jfcFichero = new JFileChooser();
        jfcFichero.setMultiSelectionEnabled(false);
        jfcFichero.setFileSelectionMode(JFileChooser.FILES_ONLY);
        jfcFichero.setApproveButtonText("Copiar");

        if (jfcFichero.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            JFileChooser jfcDirectorio = new JFileChooser();
            jfcDirectorio.setMultiSelectionEnabled(false);
            jfcDirectorio.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);

            if (jfcDirectorio.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
                try {
                    // Sin Buffer
                    FileInputStream fis = new FileInputStream(jfcFichero.getSelectedFile().getAbsolutePath());
                    File destino = new File(jfcDirectorio.getSelectedFile(),
                        jfcFichero.getSelectedFile().getName());
                    System.out.println("Tamaño: " + jfcFichero.getSelectedFile().length() + " bytes");
                    System.out.println("Copiando a " + destino + "...");
                    FileOutputStream fos = new FileOutputStream(destino);

                    int b;
                    long inicio = System.currentTimeMillis();
                    while ((b = fis.read()) != -1) {
                        fos.write(b);
                    }
                    long fin = System.currentTimeMillis();
                    fis.close();
                    fos.close();
                    System.out.println("Tiempo empleado: " + ((fin - inicio) / 1000.0) + " sg.");
                }
                // Con Buffer
                fis = new FileInputStream(jfcFichero.getSelectedFile().getAbsolutePath());
                BufferedInputStream bis = new BufferedInputStream(fis);
                System.out.println("Copiando con buffer a " + destino + "...");
                fos = new FileOutputStream(destino);
                BufferedOutputStream bos = new BufferedOutputStream(fos);

                inicio = System.currentTimeMillis();
                while ((b = bis.read()) != -1) {
                    bos.write(b);
                }
                bos.flush();
                fin = System.currentTimeMillis();
                bis.close();
                bos.close();
                fis.close();
                fos.close();
                System.out.println("Tiempo empleado: " + ((fin - inicio) / 1000.0) + " sg.");
            } catch (FileNotFoundException ex) {
                Logger.getLogger(Ficheros.class.getName()).log(Level.SEVERE, null, ex);
            } catch (IOException ex) {
                Logger.getLogger(Ficheros.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}

```

La salida:

```

Tamaño: 550945 bytes
Copiando a C:\Users\Iñigo\Documents\Downloads\55587.pdf...
Tiempo empleado: 3.947 sg.
Copiando con buffer a C:\Users\Iñigo\Documents\Downloads\55587.pdf...
Tiempo empleado: 0.016 sg.

```

class PrintWriter (java.io.PrintWriter)

La clase `PrintWriter` permite **escribir tipos de datos básicos en formato texto** a una corriente de datos de octetos o de caracteres. Como consecuencia, los datos escritos son **legibles directamente por el usuario**. Es la clase equivalente a `DataOutputStream` con la diferencia de que los datos no se almacenan en binario sino como texto. Si la información a escribir la separamos con algún carácter especial (, ;) se dice que son **ficheros delimitados** (múltiples programas permiten escribir y leer ficheros de este tipo). Un ejemplo serían los ficheros **CSV** (Comma Separated Values).

Veamos un ejemplo para escribir los empleados en un fichero CSV que pueda posteriormente ser importado desde Excel o LibreOffice. En este caso emplearemos como separador el ; para poder escribir números con decimales. Además es conveniente que las cadenas vayan entrecomilladas (al menos si pueden estar formadas por varias palabras):

```

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.DateFormat;
import java.text.NumberFormat;
import java.util.GregorianCalendar;

```

```

import java.util.stream.Stream;
import modelo.Empleado;

public class EmpleadosCSV {
    private static Empleado[] empleados = {
        new Empleado("1111111A", "Juan", "López", "Producción", 1235.45,
                      new GregorianCalendar(2001, 0, 15).getTime()),
        new Empleado("2222222B", "Ana", "Pérez", "Administración", 1050.75,
                      new GregorianCalendar(2001, 0, 17).getTime()),
        new Empleado("3333333C", "Luis", "Simón", "Producción", 1490.30,
                      new GregorianCalendar(2001, 0, 15).getTime()),
        new Empleado("4444444D", "Carlos", "Amor", "Administración", 1235.45,
                      new GregorianCalendar(2004, 3, 25).getTime()),
        new Empleado("5555555E", "Eva", "San Martín", "Dirección", 2235.75,
                      new GregorianCalendar(2001, 0, 15).getTime()),
        new Empleado("6666666F", "Luisa", "Jiménez", "Ventas", 1250.50,
                      new GregorianCalendar(2001, 2, 20).getTime()),
        new Empleado("7777777G", "Miguel", "De Andrés", "Ventas", 1250.50,
                      new GregorianCalendar(2002, 10, 9).getTime()),
        new Empleado("8888888H", "Ángel", "Sanz", "Producción", 1490.30,
                      new GregorianCalendar(2002, 7, 5).getTime()),
        new Empleado("9999999I", "María", "López", "Producción", 1490.30,
                      new GregorianCalendar(2003, 3, 30).getTime())
    };
}

private static DateFormat df = DateFormat.getDateInstance();
private static NumberFormat nf = NumberFormat.getInstance();

public static void main(String[] args) {
    try {
        PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter("empleados.csv")));
        String cabecera = "DNI;Nombre;Apellidos;Departamento;Sueldo;Fecha_Alta";
        System.out.println(cabecera);
        pw.println(cabecera);

        Stream.of(empleados).forEach(e -> {
            String linea = e.getDNI() + ";" + e.getNombre() + ";" + e.getApellido() + ";" +
                           e.getDepartamento() + ";" + nf.format(e.getSueldo()) + ";" + df.format(e.getFechaAlta());
            System.out.println(linea);
            pw.println(linea);
        });

        pw.flush();
        pw.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
}

```

La salida:

```

DNI;Nombre;Apellidos;Departamento;Sueldo;Fecha_Alta
1111111A;"Juan";"López";"Producción";1.235,45;15-ene-2001
2222222B;"Ana";"Pérez";"Administración";1.050,75;17-ene-2001
3333333C;"Luis";"Simón";"Producción";1.490,3;15-ene-2001
4444444D;"Carlos";"Amor";"Administración";1.235,45;25-abr-2004
5555555E;"Eva";"San Martín";"Dirección";2.235,75;15-ene-2001
6666666F;"Luisa";"Jiménez";"Ventas";1.250,5;20-mar-2001
7777777G;"Miguel";"De Andrés";"Ventas";1.250,5;09-nov-2002
8888888H;"Ángel";"Sanz";"Producción";1.490,3;05-ago-2002
9999999I;"María";"López";"Producción";1.490,3;30-abr-2003

```

Si lo abrimos desde OpenOffice:

DNI	Nombre	Apellidos	Departamento	Sueldo	Fecha Alta
1111111A	Juan	López	Producción	1235.45	15-ene-2001
2222222B	Ana	Pérez	Administración	1050.75	17-ene-2001
3333333C	Luis	Simón	Producción	1490.3	15-ene-2001
4444444D	Carlos	Amor	Administración	1235.45	25-abr-2004
5555555E	Eva	San Martín	Dirección	2235.75	15-ene-2001
6666666F	Luisa	Jiménez	Ventas	1250.5	20-mar-2001
7777777G	Miguel	De Andrés	Ventas	1250.5	09-nov-2002
8888888H	Ángel	Sanz	Producción	1490.3	05-ago-2002
9999999I	María	López	Producción	1490.3	30-abr-2003

class BufferedReader (java.io.BufferedReader)

No existe una clase opuesta a `PrintWriter` que permita recuperar los datos escritos con la anterior. La única posibilidad es emplear el método `readLine()` que aparece en la clase `BufferedReader` y después emplear la clase `StringTokenizer` para **descomponer la cadena en sus partes originales** (es necesario separar cada campo con algún carácter delimitador).

Ejemplo: Vamos a ver un programa que lea los datos del fichero CSV que hemos generado en el apartado anterior:

```
import java.io.BufferedReader;
```

```

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.text.DateFormat;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author ichue
 */
public class LeerEmpleadosCSV {
    private static DateFormat df = DateFormat.getDateInstance();
    private static NumberFormat nf = NumberFormat.getInstance();

    public static void main(String[] args) {
        List<Empleado> empleados = new ArrayList<>();
        try {
            BufferedReader br = new BufferedReader(new FileReader("empleados.csv"));
            String linea = br.readLine(); // Leemos la cabecera

            while ((linea = br.readLine()) != null) {
                String[] tokens = linea.split(";");
                Empleado e = new Empleado();
                e.setDni(tokens[0]);
                e.setNombre(tokens[1]);
                e.setApellidos(tokens[2]);
                e.setDepartamento(tokens[3]);

                double sueldo = 0;
                try {
                    Number n = nf.parse(tokens[4]);
                    sueldo = n.doubleValue();
                } catch (ParseException ex) {}
                e.setSueldo(sueldo);

                Date fechaAlta = new Date();
                try {
                    fechaAlta = df.parse(tokens[5]);
                } catch (ParseException ex) {}
                e.setFechaAlta(fechaAlta);

                empleados.add(e);
            }
            br.close();
        } catch (FileNotFoundException ex) {
            Logger.getLogger(LeerEmpleadosCSV.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
            Logger.getLogger(LeerEmpleadosCSV.class.getName()).log(Level.SEVERE, null, ex);
        }

        empleados.stream().forEach(System.out::println);
    }
}

```

El resultado:

```

Empleado{dni=11111111A, nombre="Juan", apellidos="López", departamento="Producción", sueldo=1235.45, fechaAlta=Mon Jan 15 00:00:00 CET 2001}
Empleado{dni=22222222B, nombre="Ana", apellidos="Pérez", departamento="Administración", sueldo=1050.75, fechaAlta=Wed Jan 17 00:00:00 CET 2001}
Empleado{dni=33333333C, nombre="Luis", apellidos="Simón", departamento="Producción", sueldo=1490.3, fechaAlta=Mon Jan 15 00:00:00 CET 2001}
Empleado{dni=44444444D, nombre="Carlos", apellidos="Amor", departamento="Administración", sueldo=1235.45, fechaAlta=Sun Apr 25 00:00:00 CEST 2004}
Empleado{dni=55555555E, nombre="Eva", apellidos="San Martín", departamento="Dirección", sueldo=2235.75, fechaAlta=Mon Jan 15 00:00:00 CET 2001}
Empleado{dni=66666666F, nombre="Luisa", apellidos="Jiménez", departamento="Ventas", sueldo=1250.5, fechaAlta=Tue Mar 20 00:00:00 CET 2001}
Empleado{dni=77777777G, nombre="Miguel", apellidos="De Andrés", departamento="Ventas", sueldo=1250.5, fechaAlta=Sat Nov 09 00:00:00 CET 2002}
Empleado{dni=88888888H, nombre="Ángel", apellidos="Sanz", departamento="Producción", sueldo=1490.3, fechaAlta=Mon Aug 05 00:00:00 CEST 2002}
Empleado{dni=99999999I, nombre="María", apellidos="López", departamento="Producción", sueldo=1490.3, fechaAlta=Wed Apr 30 00:00:00 CEST 2003}

```

Acceso aleatorio a Ficheros (java.io.RandomAccessFile)

La clase `RandomAccessFile` permite leer y/o escribir datos en cualquier parte de un fichero. Implementa las interfaces `DataInput` y `DataOutput`, por lo que dispone de métodos para leer y escribir los tipos básicos de datos, así como para cerrar el fichero. Los ficheros de acceso aleatorio pueden abrirse en modo de sólo lectura (`r`) o en modo de lectura/escritura (`rw`). Todos los métodos pueden lanzar una `IOException`. Además los métodos de lectura (provenientes de `DataReader`) pueden lanzar una `EOFException` si se llega al final del fichero).

- `RandomAccessFile(File fich, String modo)` / `RandomAccessFile(String fich, String modo)`: Abre un fichero aleatorio a partir de un objeto `File` o de la trayectoria al mismo. El modo puede ser `r` o `rw`.
- `long getFilePointer()`: Retorna la **posición del puntero** del fichero que determina el lugar donde va a realizarse la siguiente operación de E/S.
- `long length()`: Retorna el **tamaño del fichero**
- `void seek(long pos)`: Establece la **posición del puntero** al fichero.
- `void setLength(long tam)`: Establece un **nuevo tamaño** para el fichero
- `int skipBytes(int num)`: Salta el número de octetos indicado retornando el número de octetos que realmente se han saltado. No lanza una `EOFException`.

En el siguiente ejemplo vamos a escribir los empleados en un fichero. Dado que queremos acceder a ellos de manera aleatoria, necesitamos escribir **cadenas de tamaño fijo**. Para ello se definen un par de métodos auxiliares (escribirCadena y leerCadena).

```

import java.io.DataInput;
import java.io.DataOutput;
import java.io.EOFException;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.text.DateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.text.NumberFormat;
import java.util.stream.Stream;

public class EmpleadosRandom {
    private static Empleado[] empleados = {
        new Empleado("1111111A", "Juan", "López", "Producción", 1235.45,
            new GregorianCalendar(2001, 0, 15).getTime()),
        new Empleado("2222222B", "Ana", "Pérez", "Administración", 1050.75,
            new GregorianCalendar(2001, 0, 17).getTime()),
        new Empleado("3333333C", "Luis", "Simón", "Producción", 1490.30,
            new GregorianCalendar(2001, 0, 20).getTime()),
        new Empleado("4444444D", "Carlos", "Añor", "Administración", 1235.45,
            new GregorianCalendar(2004, 3, 25).getTime()),
        new Empleado("5555555E", "Eva", "San Martín", "Dirección", 2235.75,
            new GregorianCalendar(2001, 0, 8).getTime()),
        new Empleado("6666666F", "Luisa", "Jiménez", "Ventas", 1250.50,
            new GregorianCalendar(2001, 2, 20).getTime()),
        new Empleado("7777777G", "Miguel", "De Andrés", "Ventas", 1250.50,
            new GregorianCalendar(2002, 10, 9).getTime()),
        new Empleado("8888888H", "Ángel", "Sanz", "Producción", 1490.30,
            new GregorianCalendar(2002, 7, 15).getTime()),
        new Empleado("9999999I", "María", "López", "Producción", 1490.30,
            new GregorianCalendar(2003, 3, 30).getTime())
    };

    private static DateFormat df = DateFormat.getDateInstance();
    private static NumberFormat nf = NumberFormat.getInstance();

    private static String Leer(int tamaño, DataInput in) throws IOException {
        StringBuffer temp = new StringBuffer();
        char c;
        boolean mas = true;
        int i = 0;

        while (mas && (i < tamaño)) {
            c = in.readChar();
            temp.append(c);
            i++;
            if (c == 0) {
                mas = false;
            }
        }

        in.skipBytes(2 * (tamaño - i));
        return temp.toString();
    }

    private static void escribir(String str, int tamaño, DataOutput out) throws IOException {
        for (int i = 0; i < tamaño; i++) {
            char c = 0;
            if (i < str.length()) {
                c = str.charAt(i);
            }
            out.writeChar(c);
        }
    }

    private static int TAM_EMPL = (9 + 30 + 50 + 40) * 2 + 8 + 12;

    private static void escribirEmpleado(Empleado e, RandomAccessFile raf) throws IOException {
        escribir(e.getDni(), 9, raf); // 9 * 2
        escribir(e.getApellidos(), 30, raf); // 30 * 2
        escribir(e.getNombre(), 50, raf); // 50 * 2
        escribir(e.getDepartamento(), 40, raf); // 40 * 2
        raf.writeDouble(e.getSueldo()); // 8

        GregorianCalendar gc = new GregorianCalendar();
        gc.setTime(e.getFechaAlta());
        raf.writeInt(gc.get(Calendar.DATE)); // 4
        raf.writeInt(gc.get(Calendar.MONTH)); // 4
        raf.writeInt(gc.get(Calendar.YEAR)); // 4
    }

    private static Empleado leerEmpleado(RandomAccessFile raf) throws IOException {
        Empleado e = new Empleado();
        e.setDni(Leer(9, raf));
        e.setApellidos(Leer(30, raf));
        e.setNombre(Leer(50, raf));
        e.setDepartamento(Leer(40, raf));
        e.setSueldo(raf.readDouble());
    }
}

```

```

GregorianCalendar gc = new GregorianCalendar();
gc.clear();
gc.set(Calendar.DATE, raf.readInt());
gc.set(Calendar.MONTH, raf.readInt());
gc.set(Calendar.YEAR, raf.readInt());
e.setFechaAlta(gc.getTime());

return e;
}

public static void main(String[] args) {
    try {
        File f = new File("empleados.rnd");
        RandomAccessFile raf = new RandomAccessFile(f, "rw");

        if (f.length() == 0) {
            Stream.of(empleados).forEach(e -> {
                try {
                    escribirEmpleado(e, raf);
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            });
        }

        System.out.println("Escritos: " + raf.length() / TAM_EMPL + " empleados.");

        // Leemos el PRIMER empleado
        raf.seek(0);
        Empleado e = leerEmpleado(raf);
        System.out.println("Primer Empleado: " + e);

        // Leemos el ÚLTIMO empleado
        raf.seek(TAM_EMPL * (empleados.length - 1));
        System.out.println("Último Empleado: " + leerEmpleado(raf));

        // Leemos el PRIMER empleado para MODIFICARLO
        raf.seek(0);
        e = leerEmpleado(raf);
        e.setSueldo(2000);

        // Escribimos el empleado MODIFICADO
        raf.seek(0);
        escribirEmpleado(e, raf);

        // Lo volvemos a leer para comprobar que se ha modificado correctamente
        raf.seek(0);
        System.out.println("Primer Empleado Modificado: " + leerEmpleado(raf));

        raf.close();
    } catch (EOFException ex) {
        // Fin de fichero
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
}

```

La salida:

```

Escritos: 9 empleados.
Primer Empleado: Empleado{dni=11111111A, nombre=Juan, apellidos=López, departamento=Producción, sueldo=1235.45, fechaAlta=Mon Jan 15 00:00:00 CET 2001}
Último Empleado: Empleado{dni=99999999I, nombre=María, apellidos=López, departamento=Producción, sueldo=1490.3, fechaAlta=Wed Apr 30 00:00:00 CEST 2003}
Primer Empleado Modificado: Empleado{dni=11111111A, nombre=Juan, apellidos=López, departamento=Producción, sueldo=2000.0, fechaAlta=Mon Jan 15 00:00:00 CET

```

4.5. Serialización de Objetos

Serialización de Objetos en Java

Todas las clases que hemos visto hasta ahora para gestionar corrientes de datos manipulan **datos básicos** (como mucho manipulan cadenas de texto), lo que no los hace muy útiles para trabajar con objetos. Para simplificar el almacenamiento y recuperación de objetos, en Java se implementa un mecanismo denominado **serialización** que básicamente se encarga de almacenar y recuperar objetos. Lógicamente, esto implica almacenar el tipo de objeto y los valores de sus propiedades.

Para poder guardar y leer objetos necesitamos una corriente de datos `ObjectOutputStream` y `ObjectInputStream` respectivamente. Para indicar que una clase puede serializarse, debe implementar el interfaz `java.io.Serializable`.

class ObjectOutputStream (java.io.ObjectOutputStream)

Esta clase implementa los interfaces `DataOutput` y `ObjectOutput` por lo que permite escribir tipos básicos de datos y objetos.

- `ObjectOutputStream(OutputStream)`: Crea una corriente de datos a partir de una corriente de salida de datos binarios para escritura
- `void writeObject(Object obj)`: Escribe el objeto en la corriente de datos

class ObjectInputStream (java.io.ObjectInputStream)

Esta clase implementa los interfaces `DataInput` y `ObjectInput` por lo que permite escribir tipos básicos de datos y objetos.

- `ObjectInputStream(InputStream)`: Crea una corriente de datos a partir de una corriente de salida de datos binarios para lectura

- `Object readObject()`: Lee el objeto desde la corriente de datos

En el ejemplo vamos a escribir los empleados en un fichero mediante **serialización** y los leeremos después. Dado que la clase `Date` es serializable podemos escribir la fecha sin problemas.

NOTA: Hay que implementar `Serializable` en la clase `Empleado`.

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.GregorianCalendar;
import java.util.List;
import java.util.stream.Stream;
import modelo.Empleado;

public class EmpleadosObj {
    private static Empleado[] empleados = {
        new Empleado("1111111A", "Juan", "López", "Producción", 1235.45,
                      new GregorianCalendar(2001, 0, 15).getTime()),
        new Empleado("2222222B", "Ana", "Pérez", "Administración", 1050.75,
                      new GregorianCalendar(2001, 0, 17).getTime()),
        new Empleado("3333333C", "Luis", "Simón", "Producción", 1490.30,
                      new GregorianCalendar(2001, 0, 15).getTime()),
        new Empleado("4444444D", "Carlos", "Amor", "Administración", 1235.45,
                      new GregorianCalendar(2004, 3, 25).getTime()),
        new Empleado("5555555E", "Eva", "San Martín", "Dirección", 2235.75,
                      new GregorianCalendar(2001, 0, 15).getTime()),
        new Empleado("6666666F", "Luisa", "Jiménez", "Ventas", 1250.50,
                      new GregorianCalendar(2001, 2, 20).getTime()),
        new Empleado("7777777G", "Miguel", "De Andrés", "Ventas", 1250.50,
                      new GregorianCalendar(2002, 10, 9).getTime()),
        new Empleado("8888888H", "Ángel", "Sanz", "Producción", 1490.30,
                      new GregorianCalendar(2002, 7, 15).getTime()),
        new Empleado("9999999I", "María", "López", "Producción", 1490.30,
                      new GregorianCalendar(2003, 3, 30).getTime())
    };
}

public static void main(String[] args) {
    try {
        File f = new File("empleados.obj");
        final ObjectOutputStream oos = new ObjectOutputStream(
            new BufferedOutputStream(
                new FileOutputStream(f)));
        Stream.of(empleados).forEach(e -> {
            try {
                oos.writeObject(e);
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        });
        oos.flush();
        oos.close();
    }

    ObjectInputStream ois = new ObjectInputStream(
        new BufferedInputStream(
            new FileInputStream(f)));
    List<Empleado> empleados = new ArrayList<>();
    try {
        while (true) {
            Empleado e = (Empleado) ois.readObject();
            empleados.add(e);
        }
    } catch (EOFException ex) {
        // Fin de fichero
    } catch (ClassNotFoundException ex) {
        ex.printStackTrace();
    }

    empleados.forEach(System.out::println);
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
}
```

La salida:

```
Empleado{dni=1111111A, nombre=Juan, apellidos=López, departamento=Producción, sueldo=1235.45, fechaAlta=Mon Jan 15 00:00:00 CET 2001}
Empleado{dni=2222222B, nombre=Ana, apellidos=Pérez, departamento=Administración, sueldo=1050.75, fechaAlta=Wed Jan 17 00:00:00 CET 2001}
Empleado{dni=3333333C, nombre=Luis, apellidos=Simón, departamento=Producción, sueldo=1490.3, fechaAlta=Mon Jan 15 00:00:00 CET 2001}
Empleado{dni=4444444D, nombre=Carlos, apellidos=Amor, departamento=Administración, sueldo=1235.45, fechaAlta=Sun Apr 25 00:00:00 CEST 2004}
Empleado{dni=5555555E, nombre=Eva, apellidos=San Martín, departamento=Dirección, sueldo=2235.75, fechaAlta=Mon Jan 15 00:00:00 CET 2001}
Empleado{dni=6666666F, nombre=Luisa, apellidos=Jiménez, departamento=Ventas, sueldo=1250.5, fechaAlta=Tue Mar 20 00:00:00 CET 2001}
```

```
Empleado{dni=77777777G, nombre=Miguel, apellidos=De Andrés, departamento=Ventas, sueldo=1250.5, fechaAlta=Sat Nov 09 00:00:00 CET 2002}
Empleado{dni=88888888H, nombre=Ángel, apellidos=Sanz, departamento=Producción, sueldo=1490.3, fechaAlta=Mon Aug 05 00:00:00 CEST 2002}
Empleado{dni=99999999I, nombre=Maria, apellidos=López, departamento=Producción, sueldo=1490.3, fechaAlta=Wed Apr 30 00:00:00 CEST 2003}
```

Es importante tener en cuenta que los objetos deserializados **se crean de nuevo**. Es decir, los empleados de la lista son diferentes de los de la matriz.

NOTA: Las propiedades de una clase disponen del modificador `transient` que permite especificar que no queremos serializarlas.

El mecanismo de serialización por defecto suele ser suficiente en la mayoría de los casos. De cualquier modo, si queremos modificar la forma en la que se escriben y/o leen los datos podemos redefinir los métodos:

```
private void readObject(ObjectInputStream in)
private void writeObject(ObjectOutputStream out)
```

En estos métodos, la primera llamada debiera ser a los métodos `defaultReadObject` y `defaultWriteObject` que se encargan de guardar la información sobre la definición de la clase y los datos de sus superclases.

Otra posibilidad pasa por implementar el interfaz `Externalizable` y definir los métodos:

```
public void readExternal(ObjectInput in)
public void writeExternal(ObjectOutput out)
```

En este segundo caso, el programador es el responsable de **almacenar y recuperar toda la información del objeto** (incluyendo la información de sus superclases).

Por ejemplo, podríamos incluir los métodos en la clase `Empleado`:

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.io.Serializable;
import java.text.DateFormat;
import java.text.NumberFormat;
import java.util.Date;

public class Empleado implements Serializable, Externalizable {
    // ... resto de la clase ...

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeUTF(dni);
        out.writeUTF(nombre);
        out.writeUTF(apellidos);
        out.writeUTF(departamento);
        out.writeDouble(sueldo);
        out.writeObject(fechaAlta);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        dni = in.readUTF();
        apellidos = in.readUTF();
        nombre = in.readUTF();
        departamento = in.readUTF();
        sueldo = in.readDouble();
        fechaAlta = (Date) in.readObject();
    }
}
```

La salida es exactamente la misma. Se puede comprobar que el fichero resultante es **más pequeño** (no se guarda información de la clase). Las clases que definen sus propios métodos de escritura/lectura son de un **30 a un 40% más rápidas** a la hora de manejar la información.

4.6. Actividad 25 - Corrientes de Datos

Actividad: Sistema de Gestión de Envíos para Empresa de Mensajería

[Classroom](#)

Instrucciones

En una empresa de mensajería necesitan desarrollar una aplicación que les permita gestionar los envíos realizados. Para gestionar los mismos desean emplear una clase `Envío` que disponga de los siguientes campos:

- `int idEnvío`: El identificador del envío. Será un entero único.
- `String direcciónEntrega`: La dirección donde se debe entregar el envío
- `String población`: La población donde se entregará el envío
- `Empleado empleado`: El empleado al que se ha asignado el pedido
- `Date fechaEnvío`: La fecha en la que se envía el paquete
- `Date fechaEntrega`: La fecha en la que está previsto entregar el envío (null si el envío no ha sido entregado)
- `double coste`: El importe a cobrar por el porte.

La clase `Empleado`, por su parte:

- `int idEmpleado`: El identificador del empleado. Será un entero único
- `String nombre`: El nombre
- `String apellidos`: Los apellidos

NOTA: Los datos de los empleados se almacenan en una hoja de cálculo que se exporta en formato CSV (crearla vosotr@s con unos cuantos datos de prueba).

DAO (no hace falta emplear interfaces)

Diseñar un DAO para Empleados que:

- En el constructor: leerá los datos del fichero CSV y los almacenará en una lista
- Dispondrá de los siguientes métodos (emplear streams):
- `getEmpleados()`: Retornará todos los empleados ordenados por apellidos y nombre
- `getEmpleado(int idEmpleado)`: Retornará el empleado con el idEmpleado indicado, null si no existe

Diseñar el DAO para Envío. La información de los envíos se almacenará en un fichero empleando cualquiera de las corrientes de datos vistas en clase.

- En el constructor: leerá los datos del fichero y los almacenará en una lista
- Dispondrá de los siguientes métodos:
- `getEnvios()`: Retornará los envíos ordenados en descendente por fechaEnvío
- `getEnvio(int idEnvio)`: Retornará el envío con el idEnvio indicado, null si no existe
- `añadirEnvio(Envio e)`: Añadirá el envío a la lista y almacenará todos los envíos en el fichero (hacerlo en un método privado aparte)
- `entregarEnvio(int idEnvio, Date fechaEntrega)`: Si existe el envío, le asignará la fechaEntrega indicada y se almacenarán todos los envíos en el fichero

Crear una clase de Servicio que incluya todos los métodos de las dos clases anteriores.

Aplicación

Una vez definidas las clases, se pide diseñar una aplicación que mostrará el siguiente menú:

```
Gestión de Envíos
-----
1. Añadir Envío
2. Listado Envíos
3. Envíos Entregados
4. Generar Resumen Empleados
5. Salir
Opción [1-4]:
```

Añadir Envío

Añadirá un nuevo envío a la lista con los datos pedidos al usuario (dirección de entrega, población, empleado asignado, fecha de envío y coste). El proceso podrá repetirse a petición del usuario.

Listado Envíos

Hará un listado de todos los envíos. Se mostrará cuántos envíos pendientes (fechaEntrega es null) y entregados hay y la suma de los costes en cada caso. Además, al final se deberá mostrar cuantos envíos y la suma de los costes de los envíos entregados por mes y año (hacerlo con streams).

Envíos Entregados

Pedirá el id del envío entregado y, si existe (si no dará un mensaje de error) pedirá la fecha de entrega y actualizará la información del envío localizado.

Generar Resumen Empleados

Se deberá generar un fichero CSV en el que se incluyan los siguientes datos (la primera fila son las cabeceras):

```
Empleados;Pedidos;Entregados;Pendientes;Total Coste Entregados;Total Coste Pendientes
Apellidos, nombre;cuenta;cuentaEntregados;cuentaPendientes;sumaCosteEntregados;sumaCostePendientes
...
```

Dicho fichero deberá poder leerse correctamente desde la aplicación Calc de LibreOffice.

4.7. Ficheros Comprimidos ZIP

Compresión de Datos en Java con ZipInputStream y ZipOutputStream

Entre las corrientes de datos disponibles en Java, tenemos las clases `ZipInputStream` y `ZipOutputStream` que reciben una corriente de datos y la comprimen. Habitualmente la corriente de datos es de tipo fichero (`FileInputStream` y `FileOutputStream` respectivamente) que referencian el fichero .zip.

Dado que en un mismo fichero comprimido se pueden incluir múltiples ficheros (y directorios) es necesario emplear además la clase `ZipEntry` que proporciona acceso a cada entrada (fichero) en el fichero comprimido.

Crear un fichero ZIP

Para crear un fichero zip los pasos serían:

1. Obtener un `FileOutputStream` para el fichero zip

```
java FileOutputStream fos = new FileOutputStream("fichero.zip");
```

1. Obtener una corriente de datos comprimida hacia ese fichero

```
java ZipOutputStream zos = new ZipOutputStream(fos);
```

1. Crear una entrada para el nuevo fichero

```
java ZipEntry ze = new ZipEntry("fichero.datos");
```

1. Añadir la entrada al fichero

```
java zos.putNextEntry(ze);
```

- Escribir en la corriente de salida. Los datos se almacenarán comprimidos en el fichero fichero.datos

```
java zos.write...
1. Cerrar la entrada del fichero
java zos.closeEntry();
1. Repetir los pasos 3 – 6 para el resto de los ficheros a añadir
2. Cerrar la corriente de datos
java zos.close();
```

Leer un fichero ZIP

Para recuperar el contenido de las entradas de un fichero comprimido, los pasos serían:

- Obtener un FileInputStream para el fichero zip
- ```
java FileInputStream fis = new FileInputStream("fichero.zip");
1. Obtener una corriente de datos comprimida desde ese fichero
```
- Obtener una entrada del fichero

```
java ZipInputStream zis = new ZipInputStream(fis);
1. Comprobar que la entrada existe (opcionalmente obtener información de la misma)
java if (ze != null)
1. Leer el contenido de la corriente de datos y procesarla
java zis.read...;
```

- Cerrar la entrada del fichero

```
java zis.closeEntry();
1. Repetir los pasos 3 – 6 para el resto de los ficheros a procesar (mientras ze no sea nulo)
2. Cerrar la corriente de datos
java zis.close();
```

Lógicamente, es posible anidar las corrientes comprimidas a otras corrientes como las corrientes con buffer o las corrientes de objetos.

## Ejemplo: Comprimir objetos Empleado

Veamos como comprimir los empleados en un fichero:

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.GregorianCalendar;
import java.util.List;
import java.util.stream.Stream;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;
import java.util.zip.ZipOutputStream;
import modelo.Empleado;
public class EmpleadosZIP {
 private static Empleado[] empleados = {
 new Empleado("1111111A", "Juan", "López", "Producción", 1235.45,
 new GregorianCalendar(2001, 0, 15).getTime()),
 // ... más empleados
 };
 public static void main(String[] args) {
 try {
 File f = new File("empleados.zip");
 ZipOutputStream zos = new ZipOutputStream(
 new BufferedOutputStream(
 new FileOutputStream(f)));
 ZipEntry ze = new ZipEntry("empleados.datos");
 zos.putNextEntry(ze);
 final ObjectOutputStream oos = new ObjectOutputStream(zos);
 Stream.of(empleados).forEach(e -> {

```

```

 try {
 oos.writeObject(e);
 } catch (IOException ex) {
 ex.printStackTrace();
 }
 });

 os.flush();
 os.close();

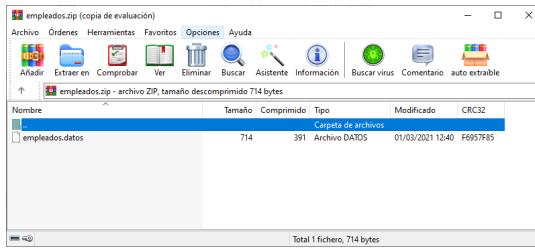
 ZipInputStream zis = new ZipInputStream(
 new BufferedInputStream(
 new FileInputStream(f)));
 zis.getNextEntry();
 ObjectInputStream ois = new ObjectInputStream(zis);

 List<Empleado> empleados = new ArrayList<>();
 try {
 while (true) {
 Empleado e = (Empleado) ois.readObject();
 empleados.add(e);
 }
 } catch (EOFException ex) {
 // Fin de fichero
 } catch (ClassNotFoundException ex) {
 ex.printStackTrace();
 }

 empleados.forEach(System.out::println);
} catch (IOException ex) {
 ex.printStackTrace();
}
}
}

```

La salida es igual a la del último ejemplo. Si miramos el fichero comprimido:



**NOTA:** Si vamos a pasar la corriente de datos por otra corriente de datos (por ejemplo por un `ObjectOutputStream`) es imprescindible hacerlo después de obtener el `ZipEntry`. Si no dará un mensaje de fin de fichero.

**NOTA:** Si queremos crear subdirectorios basta con poner la ruta en el `ZipEntry` al escribirlo (por ejemplo `ZipEntry ze = new ZipEntry("datos/empleados.obj");`

## Comprimir / Descomprimir Directorios

En el siguiente ejemplo se puede ver cómo podemos comprimir un directorio de manera recursiva:

```

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;
import javax.swing.JFileChooser;

public class Comprimir {
 private static File directorio;

 public static void main(String[] args) {
 JFileChooser jfc = new JFileChooser();
 jfc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
 jfc.setMultiSelectionEnabled(false);

 if (jfc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
 try {
 ZipOutputStream zos = new ZipOutputStream(
 new BufferedOutputStream(
 new FileOutputStream("copia.zip")));
 directorio = jfc.getSelectedFile();
 comprimirDirectorio(zos, directorio);
 zos.close();
 } catch (IOException ex) {
 // Manejo de excepciones
 }
 }
 }
}

```

```

}

private static void comprimirDirectorio(ZipOutputStream zos, File d) {
 String prefijo = "";
 if (d.equals(directorio)) {
 prefijo = d.getPath().substring(directorio.getPath().length() + 1) + "\\\";
 }

 File[] ficheros = d.listFiles();
 if (ficheros != null) {
 try {
 for (File f : ficheros) {
 if (f.isFile()) {
 zipFile(zos, prefijo, f);
 } else {
 comprimirDirectorio(zos, f);
 }
 }
 } catch (IOException ex) {
 ex.printStackTrace();
 }
 }
}

private static void zipFile(ZipOutputStream zos, String prefijo, File f) throws IOException {
 try {
 ZipEntry ze = new ZipEntry(prefijo + f.getName());
 System.out.println("Comprimiendo " + prefijo + f.getName() + "...");
 zos.putNextEntry(ze);

 BufferedInputStream bis = new BufferedInputStream(new FileInputStream(f));
 int dato;
 while ((dato = bis.read()) != -1) {
 zos.write(dato);
 }

 bis.close();
 zos.flush();
 zos.closeEntry();
 } catch (IOException ex) {
 ex.printStackTrace();
 }
}
}

```

## Descomprimir Directorios

El código para descomprimir:

```

import java.io.BufferedReader;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;
import javax.swing.JFileChooser;

public class Descomprimir {
 public static void main(String[] args) throws IOException {
 JFileChooser jfc = new JFileChooser();
 jfc.setFileSelectionMode(JFileChooser.FILES_ONLY);
 jfc.setMultiSelectionEnabled(false);

 if (jfc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
 File fichero = jfc.getSelectedFile();
 jfc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
 jfc.setMultiSelectionEnabled(false);

 if (jfc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
 try {
 System.out.println(fichero.getName());
 ZipInputStream zis = new ZipInputStream(
 new BufferedInputStream(
 new FileInputStream(fichero)));

 ZipEntry ze;
 String dir = jfc.getSelectedFile().getPath();

 while ((ze = zis.getNextEntry()) != null) {
 String fich = new File(ze.getName()).getPath();
 int pos = fich.lastIndexOf("\\\");

 if (pos != -1) {
 File d = new File(dir, fich.substring(0, pos));
 d.mkdirs();
 }
 }

 BufferedOutputStream bos = new BufferedOutputStream(
 new FileOutputStream(new File(dir, fich)));

```

#### 4.8. Sockets corrientes a través de TCP IP Threads Runnable

## Comunicación en Red con Sockets en Java

Las corrientes de datos también se pueden emplear para enviar/recibir información a través de la red. Para ello vamos a emplear los sockets que proporcionan una abstracción para enviar/recibir esos datos. En Java disponemos de una clase para el cliente y otra para los servidores.

## ServerSocket (java.net.ServerSocket)

La clase `ServerSocket` permite aceptar conexiones de clientes a través del protocolo TCP/IP. Tal y como establece el protocolo, la clase `ServerSocket` se pone a la "escucha" en un puerto (un entero) específico que permite al equipo dirigirle la información correctamente. Los clientes que quieran "hablar" con este socket deberán dirigirse a dicho puerto.

Algunos de los métodos más interesantes:

- **Constructores:**
  - ServerSocket()
  - ServerSocket(int puerto)
  - ServerSocket(int puerto, int backlog)
  - ServerSocket(int puerto, int backlog, InetAddress dirección)

Crea un socket de servidor sin asociarlo a un puerto, pudiendo especificar adicionalmente el puerto (un valor de 0 indicaría cualquier puerto libre), el número de conexiones simultáneas (backlog) y la dirección IP en la que debe escuchar (si no se especifica, escucharía en todas las direcciones del equipo).

- `Socket accept()`: Espera una conexión y la acepta cuando llega. La conexión se encapsula en un objeto `Socket` que podemos emplear para enviar/recibir información.
  - `void bind(SocketAddressEndpoint dir) / bind(SocketAddressEndpoint dir, int backlog)`: Asocia el socket a una IP y puerto encapsulados en un objeto `SocketAddressEndpoint`.
  - `void close()`: Cierra el socket.
  - `InetAddress getInetAddress() / int getLocalPort() / SocketAddress getLocalSocketAddress()`: Retorna la dirección del socket, el puerto y la dirección y el puerto al que está conectado (null si no se ha enlazado todavía en el constructor o con bind).
  - `boolean isBound() / boolean isConnected()`: Indica si el socket está enlazado/cerrado.

## Socket (java.net.Socket)

**La clase `Socket` identifica la parte cliente de una conexión y permite enviar/recibir información sobre la misma.**

Algunos de los métodos más interesantes:

- **Constructores:**
  - Socket ()
  - Socket (InetAddress dir, int puerto)
  - Socket (InetAddress dirS, int puertoS, InetAddress dirC, int puertoC)
  - Socket (Proxy p)
  - Socket (String dir, int puerto)
  - Socket (String dirS, int puertoS, InetAddress dirC, int puertoC)

Permite crear un socket y definir la IP/puerto del servidor y/o del cliente. Adicionalmente se puede especificar el proxy que queremos emplear.

- void bind(SocketAddressEndpoint dir): Asocia el socket a una IP y puerto encapsulados en un objeto `SocketAddressEndpoint`.
  - void close(): Cierra el socket.
  - void connect(SocketAddressEndpoint dir) / void connect(SocketAddressEndpoint dir, int timeout): Intenta conectarse con el servidor especificando opcionalmente definiendo un timeout (en milisegundos).
  - InetAddress getInetAddress() / SocketAddress getLocalAddress() / SocketAddress getRemoteSocketAddress() / int getPort() / int getLocalPort() / SocketAddress getLocalSocketAddress(): Retorna la dirección del servidor, del cliente, del servidor, el puerto remoto y local y la dirección y el puerto al que está conectado.

- `void setkeepAlive(boolean keep) / boolean getKeepAlive()`: Sigue/recupera si se permite el uso de conexiones persistentes. Por defecto las conexiones TCP/IP no se mantienen en cada petición. A partir de HTTP/1.1 se puede solicitar al servidor que se mantenga la conexión abierta entre peticiones.
- `InputStream getInputStream() / OutputStream getOutputStream()`: Obtiene las corrientes de datos de lectura/escritura para comunicarse con el otro extremo.
- `boolean isBound() / boolean isConnected()`: Indica si el socket está enlazado/cerrado/conectado.
- `void shutdownInput() / void shutdownOutput()`: Indica que ya no se quieren leer/escibir más datos. En el segundo caso, además, finalizan la conexión.

## Programación Multihilo (Threads, Runnable)

Las aplicaciones cliente/servidor basadas en sockets son un ejemplo de aplicaciones multihilo en las que un servidor debe hacer frente a múltiples conexiones simultáneas. La única forma de conseguirlo es que, cada vez que llegue una nueva conexión, se cree un nuevo hilo de ejecución en el que se establezca la comunicación con el cliente. Del lado cliente también puede ser interesante ejecutarlo en un hilo de ejecución diferente del hilo principal al objeto de no bloquear la aplicación.

Para que un código se ejecute en un nuevo hilo tenemos dos opciones:

- Crear una clase que extienda `Thread` y redefinir el método `run()`
- Crear una clase que implemente `Runnable` y redefinir `run()`

Una vez hecho basta con crear un objeto de la clase y ejecutar sobre él el método `start()`.

### Ejemplo: Aplicación de Chat Cliente/Servidor

Veamos un ejemplo de aplicación de mensajes cliente/servidor. Cualquier cliente podrá enviar un mensaje al servidor que lo reenviará al resto de los clientes. Cuando un cliente se conecte por primera vez enviará la cadena `100-nombre` y cuando se desconecte enviará la cadena `300-nombre`. Cada mensaje enviado se enviará con `200-mensaje`.

La aplicación servidora dispondrá de una clase `Conexion` (que será un `Thread`) a la que se le pasarán los datos de la conexión (socket y corrientes de datos) cada vez que se conecte un nuevo cliente. Dicha clase es la encargada de leer los mensajes y tendrá un método para enviar mensajes al cliente.

La propia clase servidora almacenará las conexiones en una lista y dispondrá de un método público para enviar un mensaje a todos los clientes y otro para eliminar una conexión de la lista. Ambos métodos serán llamados desde cada cliente de modo que, cuando alguna de las conexiones reciba un mensaje del cliente, decida qué hacer y se lo pase a la clase servidora para que lo envíe a todas las conexiones.

#### Código del Servidor

```
package org.zabalburu.servidoresocket;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ServidorSocket {
 private static List<Conexion> conexiones = new ArrayList<>();

 public static void enviarMensaje(String mensaje) {
 System.out.println("Enviando " + mensaje);
 conexiones.forEach(c -> {
 try {
 c.enviarMensaje(mensaje);
 } catch (IOException ex) {
 Logger.getLogger(ServidorSocket.class.getName())
 .log(Level.SEVERE, null, ex);
 }
 });
 }

 public static void desconectar(Conexion c) {
 conexiones.remove(c);
 }

 public static void main(String[] args) {
 try {
 ServerSocket server = new ServerSocket(20000);
 System.out.println("Esperando conexiones...");

 while (true) {
 Socket sck = server.accept();
 System.out.println("Conectado cliente desde " + sck.getRemoteSocketAddress());
 Conexion c = new Conexion(
 sck,
 new DataInputStream(sck.getInputStream()),
 new DataOutputStream(sck.getOutputStream())
);
 conexiones.add(c);
 c.start();
 }
 } catch (IOException ex) {
 Logger.getLogger(ServidorSocket.class.getName())
 .log(Level.SEVERE, null, ex);
 }
 }
}
```

```

class Conexion extends Thread {
 private Socket sck;
 private DataInputStream dis;
 private DataOutputStream dos;
 private int id;
 private String nombre;

 public void setId(int id) {
 this.id = id;
 }

 private static int numConexiones = 0;

 Conexion(Socket sck, DataInputStream dis, DataOutputStream dos) {
 this.sck = sck;
 this.dis = dis;
 this.dos = dos;
 this.id = ++numConexiones;
 }

 public void enviarMensaje(String mensaje) throws IOException {
 dos.writeUTF(mensaje);
 }

 public void cerrarConexion() {
 try {
 sck.close();
 } catch (IOException ex) {
 // Manejo de excepción
 }
 }

 @Override
 public void run() {
 boolean salir = false;
 while (!salir) {
 try {
 String linea = dis.readUTF();
 System.out.println(linea);

 if (linea.startsWith("100--")) {
 this.nombre = linea.substring(4);
 ServidorSocket.enviarMensaje("Se ha conectado " + this.nombre);
 } else if (linea.startsWith("200--")) {
 String mensaje = linea.substring(4);
 ServidorSocket.enviarMensaje(this.nombre + " : " + mensaje);
 } else if (linea.startsWith("300--")) {
 ServidorSocket.enviarMensaje("Se ha desconectado " + this.nombre);
 ServidorSocket.desconectar(this);
 salir = true;
 }
 } catch (IOException ex) {
 // Manejo de excepción
 }
 }
 try {
 sck.close();
 } catch (IOException ex) {
 // Manejo de excepción
 }
 }
}

```

#### Código del Cliente

La clase cliente leerá/escribirá información. Dado que los dos procesos se han de hacer en paralelo, emplearemos dos hilos de ejecución:

```

package org.zabalburu.clientesocket;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ClienteSocket {
 public static void main(String[] args) {
 try {
 Scanner sc = new Scanner(System.in);
 Socket sck = new Socket("localhost", 20000);
 System.out.println("Conectado a " + sck.getRemoteSocketAddress());
 DataInputStream dis = new DataInputStream(sck.getInputStream());
 DataOutputStream dos = new DataOutputStream(sck.getOutputStream());
 System.out.print("Nombre: ");
 String nombre = sc.nextLine();
 dos.writeUTF("100--" + nombre);
 dos.flush();

 // Hilo para recibir mensajes
 Thread t1 = new Thread() {
 @Override

```

```

 public void run() {
 super.run();
 while (true) {
 try {
 String linea = dis.readUTF();
 System.out.println(linea);
 } catch (IOException ex) {
 // Manejo de excepción
 }
 }
 };
 }

 // Hilo para enviar mensajes (con expresiones Lambda => new Thread(Runnable))
 Thread t2 = new Thread(() -> {
 while (true) {
 try {
 System.out.println(nombre + " : Mensaje (en blanco para salir)");
 String resp = sc.nextLine();
 if (resp.isEmpty()) {
 dos.writeUTF("300--" + nombre);
 sck.close();
 System.exit(0);
 } else {
 dos.writeUTF("200--" + resp);
 }
 } catch (IOException ex) {
 // Manejo de excepción
 }
 }
 });
}

```

### Configuración Maven

Modificamos el pom.xml de ambas clases para obtener un archivo jar ejecutable:

#### Servidor (pom.xml):

```

<?xml version="1.0" encoding="UTF-8"?>
<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
<modelVersion>4.0.0</modelVersion>
<groupId>org.zabalburu</groupId>
<artifactId>ServidorSocket</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>18</maven.compiler.source>
 <maven.compiler.target>18</maven.compiler.target>
 <exec.mainClass>org.zabalburu.servidoresocket.ServidorSocket</exec.mainClass>
</properties>
<build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-jar-plugin</artifactId>
 <version>3.2.0</version>
 <configuration>
 <archive>
 <manifest>
 <addClasspath>true</addClasspath>
 <mainClass>org.zabalburu.servidoresocket.ServidorSocket</mainClass>
 </manifest>
 </archive>
 </configuration>
 </plugin>
 </plugins>
</build>
</project>

```

#### Cliente (pom.xml):

```

<?xml version="1.0" encoding="UTF-8"?>
<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
<modelVersion>4.0.0</modelVersion>
<groupId>org.zabalburu</groupId>
<artifactId>ClienteSocket</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>18</maven.compiler.source>
 <maven.compiler.target>18</maven.compiler.target>
 <exec.mainClass>org.zabalburu.clientesocket.ClienteSocket</exec.mainClass>
</properties>

```

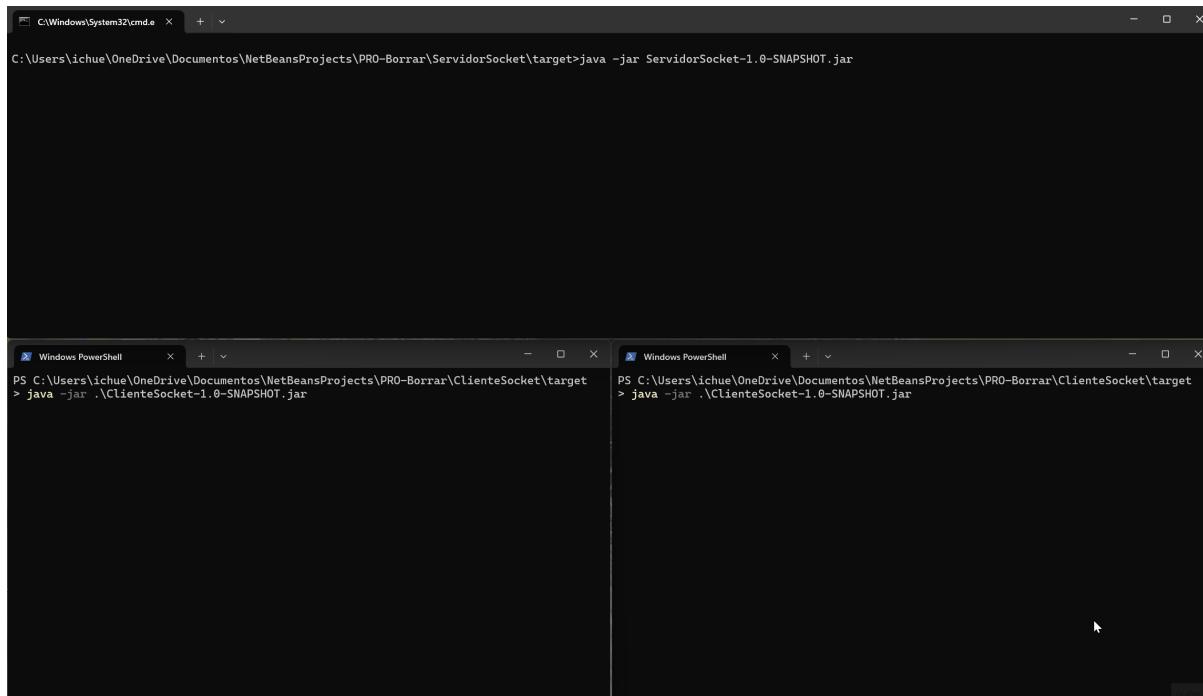
```

</properties>
<build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-jar-plugin</artifactId>
 <version>3.2.0</version>
 <configuration>
 <archive>
 <manifest>
 <addClasspath>true</addClasspath>
 <mainClass>org.zabalburu.clientesocket.ClienteSocket</mainClass>
 </manifest>
 </archive>
 </configuration>
 </plugin>
 </plugins>
</build>
</project>

```

Ahora ejecutamos Build en el menú contextual de ambos proyectos con lo que creará un archivo jar en la carpeta target del proyecto que podremos ejecutar desde la línea de comandos.

La salida (ejecutamos desde la línea de comandos para tener varios clientes):



#### 4.9. Acceso a Internet conexión a API REST

### API REST

Un **API (Application Program Interface)** es un servicio que permite comunicarse con las aplicaciones para intercambiar información. **REST (Transferencia de Estado Representacional)** es una arquitectura software que propone emplear el protocolo **HTTP** para facilitar ese intercambio de información. Un **API REST** permite el acceso a sus servicios a través de **URLs** empleando los comandos, cabeceras y respuestas **HTTP**.

**NOTA:** **HTTP** es el protocolo para la transferencia de hipertexto y es el estándar que define cómo se deben comunicar un cliente y un servidor a través de una red (Internet).

La idea es que una aplicación permite a otras comunicarse con ella a través del protocolo **HTTP**. Dado que todos los lenguajes de programación disponen de herramientas para hacer peticiones **HTTP**, esto permite que aplicaciones diferentes (y creadas con diferentes lenguajes de programación) puedan comunicarse entre sí.

Es importante recalcar que **REST** no es un estándar, por lo que no todo el mundo lo implementa igual.

### Comandos HTTP

**REST** emplea los comandos **HTTP** para las peticiones de una aplicación cliente a una servidora. Los comandos más importantes son:

- **GET** : Se emplea para solicitar información al servidor
- **POST** : Se emplea para añadir nueva información al servidor
- **PUT / PATCH** : Se emplea para modificar información en el servidor
- **DELETE** : Se emplea para eliminar información del servidor

### Respuestas HTTP

Ante una petición **REST**, el servidor realizará la operación solicitada y devolverá un código de respuesta **HTTP** y, opcionalmente, la información en el cuerpo del documento.

- **2XX** : La operación solicitada se ha gestionado con éxito (por ejemplo el código **200** que significa **OK**)
- **3XX** : El servidor pide una redirección
- **4XX** : La operación no se ha realizado por algún problema en el cliente (por ejemplo **404 Not Found** cuando se intenta localizar un recurso que no existe)
- **5XX** : La operación no se ha realizado por algún problema del servidor

## Cabeceras HTTP

Tanto el cliente como el servidor pueden enviar cabeceras en la petición y en la respuesta. Estas cabeceras son pares de valores clave/valor que indican cosas como el tipo de contenido que se solicita/envía, información de autenticación del cliente, fecha, etc.

## Parámetros

Los parámetros pueden ir indicados en la **URL** separados de ésta por **?** y unidos entre sí por el símbolo **&**. Se indican mediante la expresión **parámetro=valor**. También es posible enviar los parámetros dentro del cuerpo de la petición.

## JSON

Cuando solicitamos un recurso del servidor (o le enviamos información), el servidor y el cliente deben acordar la forma en la que se va a enviar dicha información. Básicamente hay dos opciones:

- Emplear **XML** : Era la opción más habitual hasta la aparición de **JSON**
- Emplear **JSON** : **Javascript Object Notation** permite representar un objeto de una manera sencilla, lo que ha hecho que sea el estándar de facto para las **API REST**

**Java SE** no tiene soporte nativo para **JSON**, así que habrá que añadir las librerías necesarias para poder interpretar la información en dicho formato. Existen múltiples librerías **Java** para trabajar con **JSON**. Nosotros emplearemos la librería **json** de [json.org](http://json.org).

Un ejemplo de lo que retorna una llamada a un **API** (del tiempo) en **JSON**:

```

{
 "latitude": 43.26,
 "longitude": -2.92,
 "generationtime_ms": 0.45299530029296875,
 "utc_offset_seconds": 3600,
 "timezone": "Europe/Madrid",
 "timezone_abbreviation": "CET",
 "elevation": 59,
 "daily_units": {
 "time": "iso8601",
 "temperature_2m_max": "%c"
 },
 "daily": [
 {
 "time": [
 "2023-03-03",
 "2023-03-04",
 "2023-03-05",
 "2023-03-06",
 "2023-03-07",
 "2023-03-08",
 "2023-03-09"
],
 "temperature_2m_max": [
 9.9,
 11.7,
 12.4,
 12.3,
 13.5,
 17.4,
 18.1
]
 }
]
}

```

En **JSON** tenemos los siguientes posibles elementos:

- **Objeto** : Un objeto **JSON** se encierra entre llaves **{}** y contiene sus propiedades. Las propiedades están formadas por un par "clave": valor
- **Colección** : Una colección **JSON** es un conjunto de valores (pueden ser objetos) y se encierra entre corchetes **[]**

En el ejemplo vemos que recibimos un objeto que tiene:

- **propiedades simples** : latitude, longitude...
- **propiedades que son objetos** : daily\_units (con las propiedades time y temperature\_min\_2)
- **propiedades que son colecciones** : las propiedades time y temperature\_min\_2 del objeto daily

## Creación del Proyecto con Dependencias

Como ya hemos visto, necesitamos las librerías de **JSON** para poder hacer la aplicación. Una librería no es más que una colección de clases y paquetes que puedo emplear en mi aplicación. Para poder utilizar las librerías tenemos dos opciones:

- Buscar las librerías (archivos **jar**) en Internet y añadirlas a nuestro proyecto. Este sistema tiene la pega de que, a veces, unas librerías dependen de otras, con lo que hay que ir viendo qué errores da la aplicación para buscar las librerías de las clases que nos faltan. Además, este sistema es muy engorroso si queremos cambiar de versión de una librería.
- Emplear un gestor de dependencias (**Maven**, **Gradle**...). Un gestor de dependencias permite especificar qué librería/versión queremos y se encarga de descargar la librería y todas las librerías de las que dependa (dependencias). Para ello, se registran las librerías con sus dependencias en un repositorio de Internet y se descargan de allí.

Nosotros vamos a emplear **Maven**.

Para especificar una dependencia en **Maven** tenemos que incluir cierta información en el fichero **pom.xml**. Y, para localizar dicha información, lo más cómodo es ir al repositorio de **Maven** más importante: [Maven Central](http://Maven_Central). En esa página, simplemente introducimos la información de la dependencia (librería) que queremos emplear:

Maven Repository: json.java

mvnrepository.com/search?q=json+java

**MVN REPOSITORY**

Repository

- Central 47.5K
- Sonatype 12.0K
- Spring Plugins 5.6K
- Spring Lib M 5.2K
- JCenter 2.6K
- Clojars 1.2K
- Spring Lib Release 1.1K
- IBiblio 1.0K

Group

- com.github 5.5K
- io.github 3.0K
- com.aliyun 1.3K
- org.apache 1.2K
- com.google 686
- io.opentelemetry 641
- software.amazon 496
- cn.com.antcloud 458

Category

- Android Package 743
- Java Spec 546

Found 55714 results

Sort: relevance | popular | newest

1. **JSON In Java**  
org.json » json

JSON is a light-weight, language independent, data interchange format. See <http://www.JSON.org/> The files in this package implement JSON encoders/decoders in Java. It also includes the capability to convert between JSON and XML, HTTP headers, Cookies, and CDL. This is a reference implementation. There is a large number of JSON packages in Java. Perhaps someday the Java community will standardize on one. Until then, choose carefully.

Last Release on Feb 27, 2023

2. **Gson**  
com.google.code.gson » gson

Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object.

Last Release on Jan 6, 2023

3. **JSON.simple**  
com.googlecode.json-simple » json-simple

A simple Java toolkit for JSON

Last Release on Mar 21, 2012

4. **FastJson1 Compatible**  
com.alibaba » fastjson

Fastjson is a JSON processor /JSON parser & JSON generator written in Java

5,148 usages Public

20,062 usages Apache

1,993 usages Apache

5,783 usages Apache

Categories | Popular | Contact Us

ADS VIA GARDEN

Indexed Repositories (1885)

- Central
- Atlassian
- Sonatype
- Hortonworks
- Spring Plugins
- Spring Lib M
- JCenter
- JBossEAI
- Atlassian Public
- BeDataDriven

Popular Tags

aar android apache api

Seleccionamos la librería que nos interesa y, en la siguiente pantalla, la versión:

Maven Repository: org.json » json

mvnrepository.com/artifact/org.json/json

**MVN REPOSITORY**

Indexed Artifacts (32.5M)

Home > org.json > json

**JSON In Java**

JSON is a light-weight, language independent, data interchange format. See <http://www.JSON.org/> The files in this package implement JSON encoders/decoders in Java. It also includes the capability to convert between JSON and XML, HTTP headers, Cookies, and CDL. This is a reference implementation. There is a large number of JSON packages in Java. Perhaps someday the Java community will standardize on one. Until then, choose carefully.

License: Public

Categories: JSON Libraries

Tags: json format

Ranking: #89 in MvnRepository (See Top Artifacts)  
#5 in JSON Libraries

Used By: 5,148 artifacts

| Central (23) | Jahia (1) | Redhat GA (2) | Redhat EA (1) | AKSW (1) | Jena Bio (1) | PentahoOmni (31) | MyGrid (1) | PNT (61) | OW2 Public (1) | ICM (2) |
|--------------|-----------|---------------|---------------|----------|--------------|------------------|------------|----------|----------------|---------|
| 20230227     |           |               |               |          |              |                  |            |          |                |         |
| 20220924     |           |               |               |          |              |                  |            |          |                |         |
| 20220320     |           |               |               |          |              |                  |            |          |                |         |
| 20211205     |           |               |               |          |              |                  |            |          |                |         |
| 20210307     |           |               |               |          |              |                  |            |          |                |         |
| 20201115     |           |               |               |          |              |                  |            |          |                |         |
| 20200518     |           |               |               |          |              |                  |            |          |                |         |

Version Vulnerabilities Repository Usages Date

20230227 Central 3 Feb 27, 2023

20220924 Central 258 Sep 24, 2022

20220320 Central 288 Mar 20, 2022

20211205 Central 188 Dec 05, 2021

20210307 Central 348 Mar 09, 2021

20201115 Central 227 Nov 15, 2020

20200518 Central 258 May 22, 2020

Google Cloud

Auto-reconocimiento de voz

Conversion voz-a-texto basada en machine learning. Prueba Google Cloud gratis hoy mismo.

Visitar sitio

Una vez seleccionada la versión elegimos Maven:

Maven Repository: org.json » json

mvnrepository.com/artifact/org.json/json/20230227

**MVN REPOSITORY**

Indexed Artifacts (32.5M)

Home > org.json > 20230227

**JSON In Java » 20230227**

JSON is a light-weight, language independent, data interchange format. See <http://www.JSON.org/> The files in this package implement JSON encoders/decoders in Java. It also includes the capability to convert between JSON and XML, HTTP headers, Cookies, and CDL. This is a reference implementation. There is a large number of JSON packages in Java. Perhaps someday the Java community will standardize on one. Until then, choose carefully.

License: Public

Categories: JSON Libraries

Tags: json format

HomePage: <https://github.com/douglascrockford/JSON-Java>

Date: Feb 27, 2023

Files: pom (6 KB) bundle (70 KB) View All

Repositories: Central

Ranking: #89 in MvnRepository (See Top Artifacts)  
#5 in JSON Libraries

Used By: 5,148 artifacts

```
<!-- https://mvnrepository.com/artifact/org.json/json -->
<dependency>
 <groupId>org.json</groupId>
 <artifactId>json</artifactId>
 <version>20230227</version>
</dependency>
```

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr |

ADS VIA GARDEN

LG OLED evo

Disfruta de 5 años de garantía

Panel OLED Serie G2

PREMIUM OLED 5 YEAR LIMITED PANEL WARRANTY

Este producto es el panel OLED de los televisores LG OLED modelo G2. La garantía es de 5 años para la sustitución del panel OLED en caso de fallos de garantía legal o 2 años de garantía comercial. La garantía comercial es para la sustitución del panel OLED en caso de fallos de garantía legal. La garantía no cubre la sustitución del panel OLED desgastado, desplazamiento, muro de fondo, etc. Consulte las condiciones de garantía en el sitio web de LG.

+ INFO: <https://www.lg.com/es/support/televisiones/condiciones-de-garantia>

LG OLED evo

Y ahí tenemos la información a añadir al fichero (basta con hacer clic encima para que se copie al portapapeles).

El fichero **pom.xml** quedaría así:

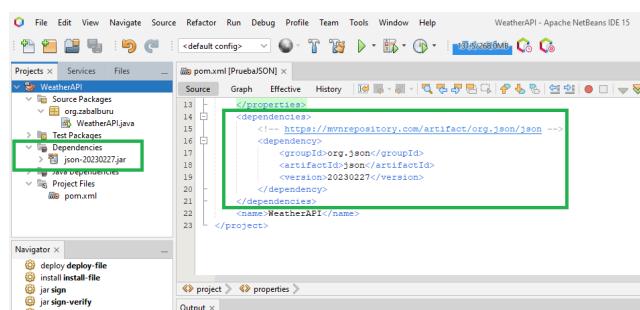
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>
 <groupId>com.mycompany</groupId>
 <artifactId>PruebaJSON</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>jar</packaging>

 <properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>18</maven.compiler.source>
 <maven.compiler.target>18</maven.compiler.target>
 <exec.mainClass>org.zabalburu.WeatherAPI</exec.mainClass>
 </properties>

 <dependencies>
 <!-- https://mvnrepository.com/artifact/org.json/json -->
 <dependency>
 <groupId>org.json</groupId>
 <artifactId>json</artifactId>
 <version>20230227</version>
 </dependency>
 </dependencies>
</project>
```

Al guardar el fichero se deberían descargar las librerías (podemos verlas en la carpeta *Dependencies* de la vista *Projects*):



## Clase HttpURLConnection

Esta clase nos permite establecer conexiones con una **URL** a través del protocolo **HTTP**.

### Establecer conexión

Para establecer una conexión, el proceso habitual es crear un objeto **URL** y emplear su método `getConneciton()`:

```
URL url = new URL("https://api.open-meteo.com/v1/forecast?" + query);
HttpURLConnection con = (HttpURLConnection) url.openConnection();
```

Adicionalmente podemos especificar el método **HTTP** que queremos emplear:

```
con.setRequestMethod("GET");
```

### Especificar cabeceras

Las cabeceras a enviar se indican con el método `setRequestProperty()`:

```
con.setRequestProperty("Content-Type", "application/json");
```

### Parámetros en la URL

Para indicar los parámetros necesitamos generar el texto en el formato que se espera del protocolo. Además, tanto los nombres como los valores de los parámetros deben ir codificados en **UTF-8**.

### Cuerpo del documento

Si necesitamos enviar información en el cuerpo del documento (habitualmente en formato **JSON**), deberíamos hacer lo siguiente:

1. Crear un `JSONObject`:

```
java JSONObject requestBody = new JSONObject();
```

1. Asignarle los valores de las propiedades (codificados):

```
java requestBody.put("propiedad", URLEncoder.encode(valor,"UTF-8"));
```

1. Abrir una corriente de datos de salida en la conexión:

```
java con.setDoOutput(true);

1. Obtener la corriente de datos de salida y escribir los datos como una cadena:

java DataOutputStream wr = new DataOutputStream(con.getOutputStream()); wr.writeBytes(requestBody.toString()); wr.flush(); wr.close();
```

## Leer la respuesta

Para leer la respuesta:

1. Abrimos la corriente de entrada de la conexión:

```
java BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
```

1. Vamos leyendo de la corriente de datos mientras haya información y la añadimos a un `StringBuilder`:

```
java String inputLine; StringBuilder response = new StringBuilder(); while ((inputLine = in.readLine()) != null) { response.append(inputLine); }
in.close();
```

1. Si está en formato **JSON**, la convertimos en un `JSONObject`:

```
java JSONObject jsonResponse = new JSONObject(response.toString());
```

Ahora podemos acceder a la información con los métodos `get` de dicha clase.

## Ejemplo

Vamos a conectarnos al **API Open-Meteo** del ejemplo para obtener las temperaturas máximas y mínimas en Bilbao en los próximos días. La **URL** que vamos a emplear es:

```
https://api.open-meteo.com/v1/forecast?latitude=43.26271&longitude=-2.92528&daily=temperature_2m_min,temperature_2m_max,weather_code&timezone=auto
```

- **latitude, longitude** : Las de Bilbao
- **daily** : Queremos los próximos días
- **temperature\_2m\_min** : Temperatura mínima a 2 mts. sobre el nivel del mar
- **temperature\_2m\_max** : Temperatura máxima a 2 mts. sobre el nivel del mar
- **weather\_code** : Un código con el tiempo más habitual en el día

Nos retornará un **JSON** como el siguiente:

```
{
 "latitude":43.26,
 "longitude":-2.9300003,
 "generationtime_ms":0.32830238342285156,
 "utc_offset_seconds":3600,
 "timezone":"Europe/Madrid",
 "timezone_abbreviation":"GMT+1",
 "elevation":21.0,
 "daily_units":{
 "time":"iso8601",
 "temperature_2m_min":">C",
 "temperature_2m_max":">C",
 "weather_code":"wmo code"
 },
 "daily":{
 "time":[
 "2025-03-10",
 "2025-03-11",
 "2025-03-12",
 "2025-03-13",
 "2025-03-14",
 "2025-03-15",
 "2025-03-16"
],
 "temperature_2m_min":[
 11.0,
 11.1,
 9.1,
 7.7,
 6.2,
 1.7,
 4.8
],
 "temperature_2m_max":[
 18.9,
 16.2,
 15.9,
 10.5,
 11.2,
 15.6,
 13.1
],
 "weather_code":[
 3,
 63,
 61,
 53,
 61,
 80,
 45
]
 }
}
```

```
}
```

Que tendremos que procesar.

El código:

```
package org.zabalburu.weatherapp;

import org.json.JSONObject;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.List;
import javax.swing.JLabel;
import javax.swing.JOptionPane;

public class WeatherApp {
 // URL de la API de OpenMeteo para Bilbao
 private static final String API_URL = "https://api.open-meteo.com/v1/forecast?latitude=43.26271&longitude=-2.92528&daily=temperature_2m_min,temperature_2m_max,weather_code";

 public static void main(String[] args) {
 try {
 // Realizamos la solicitud
 JSONObject jsonObject = sendGetRequest(API_URL);

 // Obtenemos los datos del tiempo
 JSONObject daily = jsonObject.getJSONObject("daily");
 List times = daily.getJSONArray("time").toList();
 List temp_min = daily.getJSONArray("temperature_2m_min").toList();
 List temp_max = daily.getJSONArray("temperature_2m_max").toList();
 List weather_code = daily.getJSONArray("weather_code").toList();

 // Mostrar los datos del tiempo (en un HTML "sencillo")
 String listado = "";
 listado += """
 <html>
 <style>
 table{
 border: 1px solid #90D0D0;
 width:500px;
 }
 th {
 background-color: #60a0a0;
 text-align:center;
 border-color: #90D0D0;
 }
 td {
 background-color: #96D4D4;
 text-align:center;
 border-color: #90D0D0;
 font-weight: normal;
 }
 h1 {
 text-align:center;
 }
 </style>
 <body>
 <h1>Pronóstico del tiempo en Bilbao para los próximos días</h1>
 <table>
 <tr><th>Dia</th><th>T° máxima</th><th>T° Mínima</th><th>Pronóstico</th></tr>
 """;
 for (int i = 0; i < times.size(); i++) {
 listado += """
 <tr><td>%s</td><td>%s</td><td>%s</td><td>%s</td></tr>
 """, formatted(
 changeTime(times.get(i).toString()),
 temp_max.get(i),
 temp_min.get(i),
 getWeather(weather_code.get(i).toString())
);
 }
 listado += "</table></body></html>";

 JOptionPane.showMessageDialog(null, new JLabel(listado),
 "Pronóstico",
 JOptionPane.PLAIN_MESSAGE);
 } catch (Exception e) {
 e.printStackTrace();
 }
 }

 // Método para enviar una solicitud HTTP GET
 private static JSONObject sendGetRequest(String url) throws Exception {
 HttpURLConnection connection = null;
 try {
 // Crear la conexión
 URL apiUrl = new URL(url);
 connection = (HttpURLConnection) apiUrl.openConnection();
 connection.setRequestMethod("GET");

```

```

// Leer la respuesta
BufferedReader reader = new BufferedReader(
 new InputStreamReader(connection.getInputStream())
);
StringBuilder response = new StringBuilder();
String line;
while ((line = reader.readLine()) != null) {
 response.append(line);
}
reader.close();
return new JSONObject(response.toString());
} finally {
 if (connection != null) {
 connection.disconnect();
 }
}
}

private static String changeTime(String time) {
 return time.substring(8, 10) + "/" + time.substring(5, 7) + "/" + time.substring(0, 4);
}

private static String getWeather(String weather_code) {
 String description;
 String imageUrl;

 switch (weather_code) {
 case "0":
 description = "Soleado";
 imageUrl = "http://openweathermap.org/img/wn/01d@2x.png";
 break;
 case "1":
 description = "Principalmente soleado";
 imageUrl = "http://openweathermap.org/img/wn/01d@2x.png";
 break;
 case "2":
 description = "Parcialmente nublado";
 imageUrl = "http://openweathermap.org/img/wn/02d@2x.png";
 break;
 case "3":
 description = "Nublado";
 imageUrl = "http://openweathermap.org/img/wn/03d@2x.png";
 break;
 case "45":
 description = "Nebuloso";
 imageUrl = "http://openweathermap.org/img/wn/50d@2x.png";
 break;
 case "48":
 description = "Niebla helada";
 imageUrl = "http://openweathermap.org/img/wn/50d@2x.png";
 break;
 case "51":
 description = "Llovizna ligera";
 imageUrl = "http://openweathermap.org/img/wn/09d@2x.png";
 break;
 case "53":
 description = "Llovizna";
 imageUrl = "http://openweathermap.org/img/wn/09d@2x.png";
 break;
 case "55":
 description = "Llovizna intensa";
 imageUrl = "http://openweathermap.org/img/wn/09d@2x.png";
 break;
 case "56":
 description = "Llovizna helada ligera";
 imageUrl = "http://openweathermap.org/img/wn/09d@2x.png";
 break;
 case "57":
 description = "Llovizna helada";
 imageUrl = "http://openweathermap.org/img/wn/09d@2x.png";
 break;
 case "61":
 description = "Lluvia ligera";
 imageUrl = "http://openweathermap.org/img/wn/10d@2x.png";
 break;
 case "63":
 description = "Lluvia";
 imageUrl = "http://openweathermap.org/img/wn/10d@2x.png";
 break;
 case "65":
 description = "Lluvia intensa";
 imageUrl = "http://openweathermap.org/img/wn/10d@2x.png";
 break;
 case "66":
 description = "Lluvia helada ligera";
 imageUrl = "http://openweathermap.org/img/wn/10d@2x.png";
 break;
 case "67":
 description = "Lluvia helada";
 imageUrl = "http://openweathermap.org/img/wn/10d@2x.png";
 break;
 case "71":
 description = "Nieve ligera";
 }
}

```

```

 imageUrl = "http://openweathermap.org/img/wn/13d@2x.png";
 break;
 case "73":
 description = "Nieve";
 imageUrl = "http://openweathermap.org/img/wn/13d@2x.png";
 break;
 case "75":
 description = "Nieve intensa";
 imageUrl = "http://openweathermap.org/img/wn/13d@2x.png";
 break;
 case "77":
 description = "Granos de nieve";
 imageUrl = "http://openweathermap.org/img/wn/13d@2x.png";
 break;
 case "80":
 description = "Chubascos ligeros";
 imageUrl = "http://openweathermap.org/img/wn/09d@2x.png";
 break;
 case "81":
 description = "Chubascos";
 imageUrl = "http://openweathermap.org/img/wn/09d@2x.png";
 break;
 case "82":
 description = "Chubascos intensos";
 imageUrl = "http://openweathermap.org/img/wn/09d@2x.png";
 break;
 case "83":
 description = "Chubascos de nieve ligeros";
 imageUrl = "http://openweathermap.org/img/wn/13d@2x.png";
 break;
 case "86":
 description = "Chubascos de nieve";
 imageUrl = "http://openweathermap.org/img/wn/13d@2x.png";
 break;
 case "95":
 description = "Tormenta";
 imageUrl = "http://openweathermap.org/img/wn/11d@2x.png";
 break;
 case "96":
 description = "Tormentas ligeras con granizo";
 imageUrl = "http://openweathermap.org/img/wn/11d@2x.png";
 break;
 case "99":
 description = "Tormenta con granizo";
 imageUrl = "http://openweathermap.org/img/wn/11d@2x.png";
 break;
 default:
 description = "Desconocido";
 imageUrl = "";
 break;
 }
}

return "
" + description;
}
}

```

La salida:

Pronóstico del tiempo en Bilbao para los próximos días			
Día	Tº máxima	Tº Mínima	Pronóstico
11/03/2025	15.6	11.8	Lluvia ligera
12/03/2025	15.2	10.5	Lluviosa intensa
13/03/2025	12.1	7.1	Lluvia
14/03/2025	10.2	5.3	Lluvia ligera
15/03/2025	9.9	5.4	Lluvia ligera
16/03/2025	11.1	2.6	Nebuloso
17/03/2025	18.0	3.5	Lluvia ligera

OK

#### 4.10. Hacer una aplicación ejecutable

##### Convertir una Aplicación Maven Java en Ejecutable

Se puede convertir una aplicación **Maven** en un **.jar** ejecutable modificando el fichero **pom.xml** para añadir el **plugin maven-assembly-plugin**. En el ejemplo anterior, el fichero quedaría así:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>org.zabalburu</groupId>
 <artifactId>WeatherAPI</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>jar</packaging>

 <properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>18</maven.compiler.source>
 <maven.compiler.target>18</maven.compiler.target>
 </properties>

 <dependencies>
 <!-- https://mvnrepository.com/artifact/org.json/json -->
 <dependency>
 <groupId>org.json</groupId>
 <artifactId>json</artifactId>
 <version>20230227</version>
 </dependency>
 </dependencies>

 <name>WeatherAPI</name>

 <build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-assembly-plugin</artifactId>
 <executions>
 <execution>
 <phase>package</phase>
 <goals>
 <goal>single</goal>
 </goals>
 <configuration>
 <archive>
 <manifest>
 <mainClass>org.zabalburu.weatherapp.WeatherApp</mainClass>
 </manifest>
 </archive>
 <descriptorRefs>
 <descriptorRef>jar-with-dependencies</descriptorRef>
 </descriptorRefs>
 </configuration>
 </execution>
 </executions>
 </plugin>
 </plugins>
 </build>
</project>
```

Es muy importante indicar en la opción **mainClass** el **nombre completo** de la clase que queremos que se ejecute. El siguiente paso es crear el proyecto con dependencias desde **NetBeans**.

Si ahora vamos a la carpeta **target** dentro del proyecto, podemos ejecutar la aplicación con el siguiente comando:

```
\target> java -jar WeatherAPI-1.0-SNAPSHOT-jar-with-dependencies.jar
```

**NOTA:** El nombre del **.jar** se puede cambiar sin problemas para que no sea tan largo (por ejemplo, configurando la propiedad **finalName** en el plugin).

## 5. Interfaces de Usuario Swing

### 5.1. Introducción Creando la primera ventana

#### Creando la primera ventana

A la hora de diseñar aplicaciones gráficas, en las primeras versiones de **Java** se utilizaba el **AWT (Abstract Window Toolkit)** que era (y es) un conjunto de clases e interfaces que permitía desarrollar aplicaciones en ventanas, con botones, cuadros de texto, etiquetas, imágenes, menús, etc., y que se gestionaba mediante eventos. El problema principal con **AWT** es que no era todo lo portable que debiera ser. Además, el conjunto de componentes disponibles era muy simple, por lo que sólo se podían diseñar interfaces de usuario muy sencillas.

Para solucionar este problema, *Sun*, junto con *Netscape*, diseñó un conjunto de nuevas clases orientadas al diseño de aplicaciones **Java**. A este nuevo grupo de clases e interfaces se le denominó **Swing**.

Es importante recalcar que **Swing** no sustituye por completo a **AWT**. De hecho, la gestión de eventos **Swing** es la gestión de eventos **AWT** implantada en la versión 1.1 de **Java**. Lo que ya no tiene mucho sentido emplear son los componentes gráficos **AWT**.

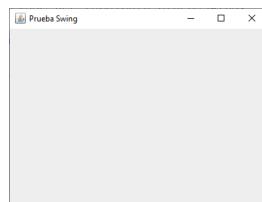
Las aplicaciones de tipo gráfico que vamos a desarrollar se ejecutan dentro de una ventana. En **Swing**, el componente básico de una ventana es un objeto de tipo `javax.swing.JFrame`. Podemos crear una ventana básica con las siguientes instrucciones:

```
import javax.swing.JFrame;

public class PruebaSwing {
 public static void main(String[] args) {
 JFrame ventana = new JFrame();
 ventana.setTitle("Prueba Swing");
 ventana.setSize(400, 300);
 ventana.setLocation(500, 200);
 ventana.setVisible(true);
 }
}
```

Como se puede ver, creamos una ventana, le ponemos un título, un tamaño y una posición, y la hacemos visible (por defecto, las ventanas se crean ocultas).

Si ejecutamos el código se mostrará algo así:



Como se puede ver, obtenemos una ventana funcional que podemos mover, redimensionar, minimizar, restaurar y cerrar.

**NOTA:** Cerrar la ventana no implica finalizar el programa. Por defecto, **Swing** oculta la ventana pero no finaliza la aplicación, por lo que tendremos que finalizarla manualmente (pulsando el botón rojo en la ventana *Output* de **NetBeans**). Esto es así porque, en una aplicación de múltiples ventanas, cerrar una no debe implicar finalizar la aplicación.

Este comportamiento lo podemos modificar con el método `setDefaultCloseOperation` de la clase `JFrame`, que admite los siguientes valores numéricos (especificados por constantes):

- **JFrame.HIDE\_ON\_CLOSE**  
Predeterminado. La ventana se oculta y la aplicación continúa en ejecución.
- **JFrame.DISPOSE\_ON\_CLOSE**  
Similar a la anterior, pero, si la ventana es la última que queda abierta, se finaliza la ejecución de la aplicación.
- **JFrame.EXIT\_ON\_CLOSE**  
Al pulsar el icono de cerrar, la aplicación finaliza.
- **JFrame.DO NOTHING ON CLOSE**  
Al pulsar el icono de cerrar, **Java** no hace nada. Si queremos cerrar la ventana (o finalizar la ejecución), deberemos hacerlo por código.

Modificamos el código para emplear `DISPOSE_ON_CLOSE` y comprobamos que ahora, al pulsar el botón de cierre, la ventana desaparece y, como no hay más ventanas, la aplicación finaliza.

Puede ser que queramos utilizar la misma ventana varias veces, por lo que es habitual que, en lugar de crear un objeto `JFrame` y modificar sus propiedades, lo que se suele hacer es crear una clase hija de `JFrame` en la que definamos sus propiedades. Luego emplearemos dicha clase para gestionar las ventanas.

Modificamos el ejemplo anterior:

```
import javax.swing.JFrame;

public class PruebaSwing extends JFrame {
 public PruebaSwing() {
 this.setTitle("Prueba Swing");
 this.setSize(400, 300);
 this.setLocation(500, 200);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 }

 public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
 }
}
```

El resultado, evidentemente, es el mismo.

### Centrando la ventana automáticamente

Es bastante habitual que queramos que nuestra ventana aparezca **centrada en pantalla**, independientemente del tamaño de la misma.

Para ello, necesitamos conocer las dimensiones de la pantalla en la que se ejecuta la aplicación. Disponemos de la clase **Toolkit** que nos permite acceder a información sobre la plataforma en la que se ejecuta **Java**. En concreto, el método para obtener las dimensiones de la ventana es:

```
Toolkit.getDefaultToolkit().getScreenSize()
```

Este método retorna un objeto de tipo `java.awt.Dimension`, una clase básica que dispone de dos propiedades enteras: `height` y `width`. Con esa información y el tamaño de la ventana, ya podemos centrarla:

```
import java.awt.Dimension;
import java.awt.Toolkit;
import javax.swing.JFrame;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400,300);

 public PruebaSwing() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba Swing");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 }

 public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
 }
}
```

Ahora la ventana aparecerá **centrada**.

Otra opción más simple es emplear el método `setLocationRelativeTo(null)`:

```
import java.awt.Dimension;
import java.awt.Toolkit;
import javax.swing.JFrame;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400, 300);

 public PruebaSwing() {
 /*
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;
 this.setLocation(x, y);
 */

 this.setTitle("Prueba Swing");
 this.setSize(dmVentana);
 // Centramos la ventana automáticamente
 this.setLocationRelativeTo(null);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 }

 public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
 }
}
```

Como se puede ver, hemos definido el tamaño de la ventana como una propiedad, de modo que sea fácil cambiarlo. Podemos probar a modificarlo y comprobar que la ventana siempre se muestra **centrada**.

## 5.2. Añadiendo elementos controles

### Controles

La cuestión ahora es: ¿cómo situamos los controles en la ventana?

Un objeto **JFrame**, o que derive de él, está compuesto de varios componentes superpuestos, uno de los cuales es lo que se denomina el **panel de contenido**. Este panel es un objeto de tipo **Container** que permite contener varios controles en él, dispuestos de una manera especial. Es en este contenedor donde deben situarse los componentes.

El uso de los componentes habitualmente sigue los siguientes pasos:

1. Declarar los controles
2. Crearlos (este paso puede hacerse junto con la declaración)
3. Modificar sus propiedades (opcional)
4. Añadirlos a la ventana para que se muestren. Esto puede hacerse de dos modos:
  5. Añadirlos directamente al panel de contenido del objeto **JFrame**
  6. Añadirlos primero a un contenedor (habitualmente un objeto **JPanel** o **JScrollPane**). Despues se añade el contenedor al panel de contenido (o a otro contenedor que, a su vez, se añadirá posteriormente al panel de contenidos).

Los controles suelen declararse como *variables de instancia* de la clase ventana para así poder ser referenciados desde cualquier otro método de la clase. Asimismo, se declaran como *elementos privados* para impedir su manipulación desde el exterior.

Para añadir un control al **panel de contenido** o a otro panel se emplea el método `add` de la clase `Container` (los paneles derivan de la clase `Container`, por lo que también disponen de ese método). Como veremos posteriormente, este método `add` puede llevar otros argumentos adicionales.

Antiguamente era necesario añadir el componente al panel de contenidos usando:

```
this.getContentPane().add(componente);
```

pero, desde hace tiempo, existe un método `add` en **JFrame** que se encarga de hacer esta operación de forma equivalente:

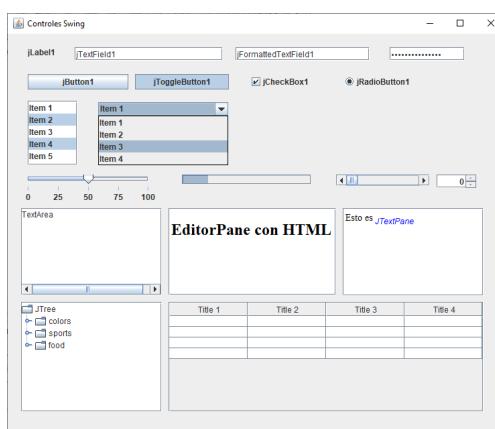
```
this.add(componente);
```

## Componentes

Los componentes son los elementos básicos en la construcción de interfaces. En **Swing**, estas clases heredan de `javax.swing.JComponent`, que a su vez hereda de `java.awt.Container`, que hereda de `java.awt.Component`, que hereda de `java.lang.Object`. Esto hace que gran parte de los métodos sean compartidos por todos los componentes. Entre estos métodos (que ya iremos viendo) destacan las funcionalidades para posicionar el componente, definir su tamaño, colores, fuentes y bordes, mostrarlo/ocultarlo, activarlo/desactivarlo, controlar su funcionamiento, etc.

Algunos de los **controles simples** de **Swing** son:

- **JLabel** : Componente que permite mostrar texto que no puede modificarse (puede ser HTML).
- **JButton** : Un botón.
- **JToggleButton** : Un botón con dos estados (presionado o no).
- **JCheckBox** : Una casilla de verificación.
- **JRadioButton** : Un botón de radio.
- **JComboBox** : Una lista desplegable de elementos.
- **JList** : Una lista de elementos.
- **JTextField** : Un cuadro de texto en el que el usuario puede añadir información (una única línea).
- **JTextArea** : Un cuadro de texto multínea.
- **JScrollBar** : Una barra de desplazamiento.
- **JSlider** : Un deslizador.
- **JProgressBar** : Una barra de progreso.
- **JFormattedTextField** : Un cuadro de texto formateado para introducir números, fechas, etc.
- **JPasswordField** : Un cuadro de texto para contraseñas (impide que se vea lo que se está tecleando).
- **JSpinner** : Un cuadro de texto numérico o de fecha que puede incrementar/decrementar su valor con botones.
- **JEditorPane** : Un cuadro de texto multínea con posibilidad de mostrar texto formateado en HTML o RTF.
- **JTextPane** : Extensión de JEditorPane, admite texto con estilos.
- **JTree** : Un árbol.
- **JTable** : Una tabla.



Vamos a probar a añadir un botón a la ventana. Como ya hemos visto, los botones son objetos de la clase **JButton**. Vamos a pasarle el texto del botón en el constructor (también podríamos hacerlo una vez creado el botón con el método `setText(String)`).

```
import java.awt.Dimension;
import java.awt.Toolkit;
import javax.swing.JButton;
import javax.swing.JFrame;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400,300);
 private JButton btnSalir = new JButton("Salir");

 public PruebaSwing() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba Swing");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 }
}
```

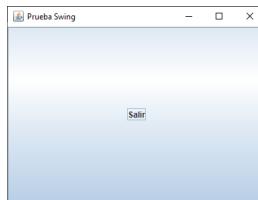
```

 this.add(btnSalir);
 }

 public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
 }
}

```

Si ejecutamos:



Como podemos ver, el botón **ocupa toda la ventana** y su tamaño cambia adaptándose a la misma. Aunque podemos pulsar el botón, no hace nada (tendremos que ser nosotros quienes indiquemos qué queremos hacer al pulsarlo, lo que veremos en temas posteriores).

¿Qué ocurre si añadimos otro componente? Vamos a añadir un cuadro de texto (`JTextField`):

```

import java.awt.Dimension;
import java.awt.Toolkit;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400,300);
 private JButton btnSalir = new JButton("Salir");
 private JTextField txtNombre = new JTextField();

 public PruebaSwing() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

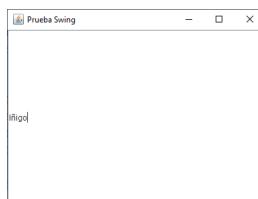
 this.setTitle("Prueba Swing");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

 this.add(btnSalir);
 this.txtNombre.setText("Iñigo"); // Modificamos el contenido
 this.add(txtNombre); // Lo añadimos a la ventana
 }

 public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
 }
}

```

Si ejecutamos:



Ahora el botón **ha desaparecido** y su lugar ha sido ocupado por el cuadro de texto, que pasa a ocupar todo el espacio de la ventana.

¿Por qué sucede esto?

Uno de los principales problemas a la hora de desarrollar interfaces de usuario son los diferentes tamaños de dispositivos en los que se pueden ejecutar. Desde el diseño de **AWT** (y posteriormente **Swing**), se pretendió diseñar un sistema de interfaces que pudiera adaptarse de la mejor forma posible a diferentes resoluciones de pantalla y que permitiera que, al redimensionar la ventana, los componentes pudieran redimensionarse también (un buen ejemplo es el propio **NetBeans**, que se adapta al tamaño de la ventana).

Para ello se crearon los denominados **Gestores de Distribución (Layouts)**.

Un **gestor de disposición** es un objeto que determina cómo se sitúan los controles en un contenedor. Para permitir que las aplicaciones **Java** puedan ejecutarse en diferentes sistemas operativos (*Windows, Mac, Linux...*), en diferentes dispositivos (*pantallas de ordenador, teléfonos móviles, ordenadores de mano...*) y a distintas resoluciones, se da la posibilidad de indicarle a **Java** cómo queremos situar los controles, de modo que sea el propio gestor de distribución quien los ubique en función del dispositivo, del sistema operativo y del tamaño de la ventana.

Esto, además, permite que los controles se adapten dinámicamente al espacio disponible cuando el usuario redimensiona la ventana (en nuestro ejemplo, el cuadro de texto ocupará siempre el espacio disponible en la ventana —en el contenedor—).

En el siguiente tema veremos algunos de los principales gestores de distribución.

**NOTA:** Siempre podemos optar por *no* utilizar un **gestor de disposición**, en cuyo caso seremos responsables de especificar la posición y el tamaño de cada componente de la ventana. Con este enfoque, dichos parámetros *no* se ajustarán automáticamente a los cambios de tamaño de la misma.

### 5.3. BorderLayout

#### BorderLayout

Como ya hemos visto en el ejemplo que estamos haciendo, al añadir más de un componente a la ventana, solo se muestra el último de ellos que ocupa todo el espacio de la misma.

Esto se debe a que, por defecto, la clase **JFrame** emplea como gestor de disposición un **BorderLayout**.

Este gestor de distribución está diseñado para las ventanas principales de la aplicación y dispone de **5 zonas**:

- **NORTH (PAGE\_START)**: Parte superior de la ventana.
- **SOUTH (PAGE\_END)**: Parte inferior de la ventana.
- **WEST (LINE\_START)**: Parte izquierda de la ventana.
- **EAST (LINE\_END)**: Parte derecha de la ventana.
- **CENTER**: Parte central de la ventana.

Está pensado para situar en la parte central los componentes principales de la aplicación y, en los bordes, elementos como barras de menú, herramientas o de estado. En cada una de las zonas solo puede situarse un control, aunque este control puede ser un contenedor que incluya otros componentes, creando así ventanas más complejas.

---

#### Añadiendo componentes a BorderLayout

Dado que tenemos 5 zonas, al añadir un componente al contenedor debemos indicar la ubicación del mismo. Para ello, utilizamos una versión sobrecargada del método `add`, en la que especificamos la zona en la que queremos situar el control mediante las constantes estáticas de la clase **BorderLayout**. Por ejemplo:

```
this.add(cmp, BorderLayout.NORTH);
```

Si no se especifica ninguna ubicación, se asume que el componente va a la zona **CENTER**.

---

#### Por qué desaparece el botón

Ahora ya sabemos qué ocurrió con el botón. Dado que solo cabe un control en cada zona y no especificamos en qué zona queríamos situarlo, el **BorderLayout** lo colocó en la zona **CENTER**. Al añadir el cuadro de texto en la misma zona, el botón fue eliminado para dar paso al cuadro de texto.

---

#### Cómo BorderLayout asigna espacio

En la disposición **BorderLayout**, primero se sitúan los controles de las esquinas. El tamaño que se les da (en el orden indicado) es:

- **Norte**: Espacio suficiente a lo alto para mostrarse. A lo ancho ocupan todo el espacio disponible.
- **Sur**: Igual que el norte.
- **Este**: Espacio suficiente a lo ancho para mostrarse. A lo alto ocupan todo el espacio disponible.
- **Oeste**: Igual que el este.
- **Centro**: Utiliza todo el espacio disponible tras situar el resto de los controles.

Si en alguna zona no se especifica un control, el espacio de esa zona lo reutilizan las demás zonas, teniendo en cuenta las reglas anteriores.

---

#### Tamaños de los controles

Los controles disponen de tres medidas (aparte de la medida real) que pueden ser especificadas:

- **Mínima**
- **Preferida**
- **Máxima**

Estos valores están predeterminados por Java. Algunos gestores de disposición emplean alguna o todas estas medidas para determinar el tamaño inicial de los controles y si pueden cambiar su tamaño y hasta cuánto. Sin embargo, no hay un funcionamiento estándar entre los diferentes gestores de disposición, por lo que habrá que estudiar cada caso individualmente.

Para modificar alguno de estos tamaños, podemos usar los métodos:

```
setMaximumSize(Dimension tamaño);
setMinimumSize(Dimension tamaño);
setPreferredSize(Dimension tamaño);
```

Además, disponemos del método `setSize` para asignar un tamaño específico al control, y los métodos correspondientes `get` para obtener los valores.

En el caso de **BorderLayout**, podemos especificar el tamaño preferido para los controles de las esquinas. Por ejemplo, si especificamos el tamaño preferido de un botón en la zona **NORTH**, solo se tendrá en cuenta la dimensión vertical, ya que a lo ancho siempre ocupará todo el espacio disponible.

**Nota:** Intentar cambiar el tamaño máximo, mínimo o el tamaño de un control en este gestor no tendrá ningún efecto.

---

#### Ejemplo práctico

Veamos un ejemplo en el que colocamos un cuadro de texto en la zona **NORTH**, un botón en la zona **CENTER**, otro botón en la zona **SOUTH** y un botón adicional en la zona **WEST**.

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Toolkit;
import javax.swing.JButton;
```

```

import javax.swing.JFrame;
import javax.swing.JTextField;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400, 300);
 private JButton btnSalir = new JButton("Salir");
 private JTextField txtNombre = new JTextField();
 private JButton btnLateral = new JButton("Lateral");
 private JButton btnCentral = new JButton("Centro");

 public PruebaSwing() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba Swing");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

 // Añadimos los componentes en distintas zonas
 this.add(btnSalir, BorderLayout.SOUTH);
 this.txtNombre.setText("Iñigo");
 this.add(txtNombre, BorderLayout.NORTH);
 this.add(btnLateral, BorderLayout.WEST);
 this.add(btnCentral, BorderLayout.CENTER);
 }

 public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
 }
}

```

---

#### Salida



#### Proceso de asignación de espacio

##### 1. Norte (NORTH):

El cuadro de texto se coloca en la parte superior. El gestor de contenido le pide su tamaño preferido, que incluye el ancho necesario para mostrar el texto y un alto de 20 píxeles. Luego, ajusta el ancho al de la ventana y el alto al preferido (20px). Finalmente, lo ubica en la posición (0, 0).

##### 2. Sur (SOUTH):

El botón "Salir" se coloca en la parte inferior. El gestor le pide sus dimensiones preferidas y ajusta el ancho al de la ventana y el alto al preferido. Luego, lo ubica en la posición (0, alto de la ventana - alto del botón).

##### 3. Oeste (WEST):

El botón "Lateral" se coloca en el lado izquierdo. El gestor le pide su tamaño preferido y ajusta el ancho al preferido y el alto al espacio disponible (alto de la ventana menos el alto del cuadro de texto y el botón inferior). Luego, lo ubica en la posición (0, alto del cuadro de texto).

##### 4. Centro (CENTER):

El botón "Centro" ocupa el espacio restante. El gestor ajusta su ancho al espacio disponible (ancho de la ventana menos el ancho del botón izquierdo) y su alto al espacio restante (alto de la ventana menos el alto del cuadro de texto y el botón inferior).

#### Modificando el tamaño preferido

Podemos modificar el tamaño preferido de los componentes. Por ejemplo, para hacer más ancho el botón en la zona WEST:

```
btnLateral.setPreferredSize(new Dimension(100, 0));
```

La salida:

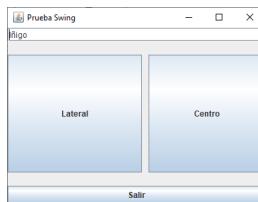


#### Espaciado entre componentes

La clase **BorderLayout** permite definir un espacio horizontal y vertical entre componentes mediante su constructor:

```
this.setLayout(new BorderLayout(10, 20));
```

- **10:** Espaciado horizontal entre componentes.
- **20:** Espaciado vertical entre componentes.



#### Añadiendo un borde interior

Si queremos añadir un margen entre los componentes y el borde de la ventana, podemos usar un borde interior con la clase `EmptyBorder`:

```
import javax.swing.border.EmptyBorder;

this.getRootPane().setBorder(new EmptyBorder(5, 5, 5, 5));
```

Esto añade un margen de 5 píxeles en cada lado.



#### 5.4. FlowLayout GridLayout NullLayout

##### FlowLayout

El gestor de disposición **FlowLayout** sitúa los componentes uno detrás de otro en filas. Los controles se colocan centrados desde la parte superior de la ventana, ocupando tantas filas como sean necesarias en función del número y del tamaño de los mismos. Al redimensionar la ventana, los controles se reubican para incluir el máximo posible en cada fila. Dado que no hay que especificar ningún parámetro adicional, los controles simplemente se añaden al contenedor.

```
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Toolkit;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400,300);
 private JButton btnSalir = new JButton("Salir");
 private JTextField txtNombre = new JTextField();
 private JButton btnLateral = new JButton("Lateral");
 private JButton btnCentral = new JButton("Centro");
 private JPanel panel = new JPanel(); // Los JPanel llevan por defecto FlowLayout

 public PruebaSwing() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

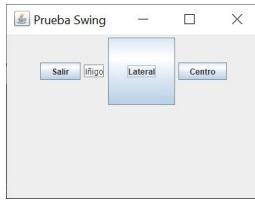
 this.setTitle("Prueba Swing");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

 // AÑADIMOS COMPONENTES AL PANEL
 this.panel.add(btnSalir); // Añadimos los controles al Panel
 this.txtNombre.setText("Iñigo");
 this.panel.add(txtNombre);
 this.btnLateral.setPreferredSize(new Dimension(100,100)); // Cambiamos el alto!
 this.panel.add(btnLateral);
 this.panel.add(btnCentral);

 this.add(panel); // Y añadimos el panel a la ventana
 }

 public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
 }
}
```

La salida:



En este gestor no se redimensiona el tamaño de los controles en función del tamaño de la ventana, sino que siempre mantienen su tamaño. El tamaño que se emplea para mostrar el componente es el tamaño preferido del mismo, como se puede comprobar en el caso del botón lateral.

Si redimensionamos el formulario, los controles se van ubicando en filas y centrados:



Al igual que en el caso anterior, podemos asignar explícitamente un **FlowLayout** a cualquier panel. En este caso, podemos especificar:

- El espaciado horizontal (**hgap**) entre componentes
- El espaciado vertical (**vgap**) entre componentes
- La alineación de los mismos en caso de que sobre espacio:
  - **FlowLayout.LEFT**
  - **FlowLayout.RIGHT**
  - **FlowLayout.CENTER** (valor por defecto)

Si cambiamos la declaración del panel:

```
private JPanel panel = new JPanel(new FlowLayout(FlowLayout.RIGHT, 20, 20));
```

La salida:



### GridLayout

Con este tipo de disposición, los controles se sitúan en una matriz de filas y columnas. El número de éstas debe especificarse al crear el objeto gestor de distribución. El espacio se divide equitativamente entre todos los controles, que deben añadirse en orden por filas y columnas. El tamaño de los componentes no puede ser modificado por ninguno de los métodos vistos.

En este caso, es necesario asignar el gestor de disposición al panel. Hay que especificar las filas, columnas y, opcionalmente, el **hgap** y el **vgap**.

A la hora de distribuir los componentes, se va a dividir verticalmente el espacio disponible entre el número de filas y después se va a calcular cuántas columnas son necesarias dividiendo el número de componentes entre el número de filas, y ése es el número de columnas a usar. Una vez creado el grid, se van añadiendo los componentes por filas hasta que no queden más. El resto del espacio se queda en blanco.

Es decir, el número de columnas que pasamos al constructor no se usa para nada.

Vamos a crear un panel con 8 componentes en un grid de 4x2:

```
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Toolkit;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400,300);
 private JButton[] botones = new JButton[8];
 private JPanel panel = new JPanel(new GridLayout(4,2,5,5)); // 4 filas, 2 columnas, gap 5x5

 public PruebaSwing() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba Swing");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 }
}
```

```

// CREAMOS Y AÑADIMOS LOS BOTONES
for (int i = 0; i < botones.length; i++) {
 this.botones[i] = new JButton("Btn:" + i);
 this.panel.add(this.botones[i]);
}

this.panel.setBorder(new EmptyBorder(5,5,5,5));
this.add(panel); // Y añadimos el panel a la ventana
}

public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
}
}

```

La salida:



¿Qué pasa si ponemos 3x2? Si seguimos lo que hemos dicho, creará 3 filas de 8 / 3 --> 3 columnas cada una:



De hecho, daría lo mismo si pusiéramos 3x0 o 3x10, dado que este layout sólo tiene en cuenta las filas.

## 5.5. Propiedades Básicas de Componentes y Controles JLabel JTextField JFormattedTextField JButton

### Componentes Swing: Propiedades y Métodos Comunes

Todos los controles Swing derivan de la clase **JComponent** (que a su vez deriva de la clase **Component**), por lo que comparten una serie de métodos comunes. Los más relevantes son:

#### Posicionamiento y tamaño

- `Rectangle getBounds() / void setBounds(Rectangle rect)`: Retorna/establece la posición y el tamaño del componente (el efecto depende del layout empleado)
- `Point getLocation() / void setLocation(Point pos)`: Retorna/establece la ubicación del componente
- `int getX() / int getY()`: Retorna las coordenadas x e y del componente
- `Dimension getMinimumSize() / void setMinimumSize(Dimension tam)`: Retorna/establece el tamaño mínimo para el componente
- `Dimension getMaximumSize() / void setMaximumSize(Dimension tam)`: Retorna/establece el tamaño máximo para el componente
- `Dimension getPreferredSize() / void setPreferredSize(Dimension tam)`: Retorna/establece el tamaño preferido para el componente
- `Dimension getSize() / void setSize(Dimension tamaño) / void setSize(int ancho, int alto)`: Retorna/establece el tamaño del control

#### Aspecto

- `Border getBorder() / void setBorder(Border borde)`: Retorna/especifica el borde del componente
- `String getToolTipText() / void setToolTipText(String pista)`: Retorna/establece el texto de la pista
- `void setBackground(Color fondo) / void setForeground(Color frente)`: Establece el color del fondo/primer plano
- `Color getBackground() / Color getForeground()`: Retorna los colores de fondo/primer plano
- `voidsetFont(Font) / Font getFont()`: Establece/retorna la fuente del componente
- `void setOpaque() / boolean isOpaque()`: Indica/comprueba si el objeto es opaco
- `void setCursor(Cursor c) / Cursor getCursor()`: Establece/retorna el cursor que se muestra cuando el ratón está sobre el componente

**Nota 1:** Los bordes son objetos que implementan el interfaz `Border`. Pueden crearse directamente o a través de la clase `BorderFactory`.

**Nota 2:** Los cursos se crean mediante la clase `Cursor` con constantes predefinidas.

Icon	Constant	Icon	Constant
	DEFAULT_CURSOR		NE_RESIZE_CURSOR
	CROSSHAIR_CURSOR		E_RESIZE_CURSOR
	HAND_CURSOR		SE_RESIZE_CURSOR
	MOVE_CURSOR		S_RESIZE_CURSOR
	TEXT_CURSOR		SW_RESIZE_CURSOR
	WAIT_CURSOR		W_RESIZE_CURSOR
	N_RESIZE_CURSOR		INVISIBLE_CURSOR

#### Accesibilidad

- void disable() / void enable(): Desactiva/activa el componente
- void requestFocus(): Solicita el enfoque para el componente
- boolean isFocusable(): Retorna true si se puede ceder el enfoque al control
- void setEnabled(boolean): Establece si el componente está activo
- boolean isEnabled(): Retorna true si el componente está activo
- void setVisible(boolean): Establece si el objeto debe ser visible
- boolean isVisible(): Retorna true si el componente es visible

#### Etiquetas (JLabel)

La clase `JLabel` permite mostrar texto no editable e imágenes:

```

• JLabel() / JLabel(Icon) / JLabel(Icon,int) / JLabel(String) / JLabel(String,Icon,int) / JLabel(String,int): Crea una nueva etiqueta
• Icon getDisabledIcon() / void setDisabledIcon(Icon): Establece/retorna el icono para cuando la etiqueta está desactivada
• int getHorizontalAlignment() / void setHorizontalAlignment(int): Retorna/establece la orientación horizontal
• Icon getIcon() / void setIcon(Icon): Retorna/establece la imagen a mostrar
• int getIconTextGap() / void setIconTextGap(int): Retorna/establece el espacio entre imagen y texto
• String getText() / void setText(String): Retorna/establece el texto de la etiqueta
• int getVerticalAlignment() / void setVerticalAlignment(int): Retorna/establece la alineación vertical
• int getVerticalTextPosition() / void setVerticalTextPosition(int): Retorna/establece la posición vertical del texto respecto a la imagen

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.Toolkit;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400,300);

 public PruebaSwing() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba Swing");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 this.setLayout(new GridLayout(3,1,5,5));

 // Etiqueta centrada con fuente personalizada
 JLabel uno = new JLabel("Etiqueta Centrada", JLabel.CENTER);
 uno.setFont(new Font("Calibri", Font.PLAIN, 32));
 uno.setForeground(Color.blue);
 uno.setBackground(Color.white);
 this.add(uno);

 // Etiqueta con imagen
 ImageIcon im = new ImageIcon("c:\\duke.gif");
 JLabel dos = new JLabel("Con imagen", im, JLabel.RIGHT);
 dos.setForeground(Color.blue);
 dos.setBackground(Color.white);
 dos.setOpaque(true);
 this.add(dos);

 // Etiqueta con HTML
 JLabel tres = new JLabel("<html><h2>Prueba HTML</h2><hr>Texto en <i>cursiva</i></html>");
 tres.setHorizontalAlignment(JLabel.RIGHT);
 tres.setVerticalAlignment(JLabel.TOP);
 this.add(tres);
 }

 public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
 }
}

```



#### Botones (JButton)

Los botones en Swing derivan de la clase abstracta `AbstractButton`. Principales métodos:

- void doClick(): Hace clic en el botón por código
- Action getAction() / void setAction(Action): Retorna/establece la acción del botón
- String getActionCommand() / void setActionCommand(String): Retorna/establece el comando de acción

- `Icon getDisabledIcon() / void setDisabledIcon(Icon)`: Retorna/establece el ícono cuando está desactivado
- `int getHorizontalAlignment() / void setHorizontalAlignment(int)`: Retorna/establece la orientación horizontal
- `Icon getIcon() / void setIcon(Icon)`: Retorna/establece la imagen del botón
- `int getMnemonic() / void setMnemonic(int)`: Retorna/establece la tecla aceleradora (ALT + tecla)
- `String getText() / void setText(String)`: Retorna/establece el texto del botón
- `boolean isSelected() / void setSelected(boolean)`: Retorna/establece si el botón está seleccionado

```

import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Toolkit;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400,300);

 public PruebaSwing() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba Swing");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 this.setLayout(new GridLayout(3,1,5,5));

 // Botón normal con tecla aceleradora
 JButton uno = new JButton("Botón Normal");
 uno.setMnemonic('N');
 uno.setRolloverIcon(new ImageIcon("c:\\\\duke.gif"));
 this.add(uno);

 // Botón con imagen
 ImageIcon im = new ImageIcon("c:\\duke.gif");
 JButton dos = new JButton("Con imagen", im);
 dos.setForeground(Color.blue);
 dos.setBackground(Color.white);
 dos.setOpaque(true);
 this.add(dos);

 // Botón con HTML
 JButton tres = new JButton("<html><h2>Prueba HTML</h2><hr>Texto en <i>cursiva</i></html>");
 tres.setHorizontalAlignment(JButton.RIGHT);
 tres.setVerticalAlignment(JButton.TOP);
 this.add(tres);
 }

 public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
 }
}

```



### Cuadros de Texto

La clase `JTextComponent` es la superclase de todas las clases que trabajan con texto.

#### JTextField

Un componente que permite almacenar una única línea de texto:

- `JTextField() / JTextField(int) / JTextField(String) / JTextField(String,int)`: Crea un nuevo cuadro de texto
- `int getColumns() / void setColumns(int)`: Retorna/establece el número de columnas
- `int getHorizontalAlignment() / void setHorizontalAlignment(int)`: Retorna/establece la alineación

#### JPasswordField

Subclase de `JTextField` diseñada para contraseñas:

- `JPasswordField() / JPasswordField(int) / JPasswordField(String) / JPasswordField(String,int)`: Crea un nuevo cuadro de contraseñas
- `char getEchoChar() / void setEchoChar(char)`: Retorna/establece el carácter de eco
- `char[] getPassword()`: Devuelve el contenido del campo

#### JFormattedTextField

Extiende `JTextField` para añadir soporte para formatear valores:

- **JFormattedText() / JFormattedText(Format) / JFormattedText(Object):** Crea un nuevo componente con formato
- **Object getValue() / void setValue(Object):** Retorna/establece el valor del componente
- **boolean isEditValid():** True si el valor es válido

```

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Toolkit;
import java.text.DateFormat;
import java.text.NumberFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.swing.JButton;
import javax.swing.JFormattedTextField;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.text.MaskFormatter;

public class Formatted extends JFrame {
 private Dimension dmVentana = new Dimension(400,300);
 private JLabel lblEntero = new JLabel("Entero");
 private JFormattedTextField ftxEntero = new JFormattedTextField(NumberFormat.getIntegerInstance());
 private JLabel lblDecimal = new JLabel("Con decimales");
 private JFormattedTextField ftxDecimal = new JFormattedTextField(NumberFormat.getIntegerInstance());
 private JLabel lblMoneda = new JLabel("Monetario");
 private JFormattedTextField ftxMoneda = new JFormattedTextField(NumberFormat.getCurrencyInstance());
 private JLabel lblPorcentaje = new JLabel("Porcentaje");
 private JFormattedTextField ftxPorcentaje;
 private JLabel lblFCorta = new JLabel("Fecha Corta");
 private JFormattedTextField ftxFCorta = new JFormattedTextField(new DateFormat.getDateInstance(DateFormat.SHORT));
 private JLabel lblFMedia = new JLabel("Fecha Media");
 private JFormattedTextField ftxFMedia = new JFormattedTextField(new DateFormat.getDateInstance());
 private JLabel lblPersonal = new JLabel("aaaa-mm-dd");
 private JFormattedTextField ftxFFPersonal = new JFormattedTextField(new SimpleDateFormat("yyyy-MM-dd"));
 private JLabel lblIp = new JLabel("Con máscara (dirección IP)");
 private JFormattedTextField ftxIp;
 private JPanel pnl = new JPanel(new GridLayout(8,2,0,5));
 private JButton btn = new JButton("Ver Datos");

 public Formatted() {
 NumberFormat nf = NumberFormat.getPercentInstance();
 nf.setMinimumFractionDigits(2);
 nf.setMaximumIntegerDigits(2);
 ftxPorcentaje = new JFormattedTextField(nf);

 try {
 ftxIp = new JFormattedTextField(new MaskFormatter("###.###.###.###"));
 } catch (ParseException ex) { }

 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;
 this.setLocation(x, y);
 this.setSize(dmVentana);
 this.setTitle("JFormattedTextField");
 this.getRootPane().setBorder(new EmptyBorder(10,10,10,10));

 // Añadimos componentes al panel
 pnl.add(lblEntero); pnl.add(ftxEntero);
 pnl.add(lblDecimal); pnl.add(ftxDecimal);
 pnl.add(lblMoneda); pnl.add(ftxMoneda);
 pnl.add(lblPorcentaje); pnl.add(ftxPorcentaje);
 pnl.add(lblFCorta); pnl.add(ftxFCorra);
 pnl.add(lblFMedia); pnl.add(ftxFMedia);
 pnl.add(lblFFPersonal); pnl.add(ftxFFPersonal);
 pnl.add(lblIp); pnl.add(ftxIp);

 // Establecemos valores iniciales
 ftxEntero.setValue(12345);
 ftxDecimal.setValue(12345.6789);
 ftxPorcentaje.setValue(0.23545);
 ftxMoneda.setValue(12345.6789);
 Date fecha = new Date();
 ftxFCorta.setValue(fecha);
 ftxFMedia.setValue(fecha);
 ftxFFPersonal.setValue(fecha);
 ftxIp.setValue("192.168.001.255");

 this.add(pnl);
 this.add(btn, BorderLayout.SOUTH);

 // Añadimos acción al botón
 btn.addActionListener((e) -> {
 System.out.println("Entero : " + ((Number)ftxEntero.getValue()).intValue());
 System.out.println("Decimal : " + ((Number)ftxDecimal.getValue()).doubleValue());
 System.out.println("Porcentaje : " + ((Number)ftxPorcentaje.getValue()).doubleValue());
 System.out.println("Moneda : " + ((Number)ftxMoneda.getValue()).doubleValue());
 System.out.println("Fecha Corta : " + ((Date)ftxFCorra.getValue()));
 System.out.println("Fecha Media : " + ((Date)ftxFMedia.getValue()));
 System.out.println("Fecha Personalizada : " + ((Date)ftxFFPersonal.getValue()));
 });
 }
}

```

```

 System.out.println("Dirección IP : " + (ftxIp.getValue()));
 });

 this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 this.setVisible(true);
}

public static void main(String[] args) {
 new Formatted();
}
}

```



#### JTextArea

Este componente permite introducir varias líneas de texto plano:

- `JTextArea() / JTextArea(int, int) / JTextArea(String) / JTextArea(String, int, int)`: Crea un nuevo objeto
- `void append(String)`: Añade la cadena al final del texto
- `int getLineCount()`: Retorna el número de líneas
- `boolean getLineWrap() / void setLineWrap(boolean)`: Retorna/establece si se debe hacer salto de línea
- `boolean getWrapStyleWord() / void setWrapStyleWord(boolean)`: Retorna/establece si el salto se hace en límites de palabra
- `void insert(String, int)`: Inserta el texto en la posición indicada
- `void replaceRange(String, int, int)`: Reemplaza texto en las posiciones especificadas

```

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.Toolkit;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class PruebaSwing extends JFrame {
 private Dimension dmVentana = new Dimension(400, 300);
 private JTextField nombre;
 private JPasswordField clave;
 private JTextArea comentario;
 private JButton ok;
 private JButton cancel;

 public PruebaSwing() {
 this.setTitle("Prueba JText");
 this.setSize(250, 300);
 Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
 int posX = (int) ((d.getWidth() - this.getWidth()) / 2);
 int posY = (int) ((d.getHeight() - this.getHeight()) / 2);
 this.setLocation(posX, posY);
 this.getRootPane().setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

 // Panel para nombre y contraseña
 JPanel pan1 = new JPanel();
 pan1.setLayout(new GridLayout(2, 2, 5, 5));
 nombre = new JTextField(30);
 pan1.add(new JLabel("Nombre : "));
 pan1.add(nombre);
 clave = new JPasswordField(6);
 pan1.add(new JLabel("Contraseña : "));
 pan1.add(clave);

 // Panel para comentarios
 JPanel pan2 = new JPanel(new BorderLayout());
 pan2.setBorder(BorderFactory.createEmptyBorder(10, 0, 0, 0));
 comentario = new JTextArea();
 comentario.setLineWrap(true);
 comentario.setWrapStyleWord(true);
 JScrollPane sp = new JScrollPane();
 sp.setViewportView(comentario);
 JLabel l = new JLabel("Comentarios : ");
 pan2.add(l, BorderLayout.NORTH);
 pan2.add(sp, BorderLayout.CENTER);

 // Panel para botones
 JPanel pan3 = new JPanel();
 pan3.setLayout(new FlowLayout());
 }
}

```

```

pan3.setBorder(BorderFactory.createEmptyBorder(10, 0, 0, 0));
ok = new JButton("Aceptar");
cancel = new JButton("Cancelar");
pan3.add(ok);
pan3.add(cancel);
getContentPane().setDefaultButton(cancel);

// Añadimos los paneles a la ventana
this.add(pan1, BorderLayout.NORTH);
this.add(pan2, BorderLayout.CENTER);
this.add(pan3, BorderLayout.SOUTH);
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public static void main(String[] args) {
 new PruebaSwing().setVisible(true);
}
}

```



## 5.6. Creación de Interfaces Complejas

### Creación de Interfaces Complejas en Swing

Para crear interfaces complejas en Swing, la estrategia recomendada es combinar contenedores (paneles) con distintos tipos de layout e ir anidándolos. Además, podemos aprovechar las propiedades de los componentes para personalizar el diseño.

Veamos un ejemplo de cómo crear una ventana con un diseño más elaborado:



Este formulario utiliza un **BorderLayout** como base, con tres paneles principales:

- En el **Norte**: un panel con **FlowLayout** que contiene una etiqueta de título
- En el **Centro**: un panel con **GridLayout** que contiene los campos de datos
- En el **Sur**: un panel con **FlowLayout** que contiene los botones

El diseño incluye personalización de colores de fondo y texto, fuentes personalizadas, y modificación del aspecto de los botones. También muestra cómo modificar dinámicamente múltiples controles de un panel y cómo cambiar el aspecto de todos los controles de un tipo determinado.

```

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.Toolkit;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.UIManager;

public class FrmEmpleados extends JFrame {
 private Dimension dmVentana = new Dimension(450, 300);

 // Componentes de la interfaz
 private JLabel lblTitulo = new JLabel("Formulario de Empleados");
 private JLabel lblNombre = new JLabel("Nombre");
 private JTextField txtNombre = new JTextField();
 private JLabel lblApellidos = new JLabel("Apellidos");
 private JTextField txtApellidos = new JTextField();
 private JLabel lblEmail = new JLabel("Email");
 private JTextField txtEmail = new JTextField();
 private JLabel lblFechaAlta = new JLabel("Fecha Alta");
 private JTextField txtFechaAlta = new JTextField();
 private JButton btnGuardar = new JButton("Guardar");
 private JButton btnSalir = new JButton("Salir");

 // Paneles para organizar los componentes

```

```

private JPanel pnlTitulo = new JPanel();
private JPanel pnlDatos = new JPanel(new GridLayout(4, 2, 0, 10));
private JPanel pnlBotones = new JPanel();

// Colores y fuentes personalizadas
private Color clrBase = new Color(152, 74, 49);
private Color clrBotones = new Color(116, 62, 41);
private Font fntBotones = new Font("Calibri", Font.PLAIN, 18);
private Font fntBase = new Font("Calibri", Font.PLAIN, 16);

public FrmEmpleados() {
 // Configuración básica de la ventana
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;
 this.setTitle("Formulario de Empleados");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

 // Configuración del panel de título
 this.lblTitulo.setFont(new Font("Calibri", Font.PLAIN, 26));
 this.pnlTitulo.setBackground(clrBase);
 this.lblTitulo.setForeground(Color.white);
 this.pnlTitulo.add(this.lblTitulo);

 // Configuración del panel de datos
 this.pnlDatos.add(lblNombre);
 this.pnlDatos.add(txtNombre);
 this.pnlDatos.add(lblApellidos);
 this.pnlDatos.add(txtApellidos);
 this.pnlDatos.add(lblEmail);
 this.pnlDatos.add(txtEmail);
 this.pnlDatos.add(lblFechaAlta);
 this.pnlDatos.add(txtFechaAlta);

 // Modificación dinámica de los componentes del panel de datos
 for (Component c : this.pnlDatos.getComponents()) {
 JComponent cmp = (JComponent) c;
 if (cmp instanceof JLabel) {
 cmp.setForeground(Color.white);
 }
 cmp.setFont(fntBase);
 }

 this.pnlDatos.setBackground(clrBase);
 this.pnlDatos.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

 // Configuración del botón Guardar
 this.btnGuardar.setMnemonic('G');
 this.btnGuardar.setBackground(clrBotones);
 this.btnGuardar.setBorderPainted(false);
 this.btnGuardar.setFont(fntBotones);
 this.btnGuardar.setCursor(new Cursor(Cursor.HAND_CURSOR));
 this.btnGuardar.setForeground(Color.white);
 this.btnGuardar.setFocusPainted(false);
 this.pnlBotones.add(this.btnGuardar);

 // Configuración del botón Salir
 this.btnSalir.setMnemonic('S');
 this.btnSalir.setBackground(clrBotones);
 this.btnSalir.setBorderPainted(false);
 this.btnSalir.setFont(fntBotones);
 this.btnSalir.setCursor(new Cursor(Cursor.HAND_CURSOR));
 this.btnSalir.setForeground(Color.white);
 this.btnSalir.setFocusPainted(false);
 this.pnlBotones.add(this.btnSalir);

 this.pnlBotones.setBackground(clrBase);
 this.getRootPane().setDefaultButton(this.btnGuardar);

 // Añadimos los paneles a la ventana
 this.add(pnlTitulo, BorderLayout.NORTH);
 this.add(pnlDatos, BorderLayout.CENTER);
 this.add(pnlBotones, BorderLayout.SOUTH);

 // Modificación global del aspecto de los botones
 UIManager.put("Button.select", clrBase);
}
}

public static void main(String[] args) throws InterruptedException {
 new FrmEmpleados().setVisible(true);
}
}

```

#### **Aspectos destacados del diseño**

1. Estructura de paneles anidados:
2. Panel principal con BorderLayout
3. Tres paneles secundarios con diferentes layouts

4. Personalización visual:
5. Colores personalizados para fondos y textos
6. Fuentes personalizadas para diferentes elementos
7. Modificación del aspecto de los botones (sin bordes, con cursor de mano)

#### 8. Técnicas de programación:

9. Modificación dinámica de múltiples componentes usando `getComponents()`

10. Uso de `UIManager` para cambiar el aspecto global de los botones

11. Configuración de teclas mnemónicas para acceso rápido

#### 12. Detalles de usabilidad:

13. Botón por defecto configurado con `setDefaultButton`

14. Cursor personalizado para los botones

15. Espaciado adecuado con bordes

**Nota:** Para encontrar paletas de colores adecuadas, se pueden utilizar recursos como [color-hex.com](http://color-hex.com).

Esta técnica de combinar diferentes layouts y personalizar la apariencia de los componentes permite crear interfaces de usuario sofisticadas y atractivas en Swing.

### 5.7. Actividad 26 - Creación de Interfaces Complejos

#### Instrucciones

[Classroom](#)

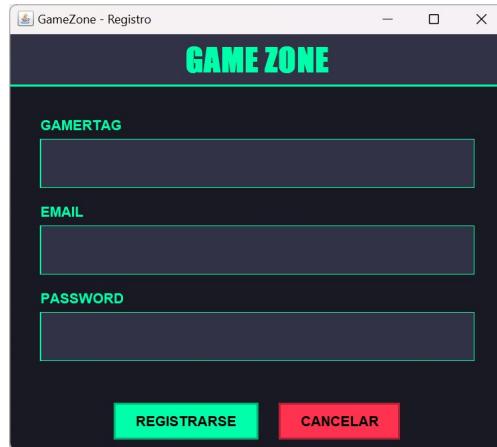
Se pide diseñar los siguientes formularios:

1

The screenshot shows a Java Swing application window titled "Mi Perfil". The window has a blue header bar with the title "Mi Perfil". The main content area is white and contains five input fields, each with a label and a text input field. The labels are: "Nombre completo", "Correo electrónico", "Teléfono", "Dirección", and "Sitio web". The "Nombre completo" field has a cursor in it. At the bottom right of the window are two buttons: "Cancelar" (gray) and "Guardar Cambios" (blue).

NOTA: Para conseguir un borde sólo en la parte inferior de un control, mirar el método `BorderFactory.createMatteBorder()`

2



Pista: En ambos formularios los controles son paneles

### 5.8. Look And Feel

#### Personalización del Look and Feel en Swing

El aspecto visual de las aplicaciones Swing puede ser modificado globalmente mediante el sistema de **Look and Feel (L&F)**. Java incluye varios L&F predeterminados, y también es posible instalar otros adicionales, tanto gratuitos como de pago.

#### Look and Feel Predeterminados

Para ver los L&F instalados en el sistema, podemos usar el siguiente código:

```
Stream.of(UIManager.getInstalledLookAndFeels())
 .forEach(l -> {
 System.out.println(l.getName() + " : " + l.getClassName());
 });
}
```

La salida típica muestra:

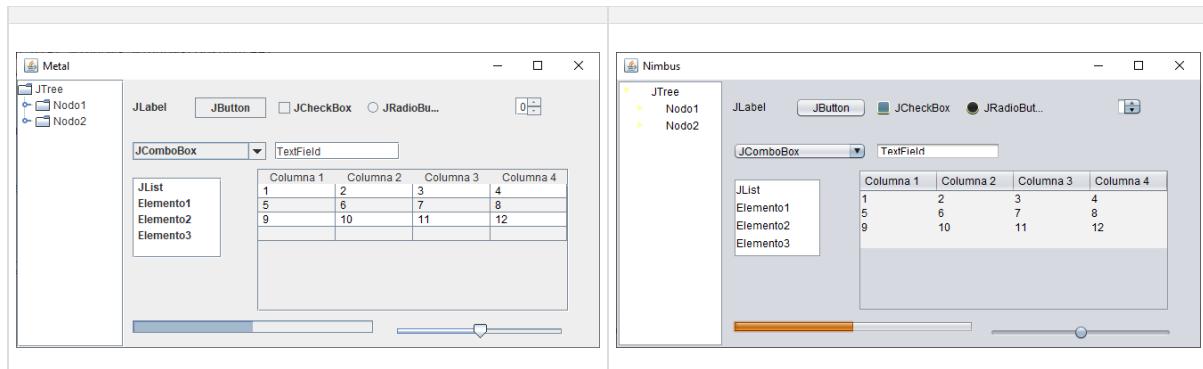
```
Metal - javax.swing.plaf.metal.MetalLookAndFeel
Nimbus - javax.swing.plaf.nimbus.NimbusLookAndFeel
CDE/Motif - com.sun.java.swing.plaf.motif.MotifLookAndFeel
Windows - com.sun.java.swing.plaf.windows.WindowsLookAndFeel
Windows Classic - com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel
```

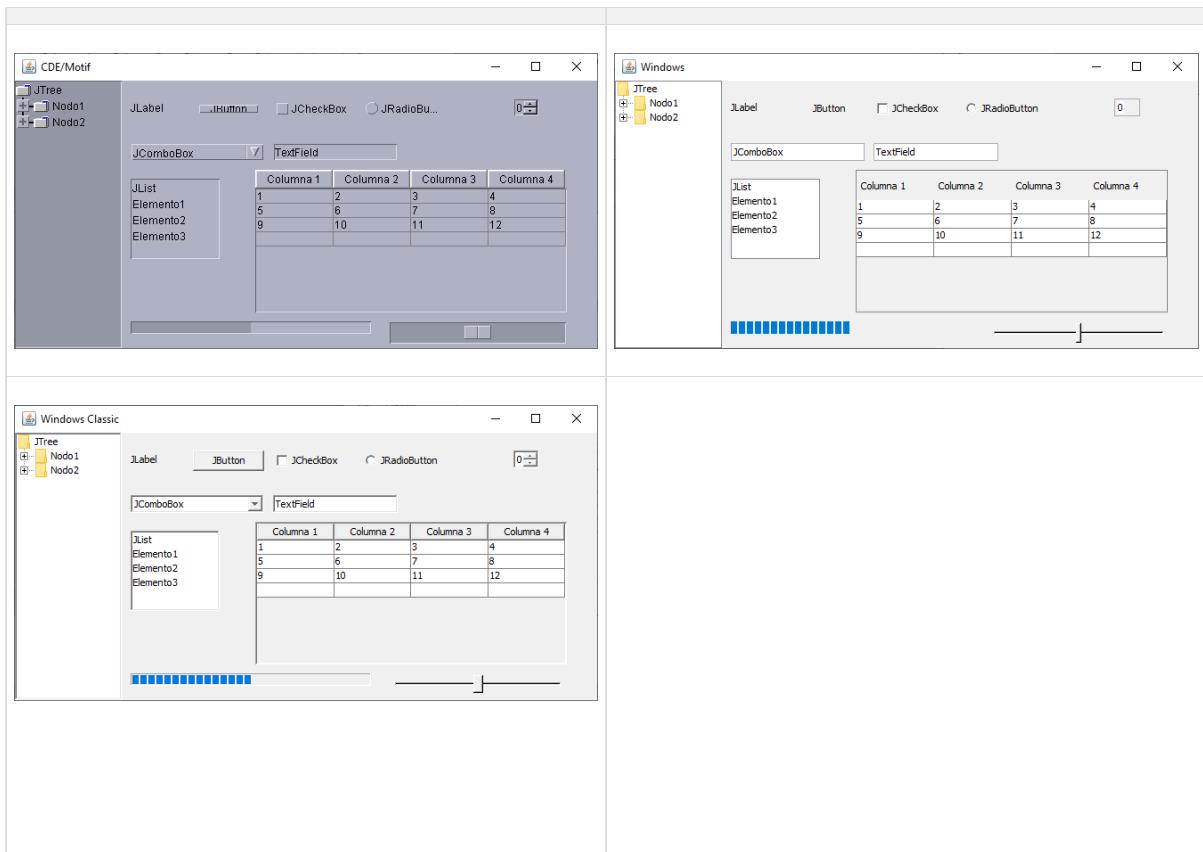
**Nota:** Si no indicamos ninguno, Java emplea Windows por defecto en sistemas Windows.

Para aplicar un Look and Feel, usamos:

```
try {
 UIManager.setLookAndFeel(new LookAndFeel());
} catch (Exception ex) {
 ex.printStackTrace();
}
```

Ejemplos de los L&F predeterminados:





#### Look and Feel Adicionales

Si los L&F predeterminados no satisfacen nuestras necesidades, podemos instalar otros. Veamos dos opciones populares:

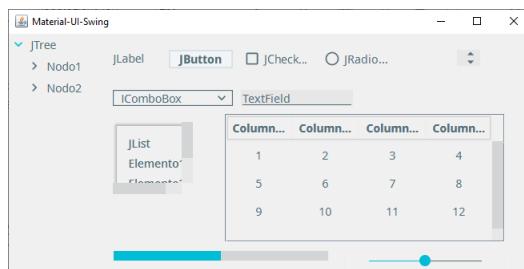
##### 1. Material-UI

Este L&F está disponible en [GitHub](#). Para instalarlo, añadimos la siguiente dependencia al proyecto Maven:

```
<dependency>
 <groupId>io.github.vincenzopalazzo</groupId>
 <artifactId>material-ui-swing</artifactId>
 <version>1.1.4</version>
</dependency>
```

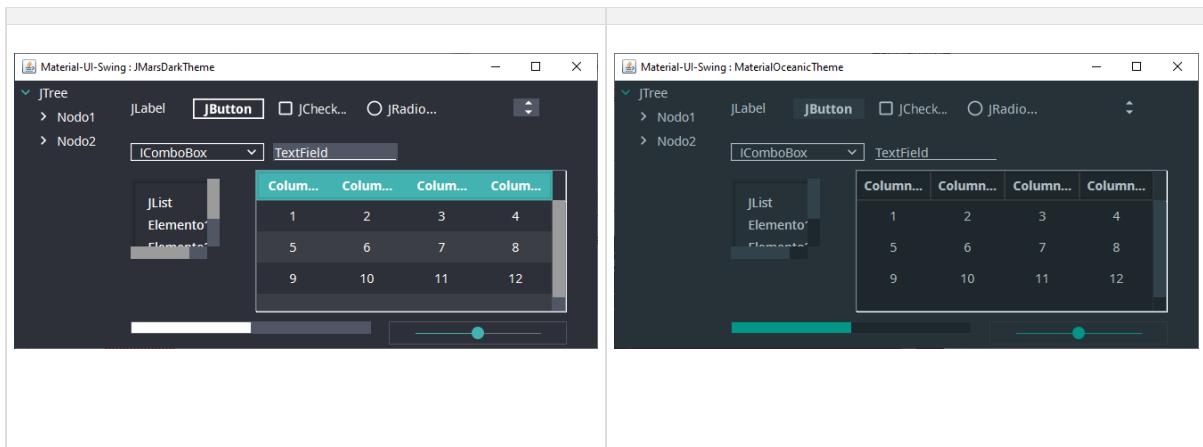
Para aplicarlo:

```
try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel());
} catch (Exception ex) {
 ex.printStackTrace();
}
```



Material-UI incluye tres temas: MaterialLiteTheme (predeterminado), JMarsDarkTheme y MaterialOceanicTheme. Para activar un tema específico:

```
try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new JMarsDarkTheme()))
} catch (Exception ex) {
 ex.printStackTrace();
}
```



## 2. FlatLaf

Este Look and Feel es utilizado por NetBeans y otros IDEs. El proyecto se encuentra en [formdev.com](http://formdev.com). Para incluirlo en un proyecto Maven:

```
<dependency>
 <groupId>com.formdev</groupId>
 <artifactId>flatlaf</artifactId>
 <version>3.0</version>
</dependency>
```

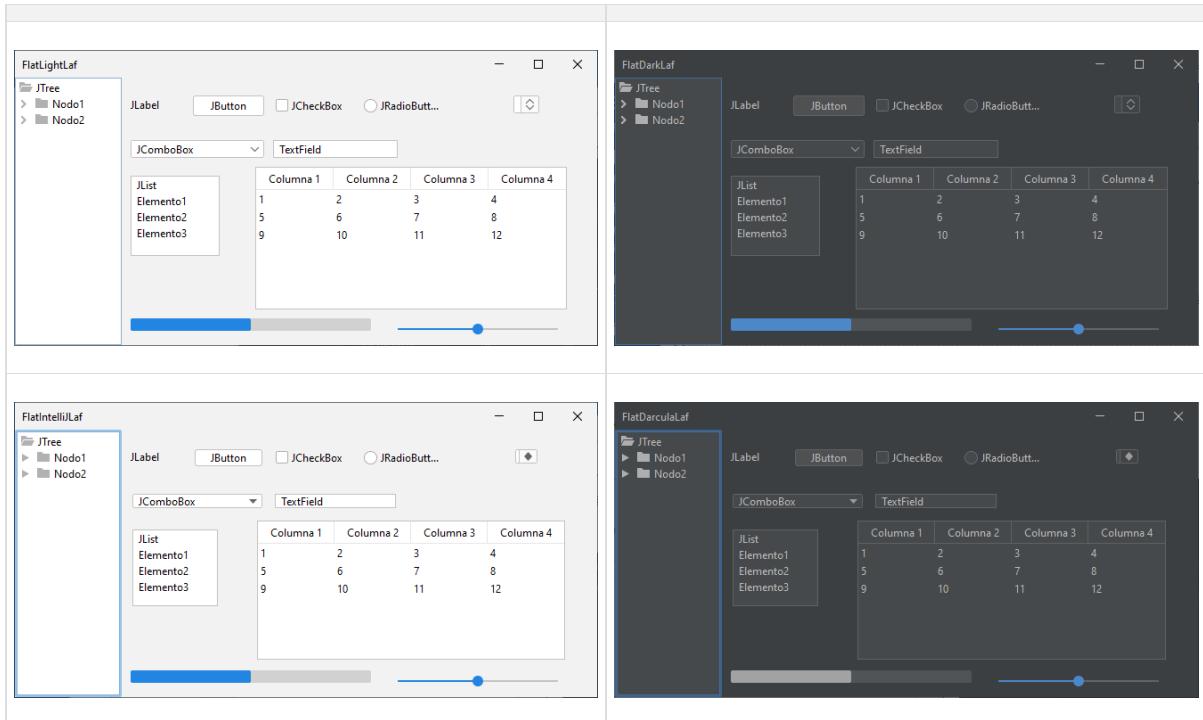
**Nota:** Si se copia la dependencia desde Maven Repository, hay que quitar el elemento `<scope>runtime</scope>`.

FlatLaf ofrece un método simplificado para su instalación:

```
Tema.setup();
```

Los temas incluidos son:

- FlatLightLaf
- FlatDarkLaf
- FlatIntelliJLaf
- FlatDarculaLaf



Además, se puede instalar un paquete adicional de temas basados en IntelliJ IDEA con el archivo `flatlaf-intellij-themes-2.0.1.jar`, que incluye más de 50 temas.

El listado completo de temas está disponible en [GitHub](https://github.com/johnyzzz/flatlaf-intellij-themes).

Para aplicar un tema específico:

```
FlatCobalt2IJTTheme.setup();
```

Para explorar todos los temas y ver cómo se aplican a la aplicación, se puede utilizar la [Demo](#) proporcionada.

## 5.9. Gestión de Eventos

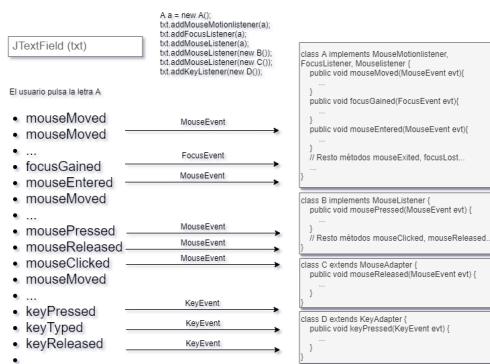
### Introducción al Modelo de Eventos en Java Swing

La programación en Java Swing se basa en un paradigma de **programación dirigida por eventos** (Event-Driven Programming). Este modelo permite que las aplicaciones respondan a las acciones del usuario, como pulsaciones de teclas, clics de ratón o cambios en los componentes.

### Fundamentos del Modelo de Eventos

Cuando un usuario interactúa con la interfaz (escribe un carácter, pulsa una tecla, etc.), se produce un **evento**. Cualquier objeto puede ser notificado de que dicho evento ha ocurrido si:

1. Implementa el interfaz adecuado para ese tipo de evento
2. Se registra como receptor de eventos (event listener) para la fuente del evento (event source)



### Tipos de Eventos y sus Interfaces

Interfaz	Eventos	Clase Adaptadora
ActionListener	void actionPerformed(ActionEvent)	
KeyListener	void keyPressed(KeyEvent) void keyReleased(KeyEvent) void keyTyped(KeyEvent)	KeyAdapter
MouseListener	void mouseClicked(MouseEvent) void mouseEntered(MouseEvent) void mouseExited(MouseEvent) void mousePressed(MouseEvent) void mouseReleased(MouseEvent)	MouseAdapter
MouseMotionListener	void mouseDragged(MouseEvent) void mouseMoved(MouseEvent)	MouseMotionAdapter
WindowListener	void windowActivated(WindowEvent) void windowClosed(WindowEvent) void windowClosing(WindowEvent) void windowDeactivated(WindowEvent) void windowDeiconified(WindowEvent) void windowIconified(WindowEvent) void windowOpened(WindowEvent)	WindowAdapter

**Nota:** Las **clases adaptadoras** implementan todos los métodos del interfaz con cuerpos vacíos, permitiendo sobreescribir solo los métodos que nos interesan.

### Eventos Semánticos vs. Eventos de Bajo Nivel

- **Eventos de bajo nivel:** Representan acciones físicas directas (clic de ratón, pulsación de tecla)
- **Eventos semánticos:** Encapsulan varios eventos de bajo nivel o representan acciones de alto nivel

Por ejemplo, un botón puede activarse de varias formas (clic, tecla espaciadora, tecla mnemónica), pero todas generan un único evento semántico `ActionEvent`.

### Procedimientos para Gestionar Eventos

Existen cuatro enfoques principales para implementar la gestión de eventos:

#### 1. Usando la Clase como Receptora del Evento

```

import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class PruebaEventos extends JFrame implements MouseListener {

```

```

private Dimension dmVentana = new Dimension(100, 100);
private JButton btn = new JButton("Púlsame!");

public PruebaEventos() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba de Eventos");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 this.add(btn);
 this.btn.addMouseListener(this);
}

@Override
public void mouseClicked(MouseEvent e) {
 System.out.println(e);
}

@Override
public void mousePressed(MouseEvent e) {
 JOptionPane.showMessageDialog(this, "Pulsado!");
 System.out.println(e);
}

@Override
public void mouseReleased(MouseEvent e) {
 System.out.println(e);
}

@Override
public void mouseEntered(MouseEvent e) {
 System.out.println(e);
}

@Override
public void mouseExited(MouseEvent e) {
 System.out.println(e);
}

public static void main(String[] args) {
 new PruebaEventos().setVisible(true);
}
}

```

## 2. Usando una Clase Interna

```

import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class PruebaEventos extends JFrame {
 private Dimension dmVentana = new Dimension(100, 100);
 private JButton btn = new JButton("Púlsame!");

 public PruebaEventos() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba de Eventos");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 this.add(btn);
 this.btn.addMouseListener(new ReceptorBoton());
 }

 public static void main(String[] args) {
 new PruebaEventos().setVisible(true);
 }
}

// Clase interna que extiende MouseAdapter
class ReceptorBoton extends MouseAdapter {
 @Override
 public void mousePressed(MouseEvent e) {
 JOptionPane.showMessageDialog(PruebaEventos.this, "Pulsado!");
 System.out.println(e);
 }
}

```

## 3. Usando una Clase Interna Anónima

```

import java.awt.Dimension;
import java.awt.Toolkit;

```

```

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class PruebaEventos extends JFrame {
 private Dimension dmVentana = new Dimension(100, 100);
 private JButton btn = new JButton("Púlsame!");

 public PruebaEventos() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba de Eventos");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 this.add(btn);

 // Clase anónima que extiende MouseAdapter
 this.btn.addMouseListener(new MouseAdapter() {
 @Override
 public void mousePressed(MouseEvent e) {
 JOptionPane.showMessageDialog(PruebaEventos.this, "Pulsado!");
 }
 });
 }

 public static void main(String[] args) {
 new PruebaEventos().setVisible(true);
 }
}

```

Es común extraer la lógica a un método separado:

```

import java.awt.Dimension;
import java.awt.HeadlessException;
import java.awt.Toolkit;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class PruebaEventos extends JFrame {
 private Dimension dmVentana = new Dimension(100, 100);
 private JButton btn = new JButton("Púlsame!");

 public PruebaEventos() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba de Eventos");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 this.add(btn);

 this.btn.addMouseListener(new MouseAdapter() {
 @Override
 public void mousePressed(MouseEvent e) {
 btnPulsado(e);
 }
 });
 }

 private void btnPulsado(MouseEvent e) throws HeadlessException {
 JOptionPane.showMessageDialog(PruebaEventos.this, "Pulsado!");
 }

 public static void main(String[] args) {
 new PruebaEventos().setVisible(true);
 }
}

```

#### 4. Usando Expresiones Lambda (Java 8+)

Las expresiones lambda son ideales para interfaces funcionales (interfaces con un solo método abstracto):

```

import java.awt.Dimension;
import java.awt.HeadlessException;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class PruebaEventos extends JFrame {
 private Dimension dmVentana = new Dimension(100, 100);
 private JButton btn = new JButton("Púlsame!");

```

```

public PruebaEventos() {
 Dimension dmPantalla = Toolkit.getDefaultToolkit().getScreenSize();
 int x = (dmPantalla.width - dmVentana.width) / 2;
 int y = (dmPantalla.height - dmVentana.height) / 2;

 this.setTitle("Prueba de Eventos");
 this.setSize(dmVentana);
 this.setLocation(x, y);
 this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
 this.add(btn);

 // Expresión lambda para ActionListener
 this.btn.addActionListener((ActionEvent e) -> {
 btnPulsado(e);
 });
}

private void btnPulsado(ActionEvent e) throws HeadlessException {
 JOptionPane.showMessageDialog(PruebaEventos.this, "Pulsado!");
}

public static void main(String[] args) {
 new PruebaEventos().setVisible(true);
}
}

```

### Consideraciones Prácticas

1. Elección del enfoque:
2. Para aplicaciones simples, las expresiones lambda son concisas y legibles
3. Para lógica compleja, las clases internas permiten mejor organización
4. Si se necesita reutilizar el código de manejo de eventos, las clases separadas son preferibles
5. Eventos semánticos vs. de bajo nivel:
6. Preferir eventos semánticos (como `ActionEvent`) cuando sea posible
7. Usar eventos de bajo nivel solo cuando se necesite control detallado
8. Acceso a componentes:
9. Desde clases internas, usar `ClaseExterna.this` para acceder a la instancia externa
10. Considerar pasar referencias a componentes relevantes en constructores de clases manejadoras

### 5.10. Actividad 27 - Gestión de Eventos

#### Instrucciones

Partiendo de la primera actividad (**Formulario de Productos**) realizada en la tarea de **Creación de Interfaces Complejos** se pide lo siguiente:

##### Actividad 1

- Crear una clase interna que implemente el interfaz `FocusListener` y que:
- En el método `focusGained` cambie el color de fondo del origen del evento (método `getSource()` del evento que se recibe - `FocusEvent` -)
- En el método `focusLost` haga lo mismo pero poniendo el fondo del componente en **blanco**
- Definir un objeto de dicha clase y añadirlo como receptor de eventos a todos los cuadros de texto del formulario (premio para el que lo haga con una repetitiva)
- Comprobar el correcto funcionamiento de lo realizado. El resultado debe ser que, cuando accedemos a un cuadro de texto, éste cambia de color y vuelve al blanco cuando lo abandonamos

##### Actividad 2

- Añadir al botón `Guardar` un objeto de una clase anónima que implemente `ActionListener` y que llame a un método que:
- Compruebe que los cuadros de texto tengan información
- De no ser así, dé un mensaje de error
- Si los cuadros tienen información se deberá mostrar dicha información en un `JOptionPane`.

##### Actividad 3

- Añadir al botón `Nuevo` un receptor de evento de tipo `ActionListener` empleando una expresión lambda que llame a un método que deje en blanco todos los controles del formulario

##### Actividad 4

- Añadir al botón `Salir` un receptor de eventos de tipo `ActionListener` (con el mecanismo que queráis) que llame a un método (NO PASÉIS EL EVENTO) que:
- Pregunte al usuario (`showConfirmDialog`) si quiere salir de la aplicación
- Si dice que sí finalice el programa con la instrucción `System.exit(0)`;

## Actividad 5

- Añadir a la ventana (`this`) un objeto de una clase anónima que extienda `WindowAdapter` y que sobrescriba el método `windowClosing` para que llame al método definido en el paso anterior
- Modificar el código de la ventana para que la opción por defecto al intentar cerrarla (`setDefaultCloseOperation`) sea no hacer nada (`DO NOTHING ON CLOSE`)
- Comprobar que, al intentar cerrar la ventana nos pide confirmación

### 5.11. Ejemplo Práctico Creación de un Formulario de Contactos

## Desarrollo de una Aplicación de Gestión de Empleados con Java Swing

El objetivo es desarrollar una aplicación Java Swing completa para la gestión de Empleados, utilizando un modelo, un DAO en memoria basado en colecciones, una capa de servicio y una interfaz atractiva con Swing.

Los datos se encuentran en [empleados.dat](#) y es un archivo escrito con `DataOutputStream`. Las fechas siguen el formato dd/MM/yyyy.

### Estructura del Proyecto

El proyecto utiliza Maven con las siguientes dependencias:

```
<dependencies>
 <dependency>
 <groupId>org.projectlombok</groupId>
 <artifactId>lombok</artifactId>
 <version>1.18.30</version>
 <scope>provided</scope>
 </dependency>
 <dependency>
 <groupId>io.github.vincenzopalazzo</groupId>
 <artifactId>material-ui-swing</artifactId>
 <version>1.1.2</version>
 </dependency>
</dependencies>
```

### Capa de Modelo

#### Clase Empleado

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
import java.util.Date;

@Data
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@NoArgsConstructor
@AllArgsConstructor
public class Empleado {
 private Integer id;
 private String apellidos;
 private String nombre;
 private String cargo;
 private Integer jefe;
 private Date fechaNacimiento;
 private Date fechaAlta;
 private String direccion;
 private String ciudad;
 private String estado;
 private String pais;
 private String codigoPostal;
 private String telefono;
 private String fax;
 private String email;
 private String foto;
}
```

### Capa de Acceso a Datos

#### Interfaz EmpleadoDAO

```
import java.util.List;

public interface EmpleadoDAO {
 List<Empleado> getEmpleados();
 Empleado getEmpleado(int id);
}
```

#### Implementación EmpleadoImpl

```
import java.io.*;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.List;
import javax.swing.JOptionPane;

public class EmpleadoImpl implements EmpleadoDAO {
```

```

private List<Empleado> empleados = new ArrayList<>();
private DateFormat df = new SimpleDateFormat("dd/MM/yyyy");

public EmpleadoImpl() {
 if (empleados.isEmpty()) {
 DataInputStream dis = null;
 try {
 dis = new DataInputStream(
 new BufferedInputStream(
 new FileInputStream("empleados.dat")));
 while (true) {
 Empleado e = new Empleado(
 dis.readInt(), // Id
 dis.readUTF(), // Apellidos
 dis.readUTF(), // Nombre
 dis.readUTF(), // Cargo
 dis.readInt(), // Jefe
 df.parse(dis.readUTF()), // Fecha Nacimiento
 df.parse(dis.readUTF()), // Fecha Alta
 dis.readUTF(), // Dirección
 dis.readUTF(), // Ciudad
 dis.readUTF(), // Estado
 dis.readUTF(), // País
 dis.readUTF(), // Código Postal
 dis.readUTF(), // Teléfono
 dis.readUTF(), // Fax
 dis.readUTF(), // Email
 dis.readUTF() // Foto
);
 empleados.add(e);
 }
 } catch (EOFException ex) {
 try { dis.close(); } catch (IOException ex1) {}
 } catch (FileNotFoundException ex) {
 JOptionPane.showMessageDialog(null,
 "No se ha encontrado el fichero");
 } catch (IOException ex) {
 JOptionPane.showMessageDialog(null,
 "Se ha producido un error inesperado al leer el fichero");
 } catch (ParseException ex) {
 JOptionPane.showMessageDialog(null,
 "Se ha producido un error inesperado al leer el fichero");
 }
 }
}

@Override
public List<Empleado> getEmpleados() {
 return empleados;
}

@Override
public Empleado getEmpleado(int id) {
 return empleados.stream()
 .filter(e -> e.getId() == id)
 .findAny()
 .orElse(null);
}
}

```

## Capa de Servicio

### Clase EmpleadoService

```

import java.util.List;

public class EmpleadoService {
 private EmpleadoDAO dao = new EmpleadoImpl();

 public EmpleadoService() {
 }

 public List<Empleado> getEmpleados() {
 return dao.getEmpleados().stream()
 .sorted((e1, e2) -> (e1.getApellidos() + "," + e1.getNombre())
 .compareToIgnoreCase(e2.getApellidos() + "," + e2.getNombre()))
 .toList();
 }
}

```

```

 public Empleado getEmpleado(int id) {
 return dao.getEmpleado(id);
 }
}

```

## Capa de Presentación

### Formulario de Contactos Básico

```

import java.awt.*;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.*;
import javax.swing.border.EmptyBorder;
import mdlaf.themes.JMarsDarkTheme;
import mdlaf.MaterialLookAndFeel;

public class ContactosForm extends JFrame {
 private EmpleadoService service = new EmpleadoService();

 private JPanel pnlDatos;
 private JTextField txtApellidos;
 private JTextField txtNombre;
 private JTextField txtCargo;
 private JTextField txtTelefono;
 private JTextField txtEmail;
 private JLabel lblActual;
 private JLabel lblFoto;

 private JPanel pnlBotones;
 private JButton btnAnterior;
 private JButton btnSiguiente;

 private int pos = 0;

 public ContactosForm() {
 inicializarUI();
 }

 private void inicializarUI() {
 setTitle("Contactos");
 setSize(800, 400);
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 // Añadiendo imagen
 lblFoto = new JLabel();
 lblFoto.setBorder(new EmptyBorder(5, 5, 5, 5));
 add(lblFoto, BorderLayout.WEST);

 pnlDatos = new JPanel(new GridLayout(5, 2, 0, 10));
 pnlDatos.setBorder(new EmptyBorder(10, 10, 10, 10));

 JLabel apellidosLabel = new JLabel("Apellidos:");
 pnlDatos.add(apellidosLabel);
 txtApellidos = new JTextField(20);
 pnlDatos.add(txtApellidos);

 JLabel nombreLabel = new JLabel("Nombre:");
 pnlDatos.add(nombreLabel);
 txtNombre = new JTextField(20);
 pnlDatos.add(txtNombre);

 JLabel cargoLabel = new JLabel("Cargo:");
 pnlDatos.add(cargoLabel);
 txtCargo = new JTextField(20);
 pnlDatos.add(txtCargo);

 JLabel telefonoLabel = new JLabel("Teléfono:");
 pnlDatos.add(telefonoLabel);
 txtTelefono = new JTextField(20);
 pnlDatos.add(txtTelefono);

 JLabel emailLabel = new JLabel("Email:");
 pnlDatos.add(emailLabel);
 txtEmail = new JTextField(20);
 pnlDatos.add(txtEmail);

 this.add(pnlDatos, BorderLayout.CENTER);

 pnlBotones = new JPanel(new FlowLayout(FlowLayout.RIGHT));
 pnlBotones.setBorder(new EmptyBorder(5, 5, 5, 5));

 btnAnterior = new JButton("Anterior");
 btnAnterior.addActionListener(e -> mostrarAnterior());
 pnlBotones.add(btnAnterior);

 btnSiguiente = new JButton("Siguiente");
 btnSiguiente.addActionListener(e -> mostrarSiguiente());
 pnlBotones.add(btnSiguiente);

 lblActual = new JLabel();
 }
}

```

```

lblActual.setBorder(new EmptyBorder(5, 5, 5, 5));
add(lblActual, BorderLayout.NORTH);

// Sólo lectura
for (Component c : pnlDatos.getComponents()) {
 if (c instanceof JTextField) {
 ((JTextField) c).setEditable(false);
 }
}

add(pnlBotones, BorderLayout.SOUTH);
setLocationRelativeTo(null);
mostrarContactoActual();
}

private void mostrarAnterior() {
 pos--;
 mostrarContactoActual();
}

private void mostrarSiguiente() {
 pos++;
 mostrarContactoActual();
}

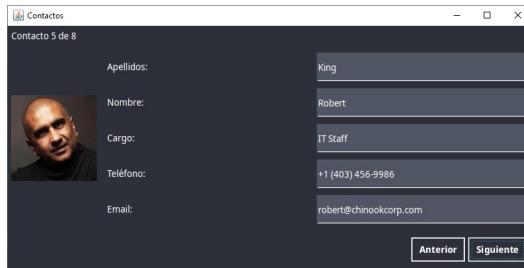
private void mostrarContactoActual() {
 Empleado empleado = service.getEmpleados().get(pos);
 txtApellidos.setText(empleado.getApellidos());
 txtNombre.setText(empleado.getNombre());
 txtCargo.setText(empleado.getCargo());
 txtTelefono.setText(empleado.getTelefono());
 txtEmail.setText(empleado.getEmail());

 try {
 lblFoto.setIcon(new ImageIcon(new URL(empleado.getFoto())));
 } catch (MalformedURLException ex) {
 lblFoto.setIcon(null);
 }
}

lblActual.setText("Contacto " + (pos + 1) + " de " + service.getEmpleados().size());
btnAnterior.setEnabled(pos > 0);
btnSiguiente.setEnabled(pos < service.getEmpleados().size() - 1);
}

public static void main(String[] args) {
 try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new JMarsDarkTheme()));
 } catch (UnsupportedLookAndFeelException ex) {
 Logger.getLogger(ContactosForm.class.getName()).log(Level.SEVERE, null, ex);
 }
 new ContactosForm().setVisible(true);
}
}

```



## Mejora con Iconos usando JIconFont

Para mejorar la interfaz, podemos añadir iconos utilizando la biblioteca JIconFont:

### Dependencias adicionales

```

<dependency>
 <groupId>com.github.jiconfont</groupId>
 <artifactId>jiconfont-swing</artifactId>
 <version>1.0.0</version>
</dependency>
<dependency>
 <groupId>com.github.jiconfont</groupId>
 <artifactId>jiconfont-google_material_design_icons</artifactId>
 <version>2.2.0.2</version>
</dependency>

```

### Código actualizado para los botones

```

import jiconfont.swing.IconFontSwing;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;

// En el método inicializarUI()
IconFontSwing.register(GoogleMaterialDesignIcons.getIconFont());

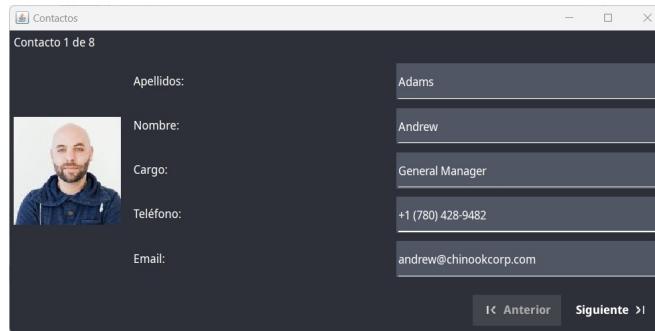
```

```

pnlBotones = new JPanel(new FlowLayout(FlowLayout.RIGHT));
pnlBotones.setBorder(new EmptyBorder(5, 5, 5, 5));
btnAnterior = new JButton("Anterior");
btnAnterior.setIcon(IconFontSwing.buildIcon(GoogleMaterialDesignIcons.FIRST_PAGE, 20, Color.WHITE));
btnAnterior.addActionListener(e -> mostrarAnterior());
pnlBotones.add(btnAnterior);

btnSiguiente = new JButton("Siguiente");
btnSiguiente.setHorizontalTextPosition(JButton.LEFT);
btnSiguiente.setIcon(IconFontSwing.buildIcon(GoogleMaterialDesignIcons.LAST_PAGE, 20, Color.WHITE));
btnSiguiente.addActionListener(e -> mostrarSiguiente());
pnlBotones.add(btnSiguiente);

```



### Personalización Final: Barra de Título Personalizada

Para dar un aspecto más moderno, podemos eliminar la barra de título estándar y crear una personalizada:

```

// Variables adicionales
private JPanel pnlBarra = new JPanel(new BorderLayout());
private JLabel lblCerrar = new JLabel("x");
private int px, py;

// Añadir al final del método inicializarUI()
setUndecorated(true); // Eliminar decoración de ventana

lblActual = new JLabel();
pnlBarra.add(lblActual);

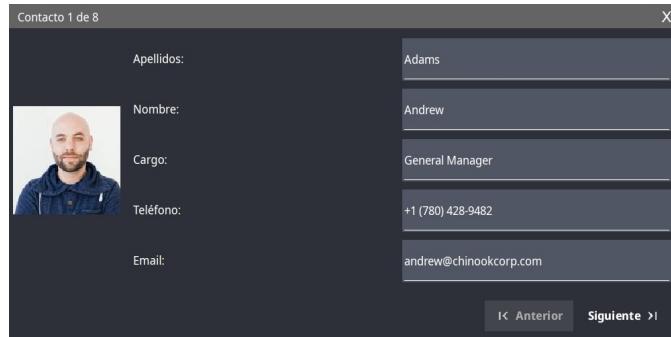
lblCerrar.setFont(new Font("Arial", Font.PLAIN, 18));
lblCerrar.setForeground(Color.WHITE);
lblCerrar.setCursor(new Cursor(Cursor.HAND_CURSOR));
lblCerrar.addMouseListener(new MouseAdapter() {
 @Override
 public void mousePressed(MouseEvent e) {
 System.exit(0);
 }
});
pnlBarra.add(lblCerrar, BorderLayout.EAST);
pnlBarra.setBorder(new EmptyBorder(5, 10, 2, 10));

lblActual.addMouseListener(new MouseAdapter() {
 public void mousePressed(MouseEvent me) {
 px = me.getX();
 py = me.getY();
 }
});

lblActual.addMouseMotionListener(new MouseAdapter() {
 public void mouseDragged(MouseEvent me) {
 ContactosForm.this.setLocation(
 getLocation().x + me.getX() - px,
 getLocation().y + me.getY() - py
);
 }
});

pnlBarra.setOpaque(true);
pnlBarra.setBackground(new Color(100, 100, 100));
this.add(pnlBarra, BorderLayout.NORTH);

```



## Conclusión

Esta aplicación implementa una arquitectura en capas clara:

1. **Capa de Modelo:** Clase `Empleado` con Lombok para reducir código boilerplate
2. **Capa de Acceso a Datos:** Interfaz `EmpleadoDAO` e implementación `EmpleadoImpl` que lee datos de un archivo binario
3. **Capa de Servicio:** `EmpleadoService` que ordena los empleados y proporciona acceso a los datos
4. **Capa de Presentación:** `ContactosForm` con una interfaz atractiva usando Material Design

### 5.12. Creación de una Aplicación Distribuible

#### Creación de una Aplicación Ejecutable

Para convertir nuestra aplicación de gestión de empleados en un archivo ejecutable (.jar), necesitamos modificar el archivo `pom.xml` para incluir la configuración del plugin `maven-assembly-plugin`. Este plugin nos permitirá crear un JAR que incluya todas las dependencias necesarias.

#### Configuración del `pom.xml`

A continuación se muestra un ejemplo completo de configuración del archivo `pom.xml`:

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

<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>org.zabalburu</groupId>
 <artifactId>PruebaImagen</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>jar</packaging>

 <properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>18</maven.compiler.source>
 <maven.compiler.target>18</maven.compiler.target>
 <exec.mainClass>org.zabalburu.app.Form</exec.mainClass>
 </properties>

 <dependencies>
 <!-- Lombok -->
 <dependency>
 <groupId>org.projectlombok</groupId>
 <artifactId>lombok</artifactId>
 <version>1.18.30</version>
 <scope>provided</scope>
 </dependency>

 <!-- Material UI Swing -->
 <dependency>
 <groupId>io.github.vincenzopalazzo</groupId>
 <artifactId>material-ui-swing</artifactId>
 <version>1.1.2</version>
 </dependency>

 <!-- JIconFont -->
 <dependency>
 <groupId>com.github.jiconfont</groupId>
 <artifactId>jiconfont-swing</artifactId>
 <version>1.0.0</version>
 </dependency>

 <dependency>
 <groupId>com.github.jiconfont</groupId>
 <artifactId>jiconfont-google_material_design_icons</artifactId>
 <version>2.2.0.2</version>
 </dependency>
 </dependencies>

 <build>
 <!-- Configuración para incluir recursos -->
 <resources>
 <resource>
 <directory>src/main/resources</directory>
 <includes>
```

```

<include>**/*.png</include>
<include>**/*.gif</include>
</includes>
</resource>
</resources>

<plugins>
 <!-- Plugin para crear JAR con dependencias -->
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-assembly-plugin</artifactId>
 <executions>
 <execution>
 <phase>package</phase>
 <goals>
 <goal>single</goal>
 </goals>
 <configuration>
 <archive>
 <manifest>
 <mainClass>
 org.zabalburu.app.ContactosForm
 </mainClass>
 </manifest>
 </archive>
 <descriptorRefs>
 <descriptorRef>jar-with-dependencies</descriptorRef>
 </descriptorRefs>
 </configuration>
 </execution>
 </executions>
 </plugin>
</plugins>
</build>
</project>

```

**Nota:** Asegúrate de reemplazar `org.zabalburu.app.ContactosForm` con la ruta completa a tu clase principal que contiene el método `main`.

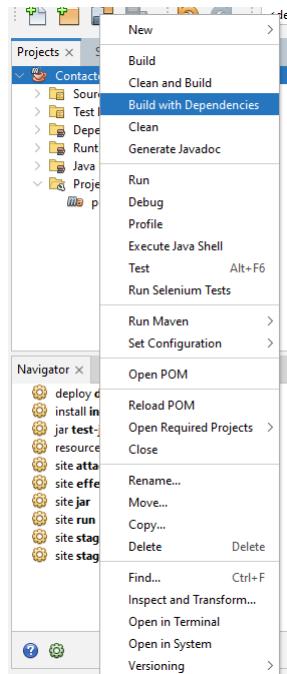
#### Elementos clave de la configuración:

1. **Recursos:** La sección `<resources>` configura qué archivos de recursos (imágenes, etc.) se incluirán en el JAR final.
2. **Plugin maven-assembly:** Este plugin es el responsable de crear un JAR ejecutable que incluye todas las dependencias.
3. La etiqueta `<mainClass>` especifica la clase que contiene el método `main` que se ejecutará al iniciar la aplicación.
4. El descriptor `jar-with-dependencies` indica que queremos incluir todas las dependencias en un único archivo JAR.

#### Proceso de construcción

Para construir el proyecto y generar el archivo JAR ejecutable:

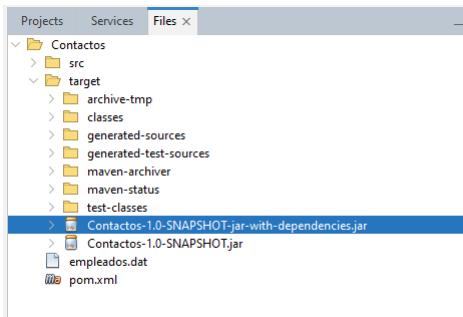
1. Haz clic derecho en el proyecto en tu IDE
2. Selecciona la opción "Build with Dependencies" del menú contextual



Esto generará dos archivos JAR en la carpeta `target` del proyecto:

- Un JAR básico sin dependencias

- Un JAR con todas las dependencias incluidas (el que usaremos)



## Distribución de la Aplicación

Para distribuir la aplicación, solo necesitas:

1. El archivo JAR con dependencias (por ejemplo, `PruebaImagen-1.0-SNAPSHOT-jar-with-dependencies.jar`)
2. El archivo de datos `empleados.dat` en la misma carpeta que el JAR

El usuario final podrá ejecutar la aplicación simplemente haciendo doble clic en el archivo JAR, siempre que tenga instalado el Java Runtime Environment (JRE) en su sistema.

**Nota:** Puedes cambiar el nombre del archivo JAR sin afectar su funcionamiento, por ejemplo a `GestionEmpleados.jar` para hacerlo más descriptivo.

## Requisitos del sistema para los usuarios finales

- Java Runtime Environment (JRE) versión 18 o superior (según la configuración del `pom.xml`)
- Sistema operativo compatible con Java (Windows, macOS, Linux)

Esta configuración permite crear una aplicación Java Swing completamente portable que puede distribuirse fácilmente a los usuarios finales.

## 5.13. Actividad 28-I - Creación de un Formulario De Productos

### Instrucciones

Se desea diseñar un formulario de productos ([Funko\\_Pops](#)) que nos vaya a permitir visualizar, añadir, modificar y eliminar productos de un fichero [funkos.dat](#). Además se proporcionan algunas imágenes de los productos en cuestión que deberán copiarse a una carpeta imágenes en la carpeta raíz del proyecto.

### Modelo

Se deberá definir una clase `Funko` que representa cada producto y que contendrá la siguiente información (en el mismo orden que está en el fichero que se ha creado con un DataStream):

- **id:** Un entero único que identifica cada producto
- **nombre:** El nombre del producto
- **dimensiones:** Una cadena donde se especifican las dimensiones del producto
- **color:** Una cadena con el color
- **precio:** Un double con el precio
- **foto:** El nombre del fichero donde se encuentra la foto del producto

La clase deberá tener los constructores y setters/getters adecuados así como un método equals basado en el id.

### DAO

Se deberá definir un interfaz `FunkoDAO` con los siguientes métodos:

- `List<Funko> getFunkos()`
- `Funko getFunko(int id)`
- `Funko nuevoFunko(Funko nuevo)`
- `void modificarFunko(Funko modificar)`
- `void eliminarFunko(int id)`

Además se definirá una clase (`FunkoFich`) que implementará dicho interfaz, definirá una propiedad `funkos` (`List<Funko>`) para almacenar los productos y que definirá (en total) los siguientes métodos:

- **constructor:** Este método leerá el fichero y, con su información, rellenará la lista de funkos
- **List getFunkos():** Retornará la lista de funkos
- **Funko getFunko(int id):** Retornará el producto de la lista con el id indicado si existe. Si no, retornará null
- **Funko nuevoFunko(Funko nuevo):** En este método se realizarán las siguientes tareas:
  - Se definirá una variable local id
  - Si la lista está vacía, se le asignará un 1
  - Si no, se asignará el id del último producto de la lista más 1
  - Se asignará dicho id al nuevo funkoo
  - Se añadirá el nuevo a la lista
  - Se llamará a guardar() (ver más adelante)
  - Se retornará
- **void modificarFunko(Funko modificar):** Se buscará la posición del producto con el mismo id que modificar. Si se encuentra, se cambiará el producto que esté en

esa posición por el producto modificar (método set del ArrayList) y se llamará a guardar()

- **void eliminarFunko(int id):** Se eliminará el producto con el id indicado si existe y se llamará a guardar(). Si no, no se hará nada.
- **void guardar():** Almacenará toda la lista de productos en el fichero (DataOutputStream).

**NOTA:** La parte de llamar a guardar() en las altas, bajas y modificaciones podéis dejarla comentada mientras hacéis las pruebas.

## Servicio

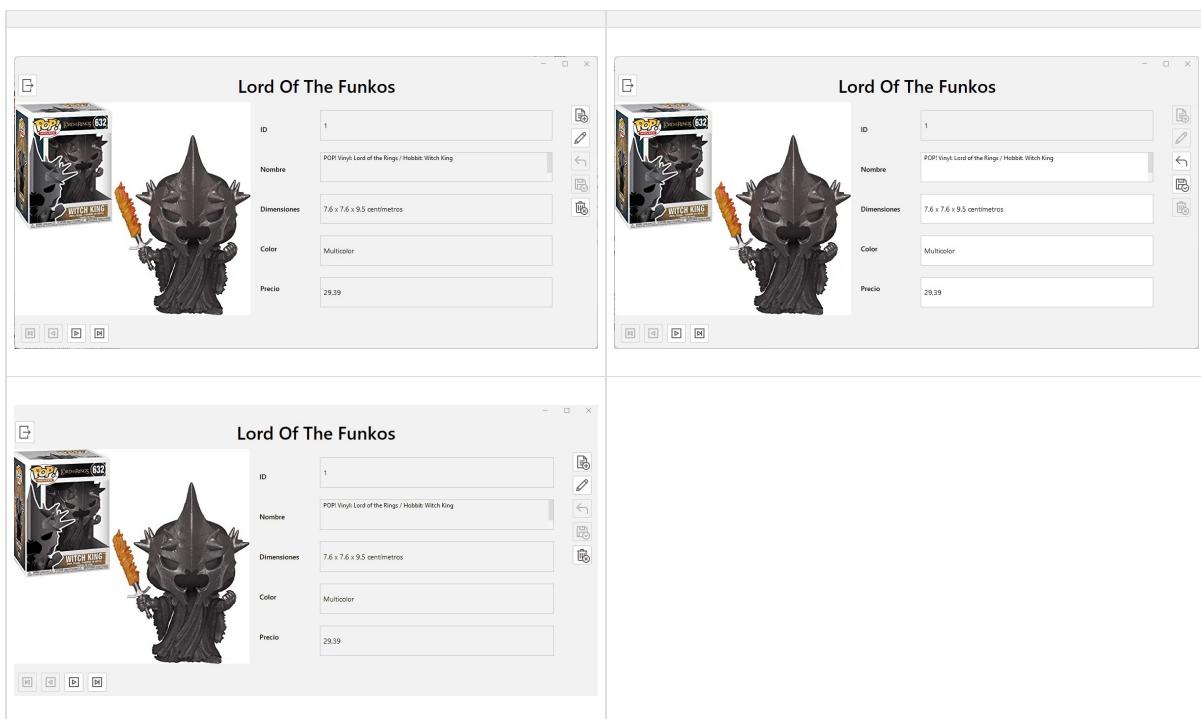
La clase de servicio (`FunkoServicio`) tendrá los mismos métodos que el DAO. En ella se definirá una propiedad de tipo `FunkoDAO` a la que se asignará un objeto de tipo `FunkoFich` y que se empleará para el código de los métodos de la clase.

## Aplicación

La aplicación se ejecutará en un JFrame que dispondrá, además de las propiedades asociadas a los campos de la ventana y al servicio (un objeto de tipo `FunkoServicio`), de tres constantes enteras llamadas `ALTA`, `MODIFICACION` y `CONSULTA` que representarán los tres posibles estados de la aplicación y de una propiedad (también entera) llamada `estado que`, inicialmente, tendrá como valor `CONSULTA`.

Además se definirá un propiedad para almacenar los productos (`List<Producto>`) y otra propiedad entera llamada `pos` donde se almacenará la posición actual en la lista de productos (inicialmente su valor será 0).

Al ejecutar la aplicación y, en función del estado, su aspecto será similar al siguiente (en modo CONSULTA, MODIFICACION y ALTA):



Se definirán los siguientes métodos:

### Constructor

En el constructor de la clase:

- Se asignarán los productos recuperados del servicio a la propiedad `productos` de la clase.
- Se añadirán los controles a la ventana (el control `precio` es un `JFormattedTextField` con formato numérico)
- Al final del constructor de la ventana se llamará a un método `mostrar()` (ver siguiente apartado)

### mostrar()

En este método:

- Si el estado es `CONSULTA` pondrá los controles `nombre`, `dimensiones`, `color` y `precio` como no editables (método `setEditable(boolean)`) para que los usuarios no puedan modificarlos
- Si no lo es, los pondrá como editables
- Se pondrán los botones como activos (`setEnabled(boolean)`) o no en función del estado y la posición:
- Los botones para ir al primer producto y al producto anterior sólo estarán activos en estado `CONSULTA` y si no estamos en el primer producto (`pos`)
- Los botones para ir al siguiente y al último producto sólo estarán activos en estado `CONSULTA` y si no estamos en el último producto
- Los botones nuevo, modificar y eliminar sólo estarán activos en estado `CONSULTA`
- Los botones cancelar y guardar sólo estarán activos en estado `ALTA` o `MODIFICACION`
- Si el estado es `ALTA` pondrá `ID`, `nombre`, `dimensiones`, `color` en blanco y asignará `0.0` al precio. Además se asignará la imagen `noimage.jpg` de la carpeta a la propiedad `icon` (`setIcon`) de la etiqueta donde está la imagen. NOTA: Adicionalmente se pondrá el valor `noimage.jpg` en la propiedad `name` de dicha etiqueta (`setName("noimage.jpg")`)
- Si no lo es, con los datos del producto que está en la posición `pos` (`get(pos)`) se rellenarán los controles del formulario. NOTA: Para la imagen, en el campo deberemos emplear una etiqueta a la que le asignaremos como icono (`setIcon`) un nuevo `ImageIcon` cuya ruta será `imagenes/foto` (donde `foto` es el campo correspondiente del producto). Además, se pondrá en la propiedad `name` de dicha etiqueta el valor del campo `foto`

**NOTA:** Si ejecutamos la aplicación, su aspecto debería ser el de la primera imagen. Si cambiamos el estado a MODIFICACION el de la segunda y si cambiamos a ALTA el de la tercera.

Una vez creado el formulario vamos a añadir la gestión de eventos:

#### Desplazamiento

Al pulsar los botones de desplazamiento, se cambiará adecuadamente el valor de pos y se llamará a mostrar(). Por ejemplo, si se pulsa el botón primero se asignará 0 a pos, si se pulsa último se le asignará la última posición (el tamaño de la lista menos 1), para siguiente se le asignará uno más y para anterior uno menos.

La salida será similar a la siguiente:



#### 5.14. Modificación de la Aplicación de Contactos

### Implementación de Operaciones CRUD en la Aplicación de Gestión de Empleados

A continuación, se detalla cómo añadir las funcionalidades de crear, modificar y eliminar empleados a nuestra aplicación de gestión de contactos.

#### Modificación del Interface DAO

Primero, ampliamos la interfaz `EmpleadoDAO` para incluir las operaciones CRUD:

```
public interface EmpleadoDAO {
 List<Empleado> getEmpleados();
 Empleado getEmpleado(int id);

 Empleado nuevoEmpleado(Empleado e);
 void modificarEmpleado(Empleado e);
 void eliminarEmpleado(Integer id);
}
```

#### Implementación de las Operaciones CRUD

Modificamos la clase `EmpleadoImpl` para implementar los nuevos métodos:

```
public class EmpleadoImpl implements EmpleadoDAO {
 private List<Empleado> empleados = new ArrayList<>();
 private DateFormat df = new SimpleDateFormat("dd/MM/yyyy");

 public EmpleadoImpl() {
 // Constructor existente...
 }

 @Override
 public List<Empleado> getEmpleados() {
 return empleados;
 }

 @Override
 public Empleado getEmpleado(int id) {
 return empleados.stream()
 .filter(e -> e.getId() == id)
 .findAny()
 .orElse(null);
 }

 @Override
 public Empleado nuevoEmpleado(Empleado e) {
 int id = empleados.stream()
 .map(emp -> emp.getId())
 .max((id1,id2) -> id1.compareTo(id2))
 .orElse(0);
 e.setId(++id);
 empleados.add(e);
 guardarCambiosFichero();
 return e;
 }

 @Override
```

```

public void modificarEmpleado(Empleado e) {
 int pos = empleados.indexOf(e);
 empleados.set(pos, e);
 guardarCambiosFichero();
}

@Override
public void eliminarEmpleado(Integer id) {
 Empleado borrar = new Empleado();
 borrar.setId(id);
 empleados.remove(borrar);
 guardarCambiosFichero();
}

private void guardarCambiosFichero() {
 DataOutputStream dos = null;
 try {
 dos = new DataOutputStream(
 new BufferedOutputStream(
 new FileOutputStream("empleados.dat")));
 for(Empleado e : empleados) {
 dos.writeInt(e.getId()); // Id
 dos.writeUTF(e.getApellidos()); // Apellidos
 dos.writeUTF(e.getNombre()); // Nombre
 dos.writeUTF(e.getCargo()); // Cargo
 dos.writeInt(e.getJefe()); // Jefe
 dos.writeUTF(df.format(e.getFechaNacimiento())); // Fecha Nacimiento
 dos.writeUTF(df.format(e.getFechaAlta())); // Fecha Alta
 dos.writeUTF(e.getDireccion()); // Dirección
 dos.writeUTF(e.getCiudad()); // Ciudad
 dos.writeUTF(e.getEstado()); // Estado
 dos.writeUTF(e.getPais()); // País
 dos.writeUTF(e.getCodigoPostal()); // Código Postal
 dos.writeUTF(e.getTelefono()); // Teléfono
 dos.writeUTF(e.getFax()); // Fax
 dos.writeUTF(e.getEmail()); // Email
 dos.writeUTF(e.getFoto()); // Foto
 }
 dos.flush();
 } catch (FileNotFoundException ex) {
 JOptionPane.showMessageDialog(null,
 "Error al escribir los datos
Asegúrese que el fichero 'empleados.dat' está en la misma carpeta que la aplicación.");
 } catch (IOException ex) {
 JOptionPane.showMessageDialog(null,
 "Se ha producido un error inesperado al escribir el fichero
 "No Encontrado",
 JOptionPane.WARNING_MESSAGE);
 }
}

```

**Nota:** Para hacer pruebas es conveniente comentar la línea de `guardarCambiosFichero()` en todos los casos.

## Formulario de Edición de Empleados

Creamos un nuevo formulario para editar los datos de los empleados:

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.border.EmptyBorder;
import javax.swing.border.LineBorder;
import mdlaf.MaterialLookAndFeel;
import mdlaf.themes.JMarsDarkTheme;
import org.zabalburu.agenda.modelo.Empleado;
import org.zabalburu.agenda.service.EmpleadoService;

public class EmpleadosFormGBL extends JFrame {
 private static EmpleadoService servicio = new EmpleadoService();

 private JTextField txtApellidos, txtNombre, txtCargo, txtJefe,
 txtFechaNacimiento, txtFechaAlta, txtDireccion, txtCiudad,
 txtEstado, txtPais, txtCodigoPostal, txtTelefono, txtFax, txtEmail, txtFoto;

 private JLabel lblApellidos, lblNombre, lblCargo, lblJefe,
 lblFechaNacimiento, lblFechaAlta, lblDireccion, lblCiudad,
 lblEstado, lblPais, lblCodigoPostal, lblTelefono, lblFax, lblEmail, lblFoto;

```

```

private Dimension dmVentana = new Dimension(700, 400);

private JLabel lblId = new JLabel();
private JLabel lblTitulo = new JLabel();
JPanel pnlBarra = new JPanel(new BorderLayout());
private JLabel lblCerrar = new JLabel("X");

JPanel panelFormulario = new JPanel(new GridLayout(0,4,10,10));
JPanel panelBotones = new JPanel(new FlowLayout(FlowLayout.RIGHT));

private Empleado empleado = null;

public Empleado getEmpleado() {
 return empleado;
}

public void setEmpleado(Empleado empleado) {
 this.empleado = empleado;
}

private DateFormat df = new SimpleDateFormat("dd/MM/yyyy");

private int px,py;

public EmpleadosFormGBL() {
 this(null);
}

public EmpleadosFormGBL(Empleado emp) {
 this.emp;
 this.setTitle("");
 this.setSize(dmVentana);

 // Inicialización de componentes
 txtApellidos = new JTextField();
 txtNombre = new JTextField();
 txtCargo = new JTextField();
 txtJefe = new JTextField();
 txtFechaNacimiento = new JTextField();
 txtFechaAlta = new JTextField();
 txtDireccion = new JTextField();
 txtCiudad = new JTextField();
 txtEstado = new JTextField();
 txtPais = new JTextField();
 txtCodigoPostal = new JTextField();
 txtTelefono = new JTextField();
 txtFax = new JTextField();
 txtEmail = new JTextField();
 txtFoto = new JTextField();

 lblApellidos = new JLabel("Apellidos");
 lblNombre = new JLabel("Nombre");
 lblCargo = new JLabel("Cargo");
 lblJefe = new JLabel("Jefe");
 lblFechaNacimiento = new JLabel("Fecha Nacimiento");
 lblFechaAlta = new JLabel("Fecha Alta");
 lblDireccion = new JLabel("Dirección");
 lblCiudad = new JLabel("Ciudad");
 lblEstado = new JLabel("Estado");
 lblPais = new JLabel("País");
 lblCodigoPostal = new JLabel("Código Postal");
 lblTelefono = new JLabel("Teléfono");
 lblFax = new JLabel("Fax");
 lblEmail = new JLabel("Email");
 lblFoto = new JLabel("Foto");

 JButton btnGuardar = new JButton("Guardar");
 JButton btnVolver = new JButton("Volver");

 // Construcción del formulario
 panelFormulario.add(new JLabel("ID:"));
 panelFormulario.add(lblId);
 panelFormulario.add(lblApellidos);
 panelFormulario.add(txtApellidos);
 panelFormulario.add(lblNombre);
 panelFormulario.add(txtNombre);
 panelFormulario.add(lblCargo);
 panelFormulario.add(txtCargo);
 panelFormulario.add(lblJefe);
 panelFormulario.add(txtJefe);
 panelFormulario.add(lblFechaNacimiento);
 panelFormulario.add(txtFechaNacimiento);
 panelFormulario.add(lblFechaAlta);
 panelFormulario.add(txtFechaAlta);
 panelFormulario.add(lblDireccion);
 panelFormulario.add(txtDireccion);
 panelFormulario.add(lblCiudad);
 panelFormulario.add(txtCiudad);
 panelFormulario.add(lblEstado);
 panelFormulario.add(txtEstado);
 panelFormulario.add(lblPais);
 panelFormulario.add(txtPais);
 panelFormulario.add(lblCodigoPostal);

```

```

panelFormulario.add(txtCodigoPostal);
panelFormulario.add(lblTelefono);
panelFormulario.add(txtTelefono);
panelFormulario.add(new JLabel("Fax:"));
panelFormulario.add(lblEmail);
panelFormulario.add(txtEmail);
panelFormulario.add(lblFoto);
panelFormulario.add(txtFoto);

// Configuración de eventos
btnGuardar.addActionListener(e -> guardarEmpleado());
panelBotones.add(btnGuardar);
btnVolver.addActionListener(e -> {
 empleado = null;
 setVisible(false);
});
panelBotones.add(btnVolver);

// Configurar el diseño principal
setLayout(new BorderLayout());
panelFormulario.setBorder(new EmptyBorder(10, 10, 10, 10));
add(panelFormulario, BorderLayout.CENTER);
add(panelBotones, BorderLayout.SOUTH);

// Configuración de la barra de título personalizada
lblTitulo = new JLabel();
pnlBarra.add(lblTitulo);
lblCerrar.setFont(new Font("Arial", Font.PLAIN, 18));
lblCerrar.setForeground(Color.WHITE);
lblCerrar.setCursor(new Cursor(Cursor.HAND_CURSOR));
lblCerrar.addMouseListener(new MouseAdapter() {
 @Override
 public void mousePressed(MouseEvent e) {
 setVisible(false);
 }
});
pnlBarra.add(lblCerrar, BorderLayout.EAST);
pnlBarra.setBorder(new EmptyBorder(5, 10, 2, 10));

// Configuración para arrastrar la ventana
lblTitulo.addMouseListener(new MouseAdapter() {
 public void mousePressed(MouseEvent me) {
 pX=me.getX();
 pY=me.getY();
 }
});
lblTitulo.addMouseMotionListener(new MouseAdapter() {
 public void mouseDragged(MouseEvent me) {
 setLocation(getLocation().x+me.getX()-pX,getLocation().y+me.getY()-pY);
 }
});
pnlBarra.setOpaque(true);
pnlBarra.setBackground(new Color(100,100,100));
this.add(pnlBarra, BorderLayout.NORTH);

setLocationRelativeTo(null); // Centrar la ventana en la pantalla
setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
this.setUndecorated(true);
mostrarDatos();
}

public static void main(String[] args) {
 try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new JMarsDarkTheme()));
 } catch (UnsupportedLookAndFeelException ex) {
 Logger.getLogger(ContactosForm.class.getName()).log(Level.SEVERE, null, ex);
 }
 new EmpleadosFormGBL(servicio.getEmpleado(1)).setVisible(true);
}

private void mostrarDatos() {
 if (this.empleado == null) {
 lblTitulo.setText("Nuevo Empleado");
 // Alta
 for (Component c : panelFormulario.getComponents()) {
 if (c instanceof JTextField){
 ((JTextField) c).setText("");
 }
 }
 } else {
 lblTitulo.setText(empleado.getApellidos() + ", " + empleado.getNombre());
 lblId.setBorder(new LineBorder(Color.WHITE));
 lblId.setHorizontalAlignment(JLabel.CENTER);
 lblId.setText(this.empleado.getId().toString());
 txtApellidos.setText(empleado.getApellidos());
 txtNombre.setText(empleado.getNombre());
 txtCargo.setText(empleado.getCargo());
 txtJefe.setText(empleado.getJefe().toString());
 txtFechaNacimiento.setText(df.format(empleado.getFechaNacimiento()));
 txtFechaAlta.setText(df.format(empleado.getFechaAlta()));
 txtDireccion.setText(empleado.getDireccion());
 }
}

```

```

 txtCiudad.setText(empleado.getCiudad());
 txtEstado.setText(empleado.getEstado());
 txtPais.setText(empleado.getPais());
 txtCodigoPostal.setText(empleado.getCodigoPostal());
 txtTelefono.setText(empleado.getTelefono());
 txtFax.setText(empleado.getFax());
 txtEmail.setText(empleado.getEmail());
 txtFoto.setText(empleado.getFoto());
 }
}

private void guardarEmpleado() {
 empleado.setApellidos(txtApellidos.getText());
 empleado.setCargo(txtCargo.getText());
 empleado.setCiudad(txtCiudad.getText());
 empleado.setCodigoPostal(txtCodigoPostal.getText());
 empleado.setDireccion(txtDireccion.getText());
 empleado.setEmail(txtEmail.getText());
 empleado.setEstado(txtEstado.getText());
 empleado.setFax(txtFax.getText());
 try {
 empleado.setFechaAlta(df.parse(txtFechaAlta.getText()));
 empleado.setFechaNacimiento(df.parse(txtFechaNacimiento.getText()));
 } catch (ParseException ex) {
 }
 empleado.setFoto(txtFoto.getText());
 empleado.setJefe(Integer.parseInt(txtJefe.getText()));
 empleado.setNombre(txtNombre.getText());
 empleado.setPais(txtPais.getText());
 empleado.setTelefono(txtTelefono.getText());
 System.out.println(empleado);
 this.setVisible(false);
}
}

```

## Modificación del Formulario Principal

Actualizamos el formulario principal para incluir las operaciones CRUD:

```

public class ContactosForm extends JFrame {
 private EmpleadoService service = new EmpleadoService();

 private JPanel pnlDatos;
 private JTextField txtApellidos;
 private JTextField txtNombre;
 private JTextField txtCargo;
 private JTextField txtTelefono;
 private JTextField txtEmail;
 private JLabel lblActual;
 private JLabel lblFoto;

 private JPanel pnlBotones;
 private JButton btnAnterior;
 private JButton btnSiguiente;
 private JButton btnModificar;
 private JButton btnNuevo;
 private JButton btnEliminar;

 JPanel pnlBarra = new JPanel(new BorderLayout());
 private JLabel lblCerrar = new JLabel("X");

 private int pos = 0;

 private int px, py;

 public ContactosForm() {
 IconFontSwing.register(GoogleMaterialDesignIcons.getIconFont());
 inicializarUI();
 }

 private void inicializarUI() {
 setTitle("Contactos");
 // Código existente...

 pnlBotones = new JPanel(new FlowLayout(FlowLayout.RIGHT));
 pnlBotones.setBorder(new EmptyBorder(5, 5, 5, 5));
 // Código existente...

 btnEliminar = new JButton("Eliminar");
 btnEliminar.addActionListener(e -> eliminarEmpleado());
 pnlBotones.add(btnEliminar);

 btnNuevo = new JButton("Nuevo");
 btnNuevo.addActionListener(e -> nuevoEmpleado());
 pnlBotones.add(btnNuevo);

 btnModificar = new JButton("Modificar");
 btnModificar.addActionListener(e -> modificarEmpleado());
 pnlBotones.add(btnModificar);

 // Código existente...
 mostrarContactoActual();
 }
}

```

```

// Métodos existentes...

private void eliminarEmpleado() {
 if (JOptionPane.showOptionDialog(this,
 """
 Eliminar el Empleado?

 Esta operación no se puede deshacer
 """,
 "Eliminar",
 JOptionPane.YES_NO_OPTION,
 JOptionPane.QUESTION_MESSAGE,
 null,
 new Object[] {"Eliminar","Cancelar"},
 "Eliminar")
 ==
 JOptionPane.YES_OPTION) {
 service.eliminarEmpleado(service.getEmpleados().get(pos).getId());
 }
 if (pos == service.getEmpleados().size()) {
 pos--;
 }
 mostrarContactoActual();
}

private void nuevoEmpleado() {
 EmpleadoForm form = new EmpleadoForm();
 form.setVisible(true);
 Empleado empleado = form.getEmpleado();
 if (empleado != null){
 empleado = service.nuevoEmpleado(empleado);
 pos = service.getEmpleados().indexOf(empleado);
 }
 mostrarContactoActual();
}

private void modificarEmpleado() {
 Empleado empleado = service.getEmpleados().get(pos);
 EmpleadoForm form = new EmpleadoForm(empleado);
 form.setVisible(true);
 empleado = form.getEmpleado();
 if (empleado != null){
 service.modificarEmpleado(empleado);
 pos = service.getEmpleados().indexOf(empleado);
 }
 mostrarContactoActual();
}
}

```

## Análisis del Código

### Formulario de Edición

- **Dos constructores:** Uno sin parámetros para altas y otro que recibe un `Empleado` para modificaciones.
- **Propiedad empleado:** Almacena el empleado a modificar o el nuevo empleado.
- **Método mostrarDatos():** Configura los campos según sea alta o modificación.
- **Método guardarEmpleado():** Actualiza los datos del empleado con los valores de los controles.

### Formulario Principal

- **eliminarEmpleado():** Sigue la confirmación y elimina el empleado actual. Ajusta la posición si se elimina el último.
- **nuevoEmpleado():** Muestra el formulario de edición sin empleado, recupera el resultado y actualiza la lista.
- **modificarEmpleado():** Muestra el formulario con el empleado actual, recupera los cambios y actualiza la lista.

## Capturas de Pantalla

### Formulario de Edición en Alta



### Formulario de Edición en Modificación

Adams, Andrew

ID:	1	Nombre:	Andrew
Apellidos:	Adams	Jefe:	0
Cargo:	General Manager	Fecha Alta:	14/08/0002
Fecha Nacimiento:	18/02/0062	Ciudad:	Edmonton
Dirección:	11120 Jasper Ave NW	País:	Canada
Estado:	AB	Teléfono:	+1 (780) 428-9482
Código Postal:	T5K 2N1	Email:	andrew@chinookcorp.com
Fax:	+1 (780) 428-3457	Foto:	/api/portraits/men/75.jpg

**Guardar** **Volver**

### Aplicación Final

Contacto 1 de 8

Apellidos:	Adams	
	Nombre:	Andrew
Cargo:	General Manager	
Teléfono:	+1 (780) 428-9482	
Email:	andrew@chinookcorp.com	

**Anterior** **Siguiente >** **Eliminar** **Nuevo** **Modificar**

### Notas Importantes

- El formulario de edición debe ser modal (extender de `JDialog` y usar `setModal(true)`) para detener la ejecución del formulario principal hasta que se cierre.
- Para la eliminación, se asume que nunca se eliminarán todos los empleados, ya que sería una situación compleja de gestionar.
- El método `guardarCambiosFichero()` puede comentarse durante las pruebas para evitar modificar el archivo de datos.
- La posición en la lista (`pos`) se actualiza después de cada operación para mantener la coherencia en la navegación.

### 5.15. Actividad 28-II - Creación de un Formulario De Productos

#### Instrucciones

Añadir a la aplicación la siguiente funcionalidad

##### Cambio de Estado

Los botones `nuevo`, `modificar` y `cancelar` cambiarán el estado de la aplicación. En el caso del botón `nuevo` se pondrá como estado `ALTA` y se llamará a `mostrar()`. De manera similar, el botón `modificar` pondrá `MODIFICACION` como estado y llamará a `mostrar()`. Por su parte, el botón `cancelar`, pondrá `CONSULTA` como estado y llamará también a `mostrar()`.

##### Guardar

Al pulsar el botón guardar se realizarán las siguientes tareas:

- Se creará un nuevo producto
- Si el estado es `MODIFICACION` se le asignará como `id` el contenido del control **ID** del formulario
- Se asignarán a los campos del producto los valores de los controles correspondientes del formulario (menos el campo `foto` al que se asignará el valor de la propiedad `name` de la etiqueta)
- Si estamos en estado `ALTA`:
- Llamaremos al método `nuevoFunko` del servicio con el **producto** que hemos modificado
- Pondremos a `pos` el valor de la **última posición** de la lista
- Pasaremos a estado `CONSULTA`
- Llamaremos a `mostrar()`
- Si no
- Llamaremos al método `modificarFunko` del servicio con el producto que hemos modificado
- Pasaremos a estado `CONSULTA`
- Llamaremos a `mostrar()`

El resultado hasta el momento:

## Lord Of The Funkos

	<p><b>ID</b> <input type="text" value="1"/></p> <p><b>Nombre</b> <input type="text" value="POP! Vinyl: Lord of the Rings / Hobbit: Witch King"/></p> <p><b>Dimensiones</b> <input type="text" value="7.6 x 7.6 x 9.5 centímetros"/></p> <p><b>Color</b> <input type="text" value="Multicolor"/></p> <p><b>Precio</b> <input type="text" value="29.39"/></p>
-----------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Adicional

Añadir **validación** para que no se puedan añadir productos a los que les faltén datos

### Borrar

Al pulsar el botón **borrar**, se realizarán las siguientes tareas:

- Se pedirá **confirmación** para eliminar el producto
- Si se da
- Se obtendrá el **id** del producto del control `ID` y se llamará al método `eliminarFunko` del servicio con dicho `id`
- **Si tras borrar** la posición `pos` coincide con el **tamaño actual** de la lista quiere decir que **acabamos de eliminar el último producto**. En este caso, **restaremos uno a pos**
- Llamaremos a `mostrar()`

Tras estos cambios :

## Lord Of The Funkos

	<p><b>ID</b> <input type="text" value="1"/></p> <p><b>Nombre</b> <input type="text" value="POP! Vinyl: Lord of the Rings / Hobbit: Witch King"/></p> <p><b>Dimensiones</b> <input type="text" value="7.6 x 7.6 x 9.5 centímetros"/></p> <p><b>Color</b> <input type="text" value="Multicolor"/></p> <p><b>Precio</b> <input type="text" value="29.39"/></p>
-------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### AVANZADO

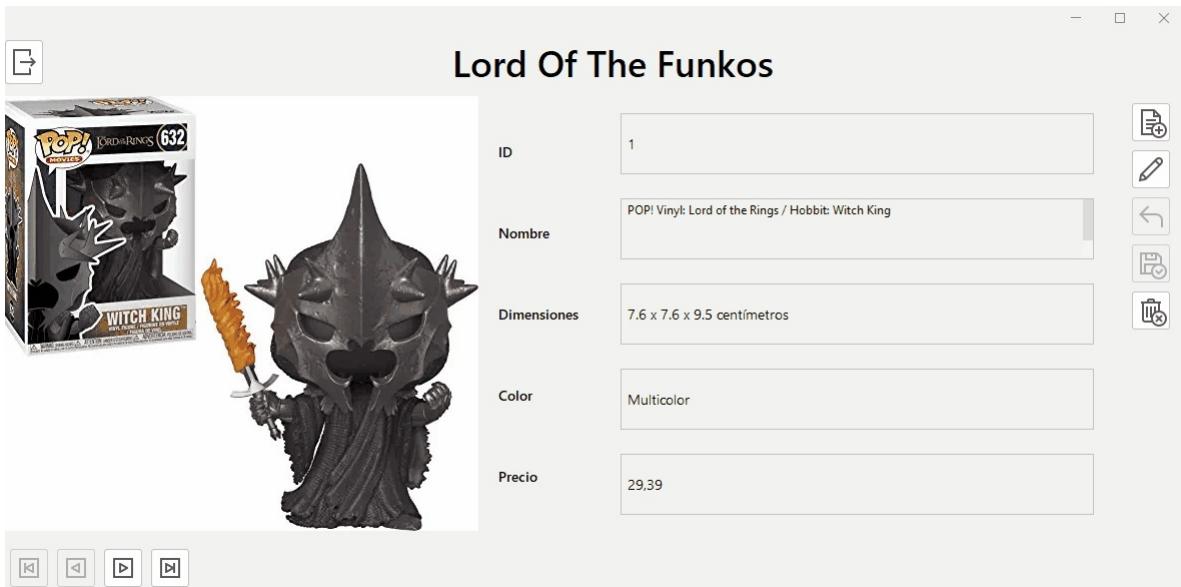
#### Modificar la Foto

Queremos que **el usuario pueda cambiar la foto de un producto** (sólo en estado `ALTA` o `MODIFICACION`). Para ello vamos a hacer que, **cuando se haga clic en la imagen** es estos estados:

- Se muestre un cuadro de diálogo para que el usuario seleccione la nueva foto (previo si sólo se permite seleccionar imágenes JPG o PNG). Por simplicidad vamos a situar previamente las nuevas fotos en la carpeta `imágenes`

- Al seleccionar la foto, se asigne como imagen a la etiqueta y se ponga su nombre en la propiedad name.
  - Conseguir que aparezca el cursor con la mano cuando se puede modificar.

El resultado:



#### **Problema si se borran todos los productos**

Tal y como está la aplicación, no funciona bien si se borran todos los productos. Descubrir por qué y solucionarlo.

## Guardar los Cambios

Una vez hachas todas las pruebas, reactivar en el DAO la opción para guardar los cambios en el fichero. Comprobar que los cambios en los productos son, ahora, permanentes

### 5.16. Más Gestores de Distribución GridBagLayout

## GridBagLayout

El gestor de diseño más complejo. Es similar a un `GridLayout` en el que un **componente puede ocupar varias celdas**, aunque en este caso las filas no tienen por qué tener la misma altura, ni las columnas la misma anchura y además, los componentes pueden ocupar varias celdas. En esencia, se coloca el componente en una celda y se usa su tamaño preferido para ver cómo debe ser la celda de grande. La forma en la que los componentes emplean el espacio disponible al redimensionar la ventana se especifica mediante **restricciones** que no son sino **propiedades** de un objeto  `GridBagConstraints` que es utilizado por el `GridBagLayout` a la hora de decidir su tamaño y cómo se debe redimensionar. La forma de trabajar es la siguiente:

- Se crea un objeto `GridBagLayout` (`gbl`)
  - Se crea un objeto `GridBagConstraints` (`gbc`)
  - Se modifican las propiedades del segundo objeto para el componente a añadir. Para ello, el objeto `GridBagConstraints` tiene una serie de **propiedades** a las que se pueden asignar diferentes valores para indicar cómo se debe comportar el componente (`gbc.propiedad = valor`).
  - Se asignan las restricciones al componente a través del `gbl` creado previamente (`gbl.setConstraints(componente, gbc)`)
  - Se añade el componente al objeto (`panel.add(componente)`)

Se puede reutilizar el mismo objeto de restricciones para todos los componentes, modificando en cada uno de ellos lo que sea necesario. Veamos un ejemplo:

```
package net.zabalburu.pruebaswing;
```

```
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class EjemploGridBag extends JFrame {
 private GridBagLayout grid = new GridBagLayout();
 private JPanel pnl1 = new JPanel(grid);
 private JLabel lblNombre = new JLabel("Nombre:");
 private JTextField txtNombre = new JTextField();
 private JLabel lblObservaciones = new JLabel("Observaciones:");
 private JTextArea txaObservaciones = new JTextArea();
 private JButton btnAceptar = new JButton("Aceptar");
 private JButton btnCancelar = new JButton("Cancelar");
 private JScrollPane jspObservaciones = new JScrollPane(txaObservaciones);
```

```

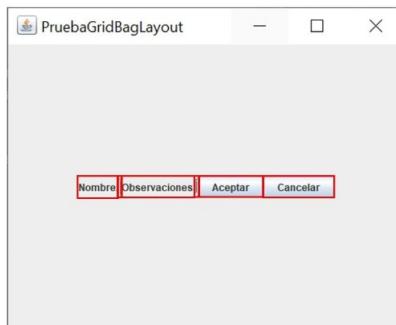
public EjemploGridBag() {
 this.setSize(500,400);
 this.setLocation(200,200);
 this.setTitle("PruebaGridBagLayout");
 pnl.add(lblNombre);
 pnl.add(txtNombre);
 pnl.add(lblObservaciones);
 pnl.add(jspObservaciones);
 pnl.add(btnAceptar);
 pnl.add(btnCancelar);
 this.add(pnl);
 this.setVisible(true);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public static void main(String[] args) {
 new EjemploGridBag();
}

}

```

La salida (en la imagen se ha resaltado la distribución en celdas de los componentes):



Como se puede ver por defecto los componentes se sitúan uno detrás de otro en una única fila, empleando su tamaño preferido. El alto de la fila se toma del mayor alto de todos los componentes (en este caso los botones) y cada componente toma como ancho, el ancho preferido del mismo. Además los todo el espacio restante se distribuye alrededor de dicha fila de modo que los componentes aparezcan centrados en horizontal y en vertical. El cuadro de texto y el área de texto prácticamente no se ven dado que su ancho preferido es el mínimo para mostrar el borde. Aunque podríamos emplear `setPreferredSize` para modificar este comportamiento no los vamos a mostrar así sino a través de las restricciones como vamos a ir viendo poco a poco.

#### Posicionamiento

Lo primero que podemos cambiar en el diseño que tenemos es la posición de los diferentes componentes. En este sentido hay que tener en cuenta que, el número de filas y columnas depende del número de componentes que tengamos dado que no se deben dejar filas y/o columnas sin ningún componente. Para modificar la posición de los componentes emplearemos las propiedades `gridx` y `gridy` del objeto `GridBagConstraints`:

```

package net.zabalburu.pruebaswing;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class EjemploGridBag extends JFrame {
 private GridBagLayout grid = new GridBagLayout();
 private JPanel pnl = new JPanel(grid);
 private JLabel lblNombre = new JLabel("Nombre:");
 private JTextField txtNombre = new JTextField();
 private JLabel lblObservaciones = new JLabel("Observaciones:");
 private JTextArea txaObservaciones = new JTextArea();
 private JButton btnAceptar = new JButton("Aceptar");
 private JButton btnCancelar = new JButton("Cancelar");
 private JScrollPane jspObservaciones = new JScrollPane(txaObservaciones);
 private GridBagConstraints gbc = new GridBagConstraints(); // AÑADIMOS EL OBJETO PARA LAS RESTRICCIONES
 public EjemploGridBag() {
 this.setSize(500,400);
 this.setLocation(200,200);
 this.setTitle("PruebaGridBagLayout");

 gbc.gridx = 0;
 gbc.gridy = 0;
 grid.setConstraints(lblNombre, gbc);
 pnl.add(lblNombre);

 gbc.gridx = 1;
 gbc.gridy = 0; // No haría falta dado que estamos empleando el mismo objeto
 grid.setConstraints(txtNombre, gbc);
 pnl.add(txtNombre);

 gbc.gridx = 0;

```

```

gbc.gridx = 1;
grid.setConstraints(lblObservaciones, gbc);
pnl.add(lblObservaciones);

gbc.gridx = 0;
gbc.gridy = 2;
grid.setConstraints(jspObservaciones, gbc);
pnl.add(jspObservaciones);

gbc.gridx = 0;
gbc.gridy = 3;
grid.setConstraints(btnAceptar, gbc);
pnl.add(btnAceptar);

gbc.gridx = 1;
gbc.gridy = 3;
grid.setConstraints(btnCancelar, gbc);
pnl.add(btnCancelar);

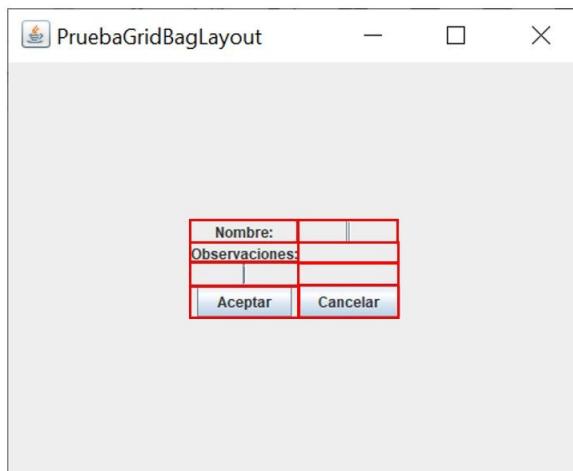
this.add(pnl);
this.setVisible(true);
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

}

public static void main(String[] args) {
 new EjemploGridBag();
}
}

```

La salida:



Como se puede ver, los componentes siempre se sitúan en celdas. La altura de la fila la determina el componente más alto en dicha fila (el caso de la primera fila es la altura del cuadro de texto) y la anchura de cada columna la determina el componente con mayor anchura en la misma (la etiqueta `Observaciones` en la primera columna y el botón `Cancelar` en la segunda). Aunque en la imagen no se aprecia demasiado, **los componentes se centran tanto en horizontal como en vertical dentro de su celda**.

NOTA: A la hora de especificar `gridx` y `gridy` se puede emplear la constante `GridBagConstraints.RELATIVE` (que es el valor por defecto para `gridx`) que indica que se emplea la siguiente columna (`gridx`) o fila (`gridy`) en base al último componente añadido. En cualquier caso, emplea este valor hace el código más difícil de controlar.

#### Tamaño

Como ya se ha comentado, los componentes en un `GridLayout` pueden ocupar varias celdas (tanto en horizontal como en vertical) pero siempre teniendo en cuenta cuántas filas / columnas hay (en función de los componentes). Para especificar el ancho / alto en celdas de un componente se emplean las propiedades `gridwidth` y `gridheight` (por defecto valen 1). En nuestro ejemplo tenemos cuatro filas y dos columnas. Vamos a hacer que la etiqueta y el área de texto ocupen dos columnas:

```

package net.zabaluru.pruebaswing;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class EjemploGridBag extends JFrame {
 private GridBagLayout grid = new GridBagLayout();
 private JPanel pnl = new JPanel(grid);
 private JLabel lblNombre = new JLabel("Nombre:");
 private JTextField txtNombre = new JTextField();
 private JLabel lblObservaciones = new JLabel("Observaciones:");
 private JTextArea txaObservaciones = new JTextArea();

```

```

private JButton btnAceptar = new JButton("Aceptar");
private JButton btnCancelar = new JButton("Cancelar");
private JScrollPane jspObservaciones = new JScrollPane(txaObservaciones);
private GridBagConstraints gbc = new GridBagConstraints();
public EjemploGridBag() {
 this.setSize(500,400);
 this.setLocation(200,200);
 this.setTitle("PruebaGridBagLayout");

 gbc.gridx = 0;
 gbc.gridy = 0;
 grid.setConstraints(lblNombre, gbc);
 pnl.add(lblNombre);

 gbc.gridx = 1;
 gbc.gridy = 0; // No haria falta dado que estamos empleando el mismo objeto
 grid.setConstraints(txtNombre, gbc);
 pnl.add(txtNombre);

 gbc.gridx = 0;
 gbc.gridy = 1;
 gbc.gridwidth = 2; // La etiqueta ocupa DOS COLUMNAS
 grid.setConstraints(lblObservaciones, gbc);
 pnl.add(lblObservaciones);

 gbc.gridx = 0;
 gbc.gridy = 2;
 grid.setConstraints(jspObservaciones, gbc); // El textarea SIGUE OCUPANDO DOS COLUMNAS
 pnl.add(jspObservaciones);

 gbc.gridx = 0;
 gbc.gridy = 3;
 gbc.gridwidth = 1; // Se resatblece el ANCHO A UNA COLUMNA para el resto de componentes
 grid.setConstraints(btnAceptar, gbc);
 pnl.add(btnAceptar);

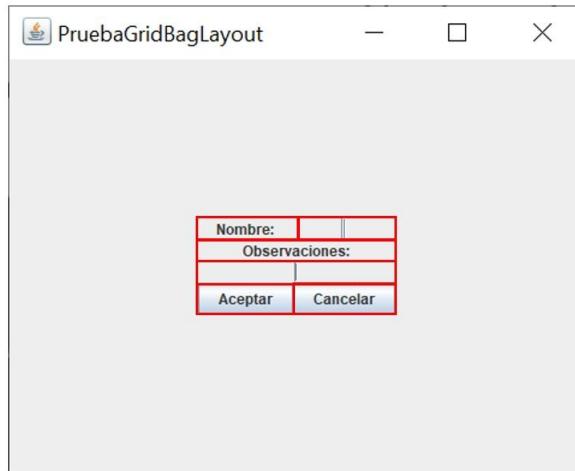
 gbc.gridx = 1;
 gbc.gridy = 3;
 grid.setConstraints(btnCancelar, gbc);
 pnl.add(btnCancelar);

 this.add(pnl);
 this.setVisible(true);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public static void main(String[] args) {
 new EjemploGridBag();
}
}

```

La salida:



¿Qué ocurre en este caso? Los altos de las filas no se modifican desde el ejemplo anterior, pero los anchos de las columnas sí. En el caso de la primera columna ahora sólo se tienen en cuenta los anchos de la etiqueta Nombre y del botón Aceptar (dado que las observaciones no acaban en esta columna). Como el botón es más ancho, se toma como ancho de la primera columna el del botón. En cuanto a la segunda columna, se consideran el cuadro de texto, la etiqueta y el área de texto de Observaciones y el botón Cancelar. De todos ellos (y teniendo en cuenta el ancho de la primera columna) el componente más ancho es el botón Cancelar por lo que ese se toma como ancho de la segunda columna. Es decir el espacio necesario para mostrar el botón Cancelar sería el ancho de la primera columna más el ancho del propio botón. Este espacio es claramente mayor que el necesario para mostrar la etiqueta de Observaciones por lo que es el ancho que se toma. Una vez más, los componentes se centran en horizontal y vertical en sus respectivas celdas.

#### Alineación

Como hemos visto, los componentes se centran en sus celdas en horizontal y vertical. Este comportamiento lo podemos modificar con la propiedad anchor del objeto GridBagConstraints que admite los siguientes valores:

- `GridBagConstraints.NORTH / GridBagConstraints.PAGE_START`: En vertical, en la parte superior y centrado en horizontal
- `GridBagConstraints.NORTHEAST / GridBagConstraints.FIRST_LINE_END`: En vertical, en la parte superior y en la parte derecha en horizontal
- `GridBagConstraints.EAST / GridBagConstraints.LINE_END`: En vertical centrado y en la parte derecha en horizontal
- `GridBagConstraints.SOUTHEAST / GridBagConstraints.LAST_LINE_END`: En vertical, en la parte inferior y en la parte derecha en horizontal
- `GridBagConstraints.SOUTH / GridBagConstraints.PAGE_END`: En vertical, en la parte inferior y en la parte derecha centrado
- `GridBagConstraints.SOUTHWEST / GridBagConstraints.LAST_LINE_START`: En vertical, en la parte inferior y en la parte izquierda en horizontal
- `GridBagConstraints.WEST / GridBagConstraints.LINE_START`: En vertical centrado y en la parte izquierda en horizontal
- `GridBagConstraints.NORTHWEST / GridBagConstraints.FIRST_LINE_START`: En vertical, en la parte superior y en la parte izquierda en horizontal
- `GridBagConstraints.CENTER`: Centrado en horizontal y en vertical (valor por defecto)

Modificamos la alineación de los componentes para que se sitúen al OESTE:

```
package net.zabalburu.pruebaswing;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class EjemploGridBag extends JFrame {
 private GridBagLayout grid = new GridBagLayout();
 private JPanel pnl = new JPanel(grid);
 private JLabel lblNombre = new JLabel("Nombre:");
 private JTextField txtNombre = new JTextField();
 private JLabel lblObservaciones = new JLabel("Observaciones:");
 private JTextArea txaObservaciones = new JTextArea();
 private JButton btnAceptar = new JButton("Aceptar");
 private JButton btnCancelar = new JButton("Cancelar");
 private JScrollPane jspObservaciones = new JScrollPane(txaObservaciones);
 private GridBagConstraints gbc = new GridBagConstraints();
 public EjemploGridBag() {
 this.setSize(500,400);
 this.setLocation(200,200);
 this.setTitle("PruebaGridBagLayout");

 gbc.gridx = 0;
 gbc.gridy = 0;
 gbc.anchor = GridBagConstraints.WEST; // A partir de aquí, todos los componentes a la derecha
 grid.setConstraints(lblNombre, gbc);
 pnl.add(lblNombre);

 gbc.gridx = 1;
 gbc.gridy = 0;
 grid.setConstraints(txtNombre, gbc);
 pnl.add(txtNombre);

 gbc.gridx = 0;
 gbc.gridy = 1;
 gbc.gridwidth = 2;
 grid.setConstraints(lblObservaciones, gbc);
 pnl.add(lblObservaciones);

 gbc.gridx = 0;
 gbc.gridy = 2;
 grid.setConstraints(jspObservaciones, gbc);
 pnl.add(jspObservaciones);

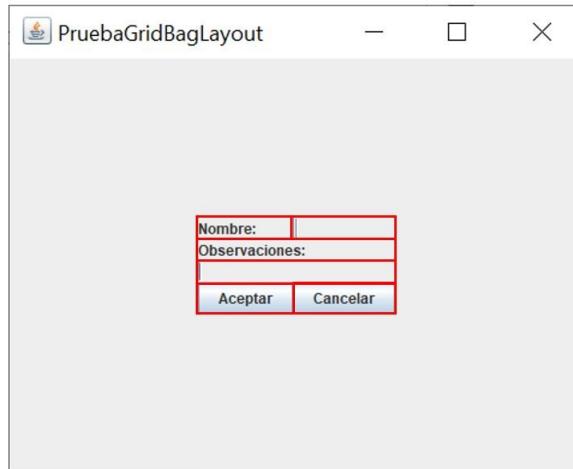
 gbc.gridx = 0;
 gbc.gridy = 3;
 gbc.gridwidth = 1;
 grid.setConstraints(btnAceptar, gbc);
 pnl.add(btnAceptar);

 gbc.gridx = 1;
 gbc.gridy = 3;
 grid.setConstraints(btnCancelar, gbc);
 pnl.add(btnCancelar);

 this.add(pnl);
 this.setVisible(true);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
 }

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

La salida:



### Espaciado

En cuanto al espaciado, podemos especificar dos tipos de espaciado: el **espaciado interno y el externo**. El espaciado interno hace más grande el componente, mientras que el espaciado externo funciona como un margen. Para modificar el espaciado interno de un componente tenemos las propiedades `ipadx` e `ipady` que especifican cuánto crece el componente en horizontal y en vertical. El espaciado exterior se especifica mediante la propiedad `insets` que admite un objeto de tipo `Insets` en el que se puede especificar el espaciado superior, izquierda, inferior y derecha a dejar entre la celda y el componente. En el ejemplo vamos a dejar espaciado entre todos los componentes de 5 pixels en cada dirección (10 entre componentes) y haremos los botones más grandes.

```
package net.zabalburu.pruebaswing;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class EjemploGridBag extends JFrame {
 private GridBagLayout grid = new GridBagLayout();
 private JPanel pnl = new JPanel(grid);
 private JLabel lblNombre = new JLabel("Nombre:");
 private JTextField txtNombre = new JTextField();
 private JLabel lblObservaciones = new JLabel("Observaciones:");
 private JTextArea txaObservaciones = new JTextArea();
 private JButton btnAceptar = new JButton("Aceptar");
 private JButton btnCancelar = new JButton("Cancelar");
 private JScrollPane jspObservaciones = new JScrollPane(txaObservaciones);
 private GridBagConstraints gbc = new GridBagConstraints();
 public EjemploGridBag() {
 this.setSize(500,400);
 this.setLocation(200,200);
 this.setTitle("PruebaGridLayout");

 gbc.gridx = 0;
 gbc.gridy = 0;
 gbc.anchor = GridBagConstraints.WEST;
 gbc.insets = new Insets(5,5,5,5);
 grid.setConstraints(lblNombre, gbc);
 pnl.add(lblNombre);

 gbc.gridx = 1;
 gbc.gridy = 0;
 grid.setConstraints(txtNombre, gbc);
 pnl.add(txtNombre);

 gbc.gridx = 0;
 gbc.gridy = 1;
 gbc.gridwidth = 2;
 grid.setConstraints(lblObservaciones, gbc);
 pnl.add(lblObservaciones);

 gbc.gridx = 0;
 gbc.gridy = 2;
 grid.setConstraints(jspObservaciones, gbc);
 pnl.add(jspObservaciones);

 gbc.gridx = 0;
 gbc.gridy = 3;
 gbc.gridwidth = 1;
 gbc.ipadx = 10;
 gbc.ipady = 5;
 }
}
```

```

 grid.setConstraints(btnAceptar, gbc);
 pnl.add(btnAceptar);

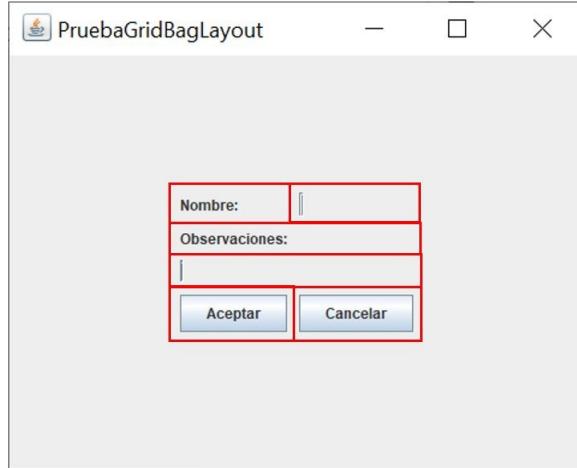
 gbc.gridx = 1;
 gbc.gridy = 3;
 grid.setConstraints(btnCancelar, gbc);
 pnl.add(btnCancelar);

 this.add(pnl);
 this.setVisible(true);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
 }

 public static void main(String[] args) {
 new EjemploGridBag();
 }
}

```

La salida



Como se puede ver las celdas aumentan su tamaño y los componentes quedan más separados entre si. Por otro lado los botones se han hecho más grandes gracias al nuevo espacio interior.

#### Espacio libre

Ahora tenemos que decidir qué hacer con el espacio libre que queda después de situar los componentes. Como se puede ver dicho espacio se reparte equitativamente alrededor de los componentes.

Para repartir el espacio libre tenemos las propiedades `weightx` (para el espacio libre horizontal) y `weighty` (para el vertical). Ambas propiedades se pueden asignar a cada componente pero hay que tener cuidado al hacerlo si queremos controlar cómo se reparte dicho espacio.

Si nos centramos en el espacio horizontal (`weightx`), cada componente tiene su propio peso y el espacio se reparte proporcionalmente en función del mismo. Si el peso es 0 no se asigna espacio adicional a dicho componente.

Lo mejor es centrarse en los componentes de una fila y el resto dejarlo a 0.

Por ejemplo, si cogemos la primera fila tenemos dos componentes: la etiqueta y el cuadro de texto. Si `w1` es el peso de la etiqueta y `w2` es el del cuadro de texto y `sp` es el espacio libre, a la etiqueta se le asignará: `sp1 = w1 / (w1+w2) * sp` y al cuadro de texto `sp2 = w2 / (w1+w2) * sp`. Si `w1` es 2, `w2` es 3 y el espacio disponible son 200 px `sp1 = 2 / 5 * 200 --> 80 pixels` `sp3 = 3 / 5 * 200 --> 120 pixels`. Es decir dos quintos del espacio van para la etiqueta y tres quintos para el cuadro de texto. Igual resulta más sencillo pensar en porcentajes. Por ejemplo, si queremos que el cuadro de texto ocupe el 80% del espacio libre y la etiqueta el 20% podemos asignarles 80 y 20 respectivamente.

NOTA: Si un componente ocupa varias columnas y se le asigna espacio libre este espacio se asignará únicamente a la última columna.

Modificamos el peso en el eje x tal y como hemos dicho:

```

package net.zabalburu.pruebaswing;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

```

```

public class EjemploGridBag extends JFrame {
 private GridBagLayout grid = new GridBagLayout();
 private JPanel pnl = new JPanel(grid);
 private JLabel lblNombre = new JLabel("Nombre:");
 private JTextField txtNombre = new JTextField();

```

```

private JLabel lblObservaciones = new JLabel("Observaciones:");
private JTextArea txaObservaciones = new JTextArea();
private JButton btnAceptar = new JButton("Aceptar");
private JButton btnCancelar = new JButton("Cancelar");
private JScrollPane jspObservaciones = new JScrollPane(txaObservaciones);
private GridBagConstraints gbc = new GridBagConstraints();
public EjemploGridBag() {
 this.setSize(500,400);
 this.setLocation(200,200);
 this.setTitle("PruebaGridBagLayout");

 gbc.gridx = 0;
 gbc.gridy = 0;
 gbc.anchor = GridBagConstraints.WEST;
 gbc.insets = new Insets(5,5,5,5);
 gbc.weightx = 20; // Un 20% del espacio
 grid.setConstraints(lblNombre, gbc);
 pnl.add(lblNombre);

 gbc.gridx = 1;
 gbc.gridy = 0;
 gbc.weightx = 80; // Un 80% del espacio
 grid.setConstraints(txtNombre, gbc);
 pnl.add(txtNombre);

 gbc.gridx = 0;
 gbc.gridy = 1;
 gbc.gridwidth = 2;
 gbc.weightx = 0; // Lo desactivamos para el resto
 grid.setConstraints(lblObservaciones, gbc);
 pnl.add(lblObservaciones);

 gbc.gridx = 0;
 gbc.gridy = 2;
 grid.setConstraints(jspObservaciones, gbc);
 pnl.add(jspObservaciones);

 gbc.gridx = 1;
 gbc.gridy = 3;
 grid.setConstraints(btnAceptar, gbc);
 pnl.add(btnAceptar);

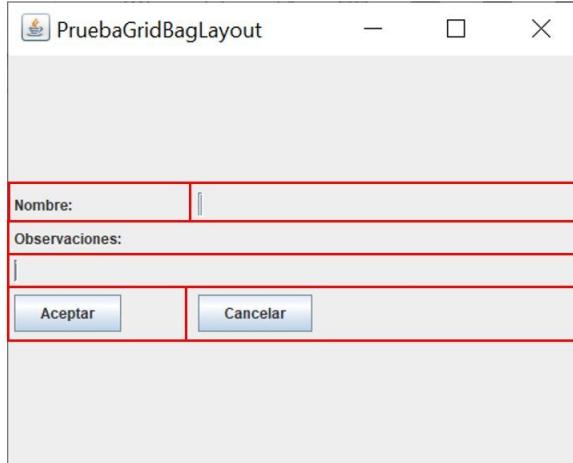
 gbc.gridx = 1;
 gbc.gridy = 3;
 grid.setConstraints(btnCancelar, gbc);
 pnl.add(btnCancelar);

 this.add(pnl);
 this.setVisible(true);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public static void main(String[] args) {
 new EjemploGridBag();
}
}

```

La salida:



En el caso del eje y, hacemos algo similar pero cogiendo los componentes de una columna. Si cogemos la primera columna vamos a darle a las etiquetas nombre y observaciones y al botón un 5% a cada uno y el resto (85%) se lo daremos al cuadro de texto:

```
package net.zabalburu.pruebaswing;
```

```

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class EjemploGridBag extends JFrame {
 private GridBagLayout grid = new GridBagLayout();
 private JPanel pnl = new JPanel(grid);
 private JLabel lblNombre = new JLabel("Nombre:");
 private JTextField txtNombre = new JTextField();
 private JLabel lblObservaciones = new JLabel("Observaciones:");
 private JTextArea txaObservaciones = new JTextArea();
 private JButton btnAceptar = new JButton("Aceptar");
 private JButton btnCancelar = new JButton("Cancelar");
 private JScrollPane jspObservaciones = new JScrollPane(txaObservaciones);
 private GridBagConstraints gbc = new GridBagConstraints();
 public EjemploGridBag() {
 this.setSize(500,400);
 this.setLocation(200,200);
 this.setTitle("PruebaGridBagLayout");

 gbc.gridx = 0;
 gbc.gridy = 0;
 gbc.anchor = GridBagConstraints.WEST;
 gbc.insets = new Insets(5,5,5,5);
 gbc.weightx = 20;
 gbc.weighty = 5; // UN 5 %
 grid.setConstraints(lblNombre, gbc);
 pnl.add(lblNombre);

 gbc.gridx = 1;
 gbc.gridy = 0;
 gbc.weightx = 80;
 gbc.weighty = 0; // 0
 grid.setConstraints(txtNombre, gbc);
 pnl.add(txtNombre);

 gbc.gridx = 0;
 gbc.gridy = 1;
 gbc.gridwidth = 2;
 gbc.weightx = 0;
 gbc.weighty = 5; // UN 5 %
 grid.setConstraints(lblObservaciones, gbc);
 pnl.add(lblObservaciones);

 gbc.gridx = 0;
 gbc.gridy = 2;
 gbc.weighty = 85; // Un 85%
 grid.setConstraints(jspObservaciones, gbc);
 pnl.add(jspObservaciones);

 gbc.gridx = 0;
 gbc.gridy = 3;
 gbc.gridwidth = 1;
 gbc.ipadx = 10;
 gbc.ipady = 5;
 gbc.weighty = 5; // UN 5 %
 grid.setConstraints(btnAceptar, gbc);
 pnl.add(btnAceptar);

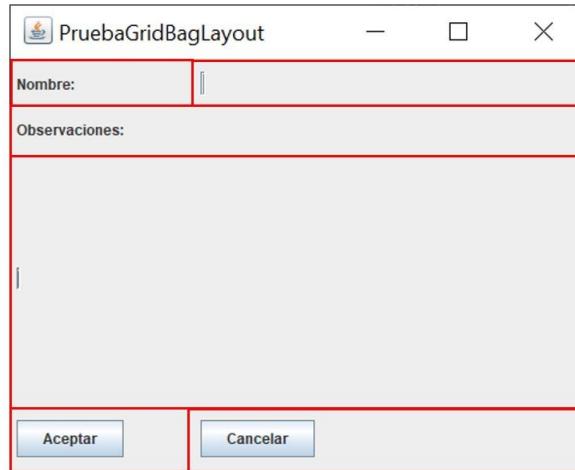
 gbc.gridx = 1;
 gbc.gridy = 3;
 gbc.weighty = 5; // Un 5%
 grid.setConstraints(btnCancelar, gbc);
 pnl.add(btnCancelar);

 this.add(pnl);
 this.setVisible(true);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
 }

 public static void main(String[] args) {
 new EjemploGridBag();
 }
}

```

La salida:



### Relleno

El último paso consiste en decidir si queremos que el componente ocupe el espacio disponible de la celda en la que está. Recordemos que el `GridLayout` tiene en cuenta el tamaño preferido así que siempre podemos poner un tamaño a los componentes.

Pero, si queremos que los componentes ocupen el espacio disponible podemos emplear la propiedad `fill` que tiene los siguientes valores:

- `GridBagConstraints.HORIZONTAL`: El componente ocupa todo el espacio disponible en horizontal
- `GridBagConstraints.VERTICAL`: El componente ocupa todo el espacio disponible en vertical
- `GridBagConstraints.BOTH`: El componente ocupa todo el espacio disponible en la celda
- `GridBagConstraints.NONE`: El componente ocupa el espacio indicado por su tamaño preferido (por defecto)

Vamos a hacer que el cuadro de texto ocupe todo el espacio en horizontal, el área de texto todo el disponible y el resto se queden como están:

```
package net.zabalburu.pruebaswing;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class EjemploGridBag extends JFrame {
 private GridBagLayout grid = new GridBagLayout();
 private JPanel pnl = new JPanel(grid);
 private JLabel lblNombre = new JLabel("Nombre:");
 private JTextField txtNombre = new JTextField();
 private JLabel lblObservaciones = new JLabel("Observaciones:");
 private JTextArea txaObservaciones = new JTextArea();
 private JButton btnAceptar = new JButton("Aceptar");
 private JButton btnCancelar = new JButton("Cancelar");
 private JScrollPane jspObservaciones = new JScrollPane(txaObservaciones);
 private GridBagConstraints gbc = new GridBagConstraints();
 public EjemploGridBag() {
 this.setSize(500,400);
 this.setLocation(200,200);
 this.setTitle("PruebaGridLayout");

 gbc.gridx = 0;
 gbc.gridy = 0;
 gbc.anchor = GridBagConstraints.WEST;
 gbc.insets = new Insets(5,5,5,5);
 gbc.weightx = 20;
 gbc.weighty = 5;
 grid.setConstraints(lblNombre, gbc);
 pnl.add(lblNombre);

 gbc.gridx = 1;
 gbc.gridy = 0;
 gbc.weightx = 80;
 gbc.weighty = 0;
 gbc.fill = GridBagConstraints.HORIZONTAL; // Todo el espacio horizontal
 grid.setConstraints(txtNombre, gbc);
 pnl.add(txtNombre);

 gbc.gridx = 0;
 gbc.gridy = 1;
 gbc.gridwidth = 2;
 gbc.weightx = 0;
 gbc.weighty = 5;
 grid.setConstraints(jspObservaciones, gbc);
 pnl.add(jspObservaciones);

 gbc.gridx = 0;
 gbc.gridy = 2;
 gbc.gridwidth = 2;
 gbc.weightx = 1;
 gbc.weighty = 1;
 grid.setConstraints(btnAceptar, gbc);
 pnl.add(btnAceptar);

 gbc.gridx = 0;
 gbc.gridy = 3;
 gbc.gridwidth = 2;
 gbc.weightx = 1;
 gbc.weighty = 1;
 grid.setConstraints(btnCancelar, gbc);
 pnl.add(btnCancelar);
 }
}
```

```

gbc.fill = GridBagConstraints.NONE; // Lo desactivamos
pnl.add(lblObservaciones);

gbc.gridx = 0;
gbc.gridy = 2;
gbc.weighty = 85;
gbc.fill = GridBagConstraints.BOTH; // Todo el espacio disponible
grid.setConstraints(jspObservaciones, gbc);
pnl.add(jspObservaciones);

gbc.gridx = 0;
gbc.gridy = 3;
gbc.gridwidth = 1;
gbc.ipadx = 10;
gbc.ipady = 5;
gbc.weighty = 5;
gbc.fill = GridBagConstraints.NONE; // Lo desactivamos
grid.setConstraints(btnAceptar, gbc);
pnl.add(btnAceptar);

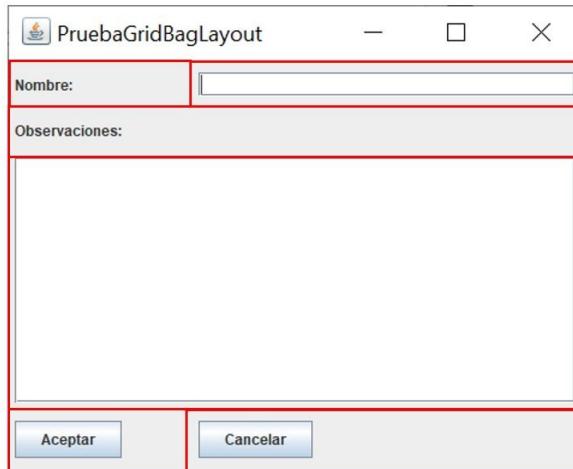
gbc.gridx = 1;
gbc.gridy = 3;
gbc.weighty = 5;
grid.setConstraints(btnCancelar, gbc);
pnl.add(btnCancelar);

this.add(pnl);
this.setVisible(true);
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

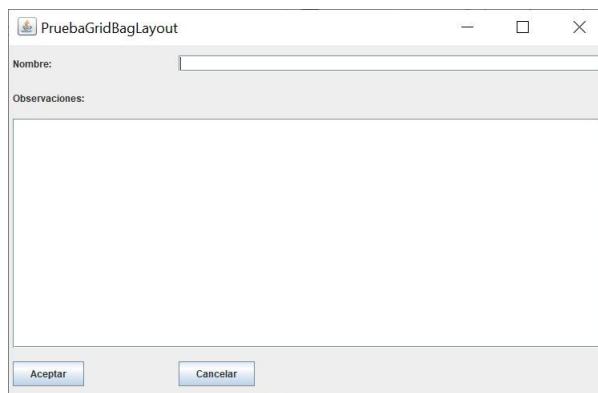
public static void main(String[] args) {
 new EjemploGridBag();
}
}

```

La salida:



El formulario ahora se redimensiona teniendo en cuenta todas las propiedades vistas:



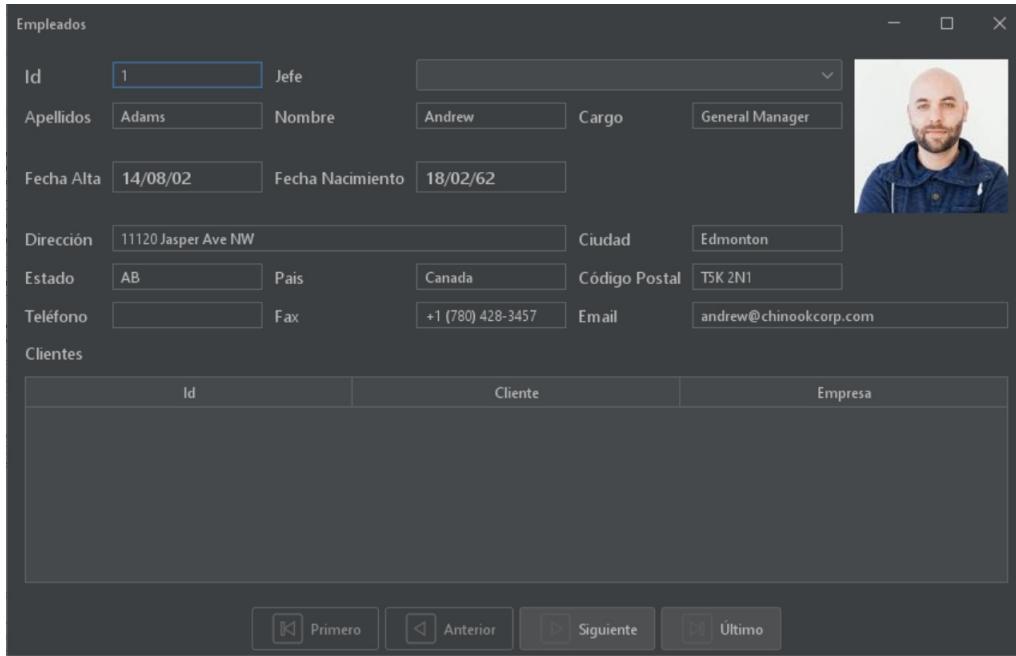
Mejoras

- Los botones se separan al redimensionar el formulario. Una solución sería meter los dos en un panel (`FlowLayout`) alineado a la izquierda y que ocupe las dos columnas.
- Se podría poner un borde a la ventana para que no esté tan pegada

## 5.17. GridBagLayout Ejemplo Complejo

### Actividad Guiada

Vamos a desarrollar, empleando `GridBagLayout`, un formulario de empleados similar al siguiente:



Para ello emplearemos este [Fichero de Empleados](#). De cada empleado se guardan los siguientes campos: id, apellidos, nombre, puesto, idJefe, fechaNacimiento, fechaAlta, dirección, ciudad, estado, país, codPostal, telefono, fax, email, urlFoto

NOTA: Si queréis probar una imagen circular, podéis usar la siguiente clase:

```
package org.zabalburu.agenda.util;

import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.RenderingHints;
import java.awt.geom.Ellipse2D;
import java.awt.image.BufferedImage;

/**
 * @author ichueca
 */
public class CircleImage {

 public static Image getCircleImage(Image img) {
 int width = img.getWidth(null);
 int height = img.getHeight(null);

 BufferedImage bi = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
 Graphics2D g2 = bi.createGraphics();

 g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
 RenderingHints.VALUE_ANTIALIAS_ON);
 g2.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY);
 int circleDiameter = Math.min(width, height);
 Ellipse2D.Double circle = new Ellipse2D.Double(0, 0, circleDiameter, circleDiameter);
 g2.setClip(circle);

 g2.drawImage(img, 0, 0, null);
 return bi;
 }
}
```

Para emplearla:

```
ImageIcon im =new ImageIcon(new URL(empleado.getFoto()));
im.setImage(CircleImage.getCircleImage(im.getImage()));
lblFoto.setIcon(im);
```

## 5.18. Actividad 29 - GridBagLayout

### Instrucciones

Se pide diseñar un formulario para los clientes de la empresa de manera similar al creado en clase. Los datos de los clientes se encuentran en este [Fichero de Clientes](#).

En dicho fichero están almacenados los datos de los clientes en el siguiente orden: id, nombre, apellidos, empresa, dirección, ciudad, estado, país, códigoPostal, teléfono, fax, email, idRepresentante, urlFoto. Donde los campos id e idRepresentante son enteros y el resto cadenas de texto. El formulario deberá emplear un diseño basado en un `GridBagLayout` y deberá redimensionarse de manera adecuada. Será necesario definir una clase `Cliente`, un interfaz `ClienteDAO`

con métodos para obtener todos los clientes, buscar un cliente en base al id y para modificar un cliente y una clase `ClienteFich` que implemente dicho interfaz. Además se añadirán una propiedad de dicha clase y los mismos métodos a la clase de servicio ya creada. Podéis, si queréis, crear una **clase de servicio**. El campo `idRepresentante` se mostrará en el formulario como un `JComboBox` (que configuraremos en una actividad posterior). El formulario mostrará todos los controles (excepto el `id`) como editables y dispondrá, aparte de los botones de desplazamiento, de botones para guardar, cancelar y salir. Esto se hará en un método `mostrar` en el que se actualizará la información del formulario en base al cliente actual (`pos`) (excepto el `JComboBox` que todavía no sabemos cómo hacerlo). El método `guardar` creará un nuevo cliente con la información del formulario (excepto el campo `idRepresentante` que se cogerá del cliente en la posición actual). Tras ello se llamará al método `modificar` del servicio y luego a `mostrar`. El método `cancelar` simplemente llamará a `mostrar` y el método `salir` finalizará la aplicación.

### 5.19. Más Gestores de Distribución CardLayout BoxLayout NullLayout

#### Card Layout

El gestor de distribución **CardLayout** permite situar los componentes como si estuvieran en una pila de cartas. De todos ellos, sólo se muestra uno cada vez y se puede cambiar de componente a través del propio layout.

Puede ser útil para llevar al usuario por una serie de pasos guiados. El funcionamiento es el siguiente:

- Creamos el `CardLayout` y lo asignamos al panel.
- Añadimos al panel los componentes que queremos (opcionalmente les damos un nombre).
- Los mostramos con los métodos del layout:
  - `previous(pnl)`: Muestra el componente anterior al actualmente mostrado.
  - `next(pnl)`: Muestra el componente siguiente al actualmente mostrado.
  - `first(pnl)`: Muestra el primer componente.
  - `last(pnl)`: Muestra el último componente.
  - `show(pnl, cmp)`: Muestra el componente con el nombre indicado.

```
package net.zabalburu.pruebaswing;

import java.awt.BorderLayout;
import java.awt.CardLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

public class EjemploCardLayout extends JFrame {
 private JPanel[] paneles = {
 new JPanel(), new JPanel(), new JPanel(), new JPanel()
 };
 private JLabel[] etiquetas = {
 new JLabel("Panel 1"), new JLabel("Panel 2"), new JLabel("Panel 3"), new JLabel("Panel 4")
 };
 private JButton btnPrimero = new JButton("Inicio");
 private JButton btnAnterior = new JButton("Anterior");
 private JButton btnSiguiente = new JButton("Siguiente");
 private JButton btnFinalizar = new JButton("Finalizar");
 private CardLayout cl = new CardLayout();
 private JPanel pnlAyuda = new JPanel(cl);
 private JPanel pnlBotones = new JPanel();
 private int pos = 1;

 public EjemploCardLayout() {
 this.setSize(500, 400);
 this.setLocation(200, 200);
 for (int i = 0; i < paneles.length; i++) {
 paneles[i].add(etiquetas[i]);
 pnlAyuda.add(paneles[i], "" + (i + 1));
 }
 cl.show(pnlAyuda, "1");
 mostrarBotones();
 });

 pnlBotones.add(btnPrimero);

 btnPrimero.addActionListener((e) -> {
 cl.first(pnlAyuda);
 pos = 1;
 mostrarBotones();
 });

 btnAnterior.addActionListener((e) -> {
 cl.previous(pnlAyuda);
 pos--;
 mostrarBotones();
 });

 btnSiguiente.addActionListener((e) -> {
 cl.next(pnlAyuda);
 pos++;
 mostrarBotones();
 });

 btnFinalizar.addActionListener((e) -> {
 JOptionPane.showMessageDialog(this, "Proceso Finalizado");
 System.exit(0);
 });
}
```

```

 });
 pnlBotones.add(btnFinalizar);

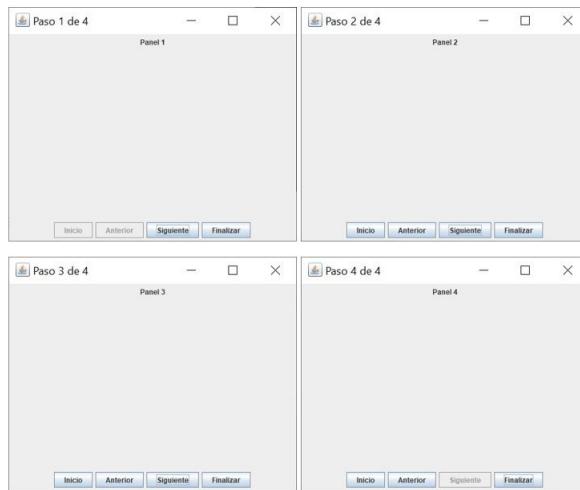
 this.add(pnlAyuda);
 this.add(pnlBotones, BorderLayout.SOUTH);
 this.setVisible(true);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

private void mostrarBotones() {
 btnPrimero.setEnabled(pos != 1);
 btnAnterior.setEnabled(pos != 1);
 btnSiguiente.setEnabled(pos != paneles.length);
 this.setTitle("Paso " + pos + " de " + paneles.length);
}

public static void main(String[] args) {
 new EjemploCardLayout();
}
}

```

**La salida:**



## BoxLayout

El **BoxLayout** es un layout que permite situar los componentes en horizontal o en vertical. En este layout, el tamaño del componente depende del tamaño preferido y máximo, y de la alineación del mismo respecto al resto de los componentes.

Además, en un **BoxLayout** se puede añadir espacio con los siguientes métodos de la clase **Box**:

- **Box.createRigidArea(Dimension)**: Crea un espacio vacío del tamaño indicado.
- **Box.filler(min, pref, max)**: Crea un espacio con las dimensiones mínimas, preferidas y máximas indicadas (el tamaño real dependerá siempre del espacio disponible y podrá ser distinto en función del mismo).
- **Box.createHorizontalGlue()**: Crea un espacio vacío elástico en el eje horizontal (en principio ocupa todo el espacio libre, pero si hay más componentes del mismo tipo, se reparten el espacio equitativamente).
- **Box.createVerticalGlue()**: Ídem en el eje vertical.

Este layout puede ser interesante para situar elementos en diferentes grupos.

**Nota:** La forma de asignar este layout es un poco especial, dado que para ser creado necesita una referencia al panel al que se va a asignar. Así que hay que dar los siguientes pasos:

1. Crear el panel.
2. Crear el layout.
3. Asignar el layout al panel (`setLayout`).

### Ejemplo en horizontal:

```

package net.zabalburu.pruebaswing;

import java.awt.BorderLayout;
import java.awt.Dimension;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;

public class EjemploBoxLayout extends JFrame {
 private JPanel pnlSur = new JPanel();
 private BoxLayout bl = new BoxLayout(pnlSur, BoxLayout.X_AXIS);
 private JButton btnPrimero = new JButton("Primero");
 private JButton btnAnterior = new JButton("Anterior");

```

```

private JButton btnSiguiente = new JButton("Siguiente");
private JButton btnUltimo = new JButton("Último");
private JButton btnNuevo = new JButton("Nuevo");
private JButton btnCancelar = new JButton("Cancelar");
private JButton btnGuardar = new JButton("Guardar");
private JButton btnSalir = new JButton("Salir");

public EjemploBoxLayout() {
 this.setSize(900, 300);
 this.setLocation(200, 200);
 this.setTitle("Prueba BoxLayout");
 pnlSur.setLayout(bl);

 pnlSur.add(Box.createRigidArea(new Dimension(10, 0)));
 pnlSur.add(btnPrimero);
 pnlSur.add(Box.createHorizontalStrut(5));
 pnlSur.add(btnAnterior);
 pnlSur.add(Box.createHorizontalStrut(5));
 pnlSur.add(btnSiguiente);
 pnlSur.add(Box.createHorizontalStrut(5));
 pnlSur.add(btnUltimo);
 pnlSur.add(Box.createHorizontalGlue());
 pnlSur.add(btnNuevo);
 pnlSur.add(Box.createHorizontalStrut(5));
 pnlSur.add(btnCancelar);
 pnlSur.add(Box.createHorizontalStrut(5));
 pnlSur.add(btnGuardar);
 pnlSur.add(Box.createHorizontalGlue());
 pnlSur.add(btnSalir);
 pnlSur.add(Box.createHorizontalStrut(10));

 pnlSur.setBorder(new EmptyBorder(10, 0, 10, 0));

 this.add(pnlSur, BorderLayout.SOUTH);
 this.setVisible(true);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public static void main(String[] args) {
 new EjemploBoxLayout();
}
}

```

#### La salida:



## NullLayout

El gestor de distribución **nulo** consiste en no usar ningún gestor de distribución. En este caso, hay que especificar la ubicación de los componentes (`setLocation`) y su tamaño (`setSize`).

Para indicar que no queremos emplear un layout, hay que pasar el valor `null` al panel.

Evidentemente, redimensionar la ventana no modifica ni el tamaño ni la posición de los componentes, por lo que puede ser una buena idea evitar que se pueda redimensionar la ventana (`setResizable(false)`).

**Nota:** Principalmente por este motivo, no es muy recomendable emplear este tipo de layouts.

### 5.20. Listas Desplegables JComboBox

#### Listas Desplegables

Las listas desplegables en Swing se gestionan mediante el componente `JComboBox`.

#### class JComboBox (javax.swing.JComboBox)

Un cuadro de lista desplegable está compuesto por dos elementos:

- Un menú emergente (una implementación de la clase `swing.plaf.basic.ComboPopup`) que, por defecto, es una subclase de `JPopupMenu` que contiene una lista (`JList`) en un panel desplazable (`JScrollPane`).
- Un botón actuando como contenedor de un componente de edición o visualización y un botón de flecha que se emplea para mostrar el menú desplegable.

Para seleccionar los elementos de la lista desplegable se emplea como modelo de datos un `ListSelectionModel` que solo permite seleccionar un elemento cada vez (`SINGLE_SELECTION`). Adicionalmente, se emplea un modelo `JComboBoxModel` para gestionar los datos del `JList`.

#### Crear un JComboBox

- **JComboBox vacío:**

```
JComboBox<Empleado> cbx = new JComboBox<>(); // Añadiremos los empleados manualmente
```

- **A partir de una matriz con datos:**

```
Empleado[] emp = { new Empleado(...), ... };
JComboBox<Empleado> cbx = new JComboBox(emp); // Con los empleados ya definidos
```

- **Si tenemos una lista:**

```
List<Empleado> empleados = servicio.getEmpleados();
JComboBox<Empleado> cbx = new JComboBox(empleados.toArray());
```

---

### Añadir y eliminar elementos a/de un JComboBox

- **Añadir elementos:**

```
cbx.addItem(new Empleado(...)); // Añade el empleado al final
cbx.insertItemAt(new Empleado(...), pos); // Inserta el empleado en la posición indicada
```

- **Eliminar elementos:**

```
cbx.removeItem(empleado); // Emplea el método equals de la clase Empleado para eliminar el empleado indicado
cbx.removeItemAt(pos); // Elimina el empleado que está en la posición indicada
cbx.removeAllItems(); // Elimina todos los elementos
```

---

### Recuperar el elemento seleccionado

- **Obtener el elemento seleccionado:**

```
Empleado emp = (Empleado) cbx.getSelectedItem(); // El empleado (null si no hay nada seleccionado)
int pos = cbx.getSelectedIndex(); // La posición del elemento seleccionado (-1 si no hay nada seleccionado)
```

---

### Seleccionar elementos por código

- **Seleccionar un elemento:**

```
cbx.setSelectedItem(emp); // Emplea el método equals de la clase Empleado para decidir cuál hay que seleccionar (null deja el JComboBox en blanco)
cbx.setSelectedIndex(pos); // Selecciona el elemento indicado (si es -1 se deja el JComboBox en blanco)
```

**Nota:** A la hora de mostrar cada elemento de la lista desplegable, se emplea por defecto una subclase de `JLabel` que implementa el interfaz `ListCellRenderer`. Para la edición, se emplea por defecto una instancia de la clase `ComboBoxEditor` que utiliza un `JTextField`. Es posible utilizar nuestros propios visualizadores y editores (se verá un ejemplo posteriormente).

---

### Eventos

Las listas desplegables generan los eventos AWT tradicionales:

- **ActionEvent:** Se lanza cuando se hace doble clic sobre un elemento o se selecciona.
- **ItemEvent:** Se lanza cuando se selecciona/deselecciona un elemento.

**Nota:** El evento `ActionEvent` también se lanza en un `JComboBox` editable cuando se modifica el cuadro de texto y se pulsa `Enter`.

- **Ejemplo con ActionEvent:**

```
cbxEmppleado.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent evt) {
 seleccionEmpleado();
 }
});
...
private void procesarEmpleado() {
 Empleado emp = (Empleado) cbxEmppleado.getSelectedItem();
 ...
}
```

---

### Recorrer un JComboBox

Para recorrer los elementos de un cuadro de lista desplegable:

```
for (int i = 0; i < cbx.getItemCount(); i++) {
 Empleado emp = (Empleado) cbx.getItemAt(i);
 ...
}
```

---

### Otras propiedades

Un elemento de lista desplegable puede ser:

- **Editable:** El usuario puede escribir en el cuadro de texto asociado.
- **No editable:** Solo se pueden seleccionar elementos de la lista.

Por defecto, el cuadro combinado no es editable. Para cambiar este comportamiento:

```
cbx.setEditable(true); // Hacer editable
cbx.setEditable(false); // Hacer no editable
```

**Nota:** Se añaden y recuperan objetos al `JComboBox`, pero lo que se muestra es una representación del mismo en una etiqueta (por defecto, el resultado de llamar al método `toString()` del objeto). Este comportamiento se puede modificar.

---

### Ejemplo

```
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;

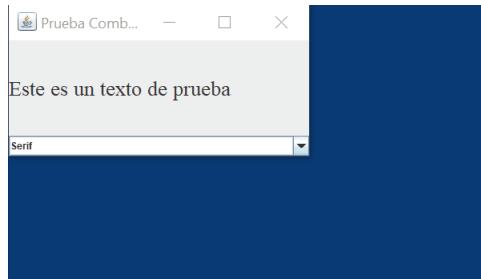
public class PruebaComboBox extends JFrame {
 JLabel et;
 JComboBox<String> cbxFuentes;

 public PruebaComboBox() {
 this.setTitle("Prueba Combo box");
 this.setSize(350, 200);
 this.setLocation(200, 200);
 JPanel con = (JPanel) this.getContentPane();
 et = new JLabel("Este es un texto de prueba");
 cbxFuentes = new JComboBox<String>();
 cbxFuentes.setEditable(true);
 cbxFuentes.addItem("Serif");
 cbxFuentes.addItem("SansSerif");
 cbxFuentes.addItem("Monospaced");
 cbxFuentes.addItem("Dialog");
 cbxFuentes.addItem("DialogInput");

 ActionListener al = new ActionListener() {
 public void actionPerformed(ActionEvent a) {
 String fuente = (String) cbxFuentes.getSelectedItem();
 Font f = new Font(fuente, Font.PLAIN, 26);
 et.setFont(f);
 int i;
 for (i = 0; i < cbxFuentes.getItemCount() && !cbxFuentes.getItemAt(i).equalsIgnoreCase(f.getFontName()); i++);
 if (i == cbxFuentes.getItemCount()) {
 cbxFuentes.addItem(f.getFontName());
 }
 }
 };
 cbxFuentes.addActionListener(al);
 cbxFuentes.setSelectedIndex(0);
 con.add(et, BorderLayout.CENTER);
 con.add(cbxFuentes, BorderLayout.SOUTH);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
 this.setVisible(true);
 }

 public static void main(String[] args) {
 PruebaComboBox pcb = new PruebaComboBox();
 }
}
```

### La salida:



---

### Cambiar el aspecto de la lista

Para cambiar el aspecto de un `JComboBox`, necesitamos una clase que implemente `ListCellRenderer`. Esto nos obliga a definir el método:

```
public Component getListCellRendererComponent(JList lista, Object elemento, int posicion, boolean seleccionado, boolean enfocado)
```

Este método debe retornar un componente (habitualmente un `JLabel`) que se mostrará para cada opción de la lista.

---

### Ejemplo con ListCellRenderer

```
import java.awt.Component;
import java.awt.Image;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.*;

public class PruebaComboBoxRender extends JFrame {
 private Pais[] paises = {
 new Pais("Austria", "Vienna", "https://www.flagsimporter.com/pub/media/catalog/category/AUSTRIA35-2.jpg"),
 new Pais("Belgium", "Brussels", "https://www.flagsimporter.com/pub/media/catalog/category/BELGIUM35-2.jpg"),
 ...
 }
```

```

 new Pais("Spain", "Madrid", "https://www.flagsimporter.com/pub/media/catalog/category/SPAIN35-2.jpg"),
 new Pais("Sweden", "Stockholm", "https://www.flagsimporter.com/pub/media/catalog/category/SWEDEN35-2.jpg"),
 new Pais("United Kingdom", "London", "https://www.flagsimporter.com/pub/media/catalog/category/UK35-2.jpg")
);
}

private JComboBox<Pais> cbxPaises = new JComboBox<>(paises);

public PruebaComboBoxRender() {
 this.setSize(400, 120);
 this.setLocation(400, 400);
 this.setTitle("Países");
 cbxPaises.setRenderer(new PaisRenderer());
 this.add(cbxPaises);
 this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 this.setVisible(true);
}

class PaisRenderer implements ListCellRenderer<Pais> {
 @Override
 public Component getListCellRendererComponent(JList<? extends Pais> list, Pais value, int index, boolean isSelected, boolean cellHasFocus) {
 JLabel lbl = new JLabel();
 lbl.setText("<html><body><h2>" + value.getPais() + "</h2><i>" + value.getCapital() + "</i></body></html>");
 lbl.setIcon(value.getBandera());
 lbl.setOpaque(true);
 if (isSelected) {
 lbl.setBackground(list.getSelectionBackground());
 } else {
 lbl.setBackground(list.getBackground());
 }
 return lbl;
 }
}

public static void main(String[] args) throws IOException {
 new PruebaComboBoxRender();
}
}

class Pais {
 private String pais;
 private String capital;
 private ImageIcon imIC;

 public Pais(String name, String capital, String bandera) {
 this.pais = name;
 this.capital = capital;
 try {
 imIC = new ImageIcon(new URL(bandera));
 Image imag = imIC.getImage().getScaledInstance(100, 40, Image.SCALE_SMOOTH);
 imIC.setImage(imag);
 } catch (MalformedURLException ex) {
 Logger.getLogger(Pais.class.getName()).log(Level.SEVERE, null, ex);
 }
 }

 public String getPais() {
 return pais;
 }

 public String getCapital() {
 return capital;
 }

 public ImageIcon getBandera() {
 return imIC;
 }
}
}

```

#### La salida:



#### 5.21. Listas JList

##### Listas

Las listas son componentes que permiten seleccionar uno o varios elementos entre un grupo de valores. La clase que implementa las listas en Swing es la clase `JList`.

## Modelos de JList

La clase `JList` dispone de dos modelos:

- `ListModel`: Gestiona los datos de la lista.
- `ListSelectionModel`: Gestiona la selección de los elementos.

La clase `JList`, al igual que `JComboBox`, también permite emplear visualizadores de datos personalizados a través del interfaz `ListCellRenderer` o de la clase por defecto `DefaultListCellRenderer` (que es la que se emplea si no se especifica el visualizador a emplear).

El visualizador por defecto funciona de manera similar al caso anterior, empleando el método `toString()` sobre los objetos de la lista para mostrarlos (excepto en el caso de iconos, en el que se emplea el ícono).

### Crear una lista

A la hora de construir una lista, se puede hacer sin datos, pasándole una matriz de objetos o un vector, o empleando un modelo de lista que determina cómo se gestionan dichos datos. También podemos usar los métodos `setListData` para asociar datos a la lista.

```
JList<Empleado> lst = new JList<>(); // Habrá que añadir los empleados a mano (mediante el modelo)
```

```
Empleado[] empleados = { new Empleado(...), ... };
JList<Empleado> lst = new JList(empleados);
```

La clase `JList` implementa el interfaz `Scrollable` para permitir desplazamientos unitarios verticales en función de la altura de la celda y de bloque en función de las filas visibles. En cualquier caso, para permitir emplear el desplazamiento, el objeto debe situarse en un `JScrollPane`.

### Añadir / Eliminar elementos

A diferencia del `JComboBox`, no podemos añadir elementos directamente a la lista. Necesitamos hacerlo a través del modelo. Así que los pasos son:

1. Crear un modelo de datos que implemente `ListModel` o extienda la clase `DefaultListModel`.
2. Añadir datos al modelo.
3. Asignar el modelo a la lista.

```
DefaultListModel<Empleado> modelo = new DefaultListModel<>();
modelo.addElement(new Empleado(...));
...
lista.setModel(modelo);
```

Si conocemos todos los elementos que queremos añadir a la lista, podemos emplear `setListData(Object[])` o `setListData(Vector)`:

```
Empleado[] empleados = { new Empleado(...), ... };
lst.setListData(empleados);
```

### Definir qué elementos se pueden seleccionar

La clase `JList` dispone de un modelo de selección que determina cómo se pueden seleccionar elementos (`setSelectionMode`). Las opciones son:

Modo de Selección	Descripción	Ejemplo
<code>ListSelectionModel.SINGLE_SELECTION</code>	Solo se puede seleccionar un elemento de la lista	
<code>ListSelectionModel.SINGLE_INTERVAL_SELECTION</code>	Un rango de elementos consecutivos	
<code>ListSelectionModel.MULTIPLE_INTERVAL_SELECTION</code>	Diferentes rangos de elementos no necesariamente seguidos (por defecto)	

Por ejemplo, para permitir seleccionar únicamente un empleado:

```
lst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

#### Recuperar los elementos seleccionados

Opciones:

```
Empleado emp = lst.getSelectedValue(); // Retorna el PRIMER empleado seleccionado
int pos = lst.getSelectedIndex(); // Retorna la posición del primer empleado seleccionado

List<Empleado> empleados = lst.getSelectedValuesList(); // Todos los empleados seleccionados
int[] pos = lst.getSelectedIndices(); // Las posiciones de todos los empleados seleccionados
```

#### Seleccionar elementos por código

Opciones:

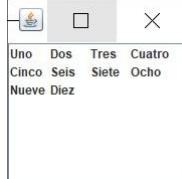
```
lst.setSelectedValue(emp, true); // Selecciona el empleado indicado (en base a equals) y, si no estuviera visible, hace que lo esté (true)
lst.setSelectedIndex(pos); // Selecciona el empleado en la posición pos
lst.setSelectionIndices(new int[]{1, 2, 4}); // Selecciona los elementos 2, 3 y 5 de la lista
lst.setSelectionInterval(1, 3); // Selecciona los empleados 2, 3 y 4
```

#### Opciones de Visibilidad

Para saber o especificar cuántas filas son visibles, disponemos de los métodos `getVisibleRowCount`. El método `ensureIndexIsVisible` se asegura de que la fila indicada sea visible.

Por defecto, el ancho de la lista es el ancho del elemento más ancho y el alto de la celda corresponde con el del elemento con mayor altura. Es posible modificar estos valores con los métodos `setFixedCellWidth` y `setFixedCellHeight`.

La forma en la que se muestran los elementos se puede especificar mediante el método `setLayoutOrientation`, que admite los siguientes valores:

Orientación	Descripción	Ejemplo
JList.HORIZONTAL_WRAP	Sitúa los elementos por filas. Se crean tantas filas como indique <code>setVisibleRowCount</code> (en el ejemplo 3).	
JList.VERTICAL_WRAP	Similar, pero se sitúan los elementos por columnas.	
JList.VERTICAL	Se sitúan tantos elementos en vertical (por defecto).	

#### Ejemplo:

```
String[] datos = {"Uno", "Dos", "Tres", "Cuatro", "Cinco", "Seis", "Siete", "Ocho", "Nueve", "Diez";
JList lista = new JList(datos);
lista.setVisibleRowCount(3);
lista.setLayoutOrientation(JList.HORIZONTAL_WRAP);
```

#### Renderizado

Al igual que en el `JComboBox`, podemos modificar el renderizador por defecto (que muestra en un `JLabel` el resultado de llamar a `toString()` para cada elemento) por un objeto que implemente `ListCellRenderer` (o extienda `DefaultListCellRenderer`). Especifiquemos el renderizador a emplear con `setCellRenderer`.

#### Eventos

El evento que permite comprobar si se ha seleccionado un elemento es `ListSelectionEvent`, cuya interfaz (`ListSelectionListener`) solo tiene un método `valueChanged` que

se llama cada vez que cambia la selección.

**Nota:** Cuando se selecciona un elemento, este evento se lanza dos veces: una para indicar el estado antes de la selección y otra para indicar el estado final.

#### Ejemplo:

```
lst.addListSelectionListener(new ListSelectionListener() {
 public void valueChanged(ListSelectionEvent evt) {
 if (!evt.getValueIsAdjusting()) {
 seleccionEmpleado(evt);
 }
 }
});

private void seleccionEmpleado(ListSelectionEvent evt) {
 Empleado emp = (Empleado) lst.getSelectedValue();
 ...
}
```

---

#### Ejemplo

En el siguiente ejemplo, se muestran las fuentes disponibles del sistema y, al seleccionar una de ellas, se muestra el texto en dicha fuente. Adicionalmente, se modifica el renderizador por defecto para que las fuentes de la lista se muestren en su propio estilo:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

public class PruebaList extends JFrame {
 JList listaFuentes;
 DefaultListModel modelo;
 MiVisualizador vista;
 JLabel texto = new JLabel("Esto es una prueba de texto");

 public PruebaList() {
 this.setTitle("Prueba JList");
 this.setSize(300, 250);
 this.setLocation(300, 200);
 listaFuentes = new JList();
 GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
 Font[] fuentes = ge.getAllFonts();
 for (int i = 0; i < fuentes.length; i++) {
 fuentes[i] = fuentes[i].deriveFont(12F);
 }
 listaFuentes.setListData(fuentes);
 vista = new MiVisualizador();
 listaFuentes.setCellRenderer(vista);
 listaFuentes.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
 JScrollPane j = new JScrollPane(listaFuentes);
 this.add(j, BorderLayout.SOUTH);
 this.add(texto, BorderLayout.CENTER);

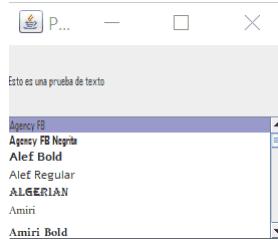
 listaFuentes.addListSelectionListener(new ListSelectionListener() {
 public void valueChanged(ListSelectionEvent lse) {
 if (!lse.getValueIsAdjusting()) {
 Font f = (Font) listaFuentes.getSelectedValue();
 texto.setFont(f);
 }
 }
 });
 }

 listaFuentes.setSelectedIndex(0);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
 this.setVisible(true);
}

public static void main(String[] args) {
 PruebaList pl = new PruebaList();
}

class MiVisualizador extends JLabel implements ListCellRenderer {
 public Component getListCellRendererComponent(JList list, Object value, int index, boolean isSelected, boolean cellHasFocus) {
 Font f = (Font) value;
 this.setFont(f);
 this.setText(f.getFontName());
 if (isSelected) {
 this.setOpaque(true);
 this.setBackground(new Color(153, 153, 204));
 } else {
 this.setOpaque(false);
 }
 return this;
 }
}
```

#### La salida:



## 5.22. Deslizadores JSlider

### JSlider

Un objeto `JSlider` permite al usuario seleccionar un valor entre un rango de valores de una manera visual.

#### Constructor

Podemos crear un objeto `JSlider` con alguno de los siguientes constructores:

```
sld = new JSlider(); // Crea un JSlider horizontal con valores de 0 a 100 y valor inicial 50
sld = new JSlider(JSlider.VERTICAL); // Similar pero en vertical (también hay JSlider.HORIZONTAL)
sld = new JSlider(1, 10); // Crea un JSlider horizontal con valores de 1 a 10 y valor inicial 5 (media)
sld = new JSlider(1, 10, 2); // Crea un JSlider horizontal con valores de 1 a 10 y valor inicial 2
sld = new JSlider(JSlider.VERTICAL, 1, 10, 2); // Idem en vertical
```

#### Asignar / recuperar valores

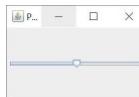
Para asignar un valor al `JSlider` empleamos:

```
sld.setValue(20); // Sitúa el slider en el valor 20
int valor = sld.getValue(); // Almacena 20 en valor
```

#### Aspecto

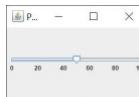
Por defecto, un `JSlider` (constructor por defecto) tiene la siguiente apariencia:

```
sld = new JSlider();
```

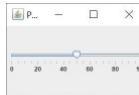


Podemos modificar su aspecto:

```
sld.setPaintLabels(true);
sld.setMinorTickSpacing(5);
sld.setMajorTickSpacing(20);
```

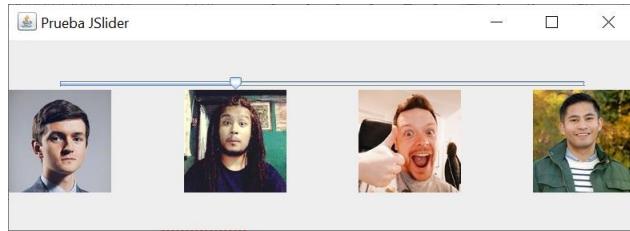


```
sld.setPaintTicks(true);
```



Por defecto, las marcas mayores se muestran mediante objetos `JLabel` que contienen el valor a mostrar, aunque se puede emplear cualquier componente creando previamente un objeto `Dictionary` (dado que es una clase abstracta, emplearemos su hija `Hashtable`, que es similar a un `HashMap`) de elementos `Integer/Component` y pasándolo al método `setLabelTable`.

```
Hashtable<Integer, JLabel> datos = new Hashtable<>();
datos.put(1, new JLabel(new ImageIcon(new URL("https://randomuser.me/api/portraits/men/1.jpg"))));
datos.put(2, new JLabel(new ImageIcon(new URL("https://randomuser.me/api/portraits/men/2.jpg"))));
datos.put(3, new JLabel(new ImageIcon(new URL("https://randomuser.me/api/portraits/men/3.jpg"))));
datos.put(4, new JLabel(new ImageIcon(new URL("https://randomuser.me/api/portraits/men/4.jpg"))));
sld = new JSlider(1, 4);
sld.setLabelTable(datos);
sld.setPaintLabels(true);
sld.setMajorTickSpacing(1);
```



Podemos forzar a que se escogen valores ajustados a las marcas con:

```
sld.setSnapToTicks(true);
```

#### Eventos

Si queremos controlar cuándo el usuario cambia la selección en un `JSlider`, debemos emplear el evento `ChangeEvent`. El interfaz `ChangeListener` solo tiene un método `stateChanged`.

#### Ejemplo

En este ejemplo, vamos a personalizar un deslizador para que muestre una valoración en modo visual representado por unas estrellas llenas, medio llenas y/o vacías. El tope es 5 estrellas:

```
import java.awt.BorderLayout;
import java.util.Hashtable;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import static javax.swing.JFrame.EXIT_ON_CLOSE;
import javax.swing.JLabel;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

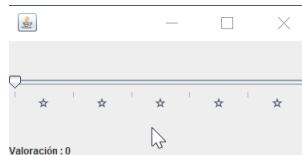
public class PruebaJSilder extends JFrame {
 private JSlider sldEstrellas = new JSlider();
 private Hashtable etiquetas;
 private Icon llena;
 private Icon medio;
 private Icon vacia;
 private JLabel lblValoracion = new JLabel("Valoración : 0");

 public PruebaJSilder() {
 vacia = new ImageIcon(getClass().getResource("/resources/star_14px.png"));
 medio = new ImageIcon(getClass().getResource("/resources/star_half_empty_14px.png"));
 llena = new ImageIcon(getClass().getResource("/resources/star_fill_14px.png"));
 this.setSize(400, 200);
 this.setLocation(200, 200);
 sldEstrellas.setMaximum(10);
 sldEstrellas.setPaintLabels(true);
 sldEstrellas.setMajorTickSpacing(2);
 sldEstrellas.setPaintTicks(true);
 sldEstrellas.setValue(0);
 etiquetas = new Hashtable();

 for (int i = 0; i <= 5; i++) {
 etiquetas.put(new Integer(i * 2 + 1), new JLabel(vacia));
 }
 sldEstrellas.setLabelTable(etiquetas);
 sldEstrellas.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent c) {
 int val = sldEstrellas.getValue();
 lblValoracion.setText("Valoración : " + val);
 int llenas = val / 2;
 int i;
 for (i = 0; i < llenas; i++)
 ((JLabel) etiquetas.get(new Integer(i * 2 + 1))).setIcon(llena);
 if (val % 2 != 0) {
 ((JLabel) etiquetas.get(new Integer(i * 2 + 1))).setIcon(medio);
 i++;
 }
 for (; i <= 5; i++)
 ((JLabel) etiquetas.get(new Integer(i * 2 + 1))).setIcon(vacia);
 sldEstrellas.repaint(); // Necesario para que se redibuje según cambia
 }
 });
 this.add(sldEstrellas);
 this.add(lblValoracion, BorderLayout.SOUTH);
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setVisible(true);
 }

 public static void main(String[] args) {
 PruebaJSilder p = new PruebaJSilder();
 }
}
```

## La salida



### 5.23. Actividad 30 - Uso de JComboBox JList y JSlider

#### Instrucciones

##### Classroom

Se desean gestionar las visitas que realizan los comerciales a los diferentes clientes de la empresa.

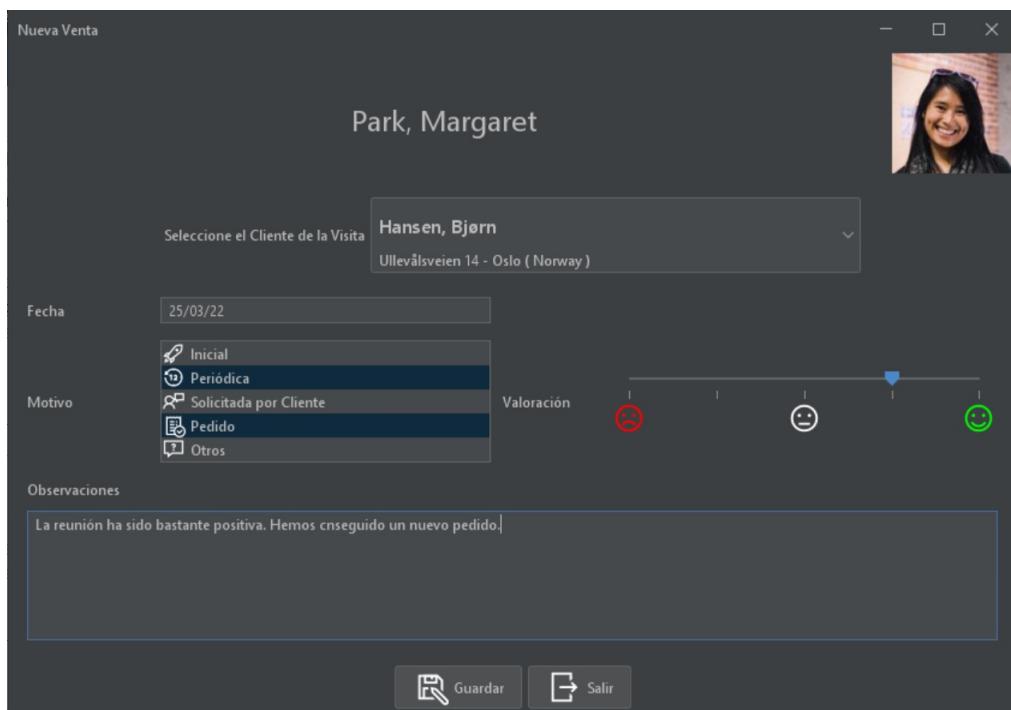
Para ello se debe definir una clase `Visita` con los campos `id (Integer)`, `idRepresentante (Integer)`, `idCliente (Integer)`, `fecha (Date)`, `motivos (List<String>)` y `valoracion (Integer)`. Además se deberá definir un DAO con métodos para recuperar todas las visitas de un empleado (`getVisitas (Integer idRepresentante)`) y otro para añadir una visita (`nuevaVisita (Visita nueva)`). Este último método deberá asignar un `id` correlativo a cada visita. La información de la visita se almacenará en un `ArrayList`.

Además, se añadirá al servicio un **método que retorne todos los clientes de un empleado** (`getClientesEmpleado (Integer idRepresentante)`) **ordenados alfabéticamente** (evidentemente este método retornará todos los empleados cuyo `idRepresentante` coincide con el recibido como parámetro).

La aplicación, al iniciarse (y antes de mostrar la ventana) pedirá el `id` del empleado que va a ejecutar la aplicación. Si el empleado **NO existe** (método `getEmpleado (id)` del servicio) se finalizará la aplicación con un mensaje adecuado. Si **existe** se obtendrán sus clientes (`getClientesEmpleado`) y, si **NO TIENE CLIENTES**, se dará un mensaje de **error** adecuado y se finalizará la aplicación. Si el empleado cumple ambas condiciones se almacenará en una propiedad de la clase y se ejecutará el código para mostrar la ventana.

NOTA: Sólo los empleados con `id 3, 4 y 5` tienen clientes.

La aplicación tendrá un aspecto similar al siguiente:



- El control `Fecha` es un `JFormattedTextField` y se inicializará con la fecha de **hoy**. El control `valoración`, por su parte tendrá asignado inicialmente un valor 3 (las valoraciones pueden ir de 1 a 5).
- El botón `guardar` añadirá la visita a la lista mediante el servicio, mostrará un mensaje indicando que la visita se ha guardado y se pondrán los controles del formulario en su valor inicial (como `fecha` la de hoy, como `valoración 3`, **seleccionado el primer cliente** de la lista, **ningún motivo seleccionado** y las `observaciones` en blanco).
- El botón `Salir` mostrará las visitas del empleado (de momento en un `JOptionPane`) como se ve en la siguiente imagen:

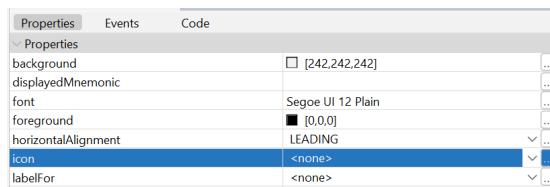
Visitas de Park, Margaret					
#	Cliente	Fecha	Valoración	Motivos	Observaciones
1	Hansen, Bjørn	25-mar-2022	3	Inicial Solicitud por Cliente	Visita de cortesía. De momento habrá que esperar.
2	Mitchell, Aaron	25-mar-2022	5	Periódica Pedido	Visita muy interesante. Hemos conseguido un nuevo pedido y parece que habrá más.

**Aceptar**

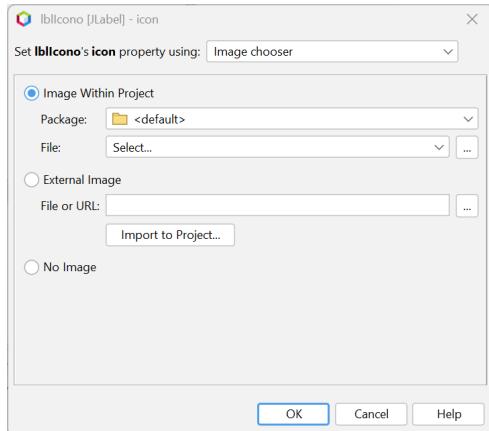
- Tras ello finalizará la aplicación.

### Gestión de recursos (imágenes, ficheros...)

Se pueden añadir imágenes directamente desde el diseñador de NetBeans. En este caso, las imágenes pueden estar fuera o dentro del proyecto. Por ejemplo, si seleccionamos una etiqueta y pulsamos sobre los tres puntos de la propiedad icon:



La ventana:

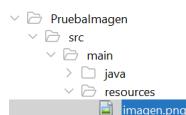


Las opciones son:

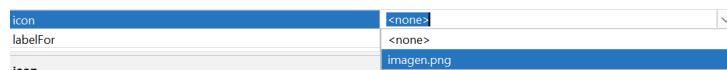
- **Image within project:** Cuando tenemos las imágenes en una carpeta (paquete) resources dentro de la carpeta `src/main` del proyecto. Desde aquí podemos seleccionarla y emplear una ruta relativa a la clase actual.
- **External image:** Las imágenes están fuera de la carpeta del proyecto. Dos opciones:
  - Si simplemente seleccionamos la imagen, se añadirá una **ruta absoluta** a la misma. Si **movemos la aplicación a otro equipo**, tendremos que **copiar las imágenes a la misma carpeta** (toda la ruta).
  - Si pulsamos sobre **Import to Project**, podemos **copiar las imágenes a una carpeta (paquete) del proyecto** y se creará una **ruta relativa** a la clase actual como en el primer caso.
- **No image:** Permite eliminar la imagen actual

Es importante tener claro que **las imágenes que van dentro del proyecto se incluyen en el archivo jax** cuando se construye el proyecto por lo que, **una vez creado el mismo, no se pueden modificar, eliminar o añadir** nuevas imágenes.

Si en NetBeans tenemos una imagen `imagen.png` que está en la carpeta :



Podemos seleccionarla en la propiedad `icon`:

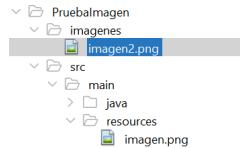


Y, en el código:

```
lblIcono.setIcon(new javax.swing.ImageIcon(getClass().getResource("/imagen.png")));
```

Si las imágenes están fuera del proyecto, es necesario incluirlas aparte del archivo jar y en la misma ruta en la que se tienen en la máquina de desarrollo. Podemos especificar rutas relativas (no desde el editor de Swing) manualmente respecto a dicho jar. Si las metemos en una carpeta, ésta debería estar dentro de la carpeta del proyecto mientras desarrollamos y debería copiarse en la carpeta donde se copie el archivo jar en el equipo en el que se ejecute.

Por ejemplo, una imagen en una carpeta imágenes en el raíz del proyecto:



Si la añadimos desde el editor, sin añadirla al proyecto nos quedaría en el código:

```
lblIcono.setIcon(new javax.swing.ImageIcon("C:\\\\Users\\\\usuario\\\\Documentos\\\\NetBeansProjects\\\\PruebaImagen\\\\imagenes\\\\imagen2.png"));
```

Con lo que, en el equipo de destino de la aplicación deberíamos copiarlo en la misma ruta.

Si queremos emplear rutas relativas, tenemos que hacerlo a mano.

```
public PruebaImagen() {
 initComponents();
 lblIcono.setIcon(new ImageIcon("imagenes/imagen2.png"));
 ...
}
```

En este caso, una vez obtenido el jar habrá que copiar la carpeta imágenes en la carpeta donde esté dicho jar.

**NOTA IMPORTANTE:** Si indicamos a NetBeans que queremos las imágenes dentro del proyecto es necesario hacer un Build para que las imágenes se copien y puedan ser accedidas desde la aplicación en desarrollo. De no hacerlo así, aparecerá un mensaje de error:

```
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException: Cannot invoke "java.net.URL.toExternalForm()" because "location" is null
```

Lo mismo se aplica a otros archivos de recursos: ficheros de datos, sonidos, etc. Pueden ubicarse fuera del jar en una ruta absoluta o relativa al proyecto o dentro del mismo en una ruta relativa a la clase.

## 5.24. Paneles JTabbedPane JSplitPane

### Paneles

Los paneles nos proporcionan una forma sencilla de situar varios componentes en un mismo panel.

#### JTabbedPane

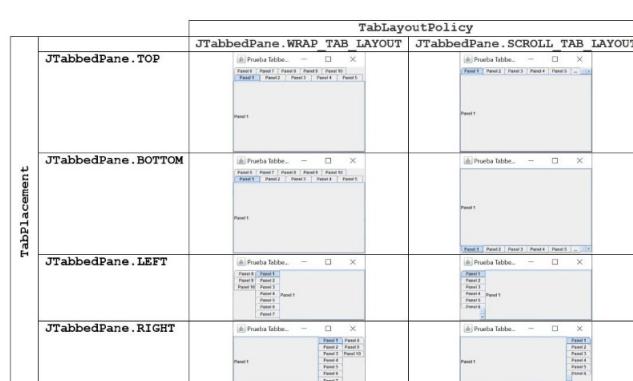
Un panel de este tipo permite ubicar los componentes y acceder a ellos mediante pestañas.

#### Constructores

```
JTabbedPane tpb = new JTabbedPane(); // Un panel por defecto con las etiquetas puestas en la parte superior
JTabbedPane tpb = new JTabbedPane(ubicación); // Ubicación puede ser JTabbedPane.TOP, JTabbedPane.BOTTOM, JTabbedPane.LEFT, o JTabbedPane.RIGHT
JTabbedPane tpb = new JTabbedPane(ubicación, política); // Política puede ser JTabbedPane.WRAP_TAB_LAYOUT o JTabbedPane.SCROLL_TAB_LAYOUT
```

#### Ubicación y Política

Podemos ver las distintas posibilidades de ubicación (setPlacement) y política (setTabLayoutPolicy) en la siguiente imagen:



#### Añadir Componentes

```
tbp.addTab(componente); // Añade el componente y pone como etiqueta el resultado del método getName() del componente
tbp.addTab("Panel1", componente); // Añade un componente con la etiqueta "Panel1"
tbp.addTab("Panel1", new ImageIcon(...), componente); // Añade el componente con una etiqueta y una imagen en la pestaña
tbp.addTab("Panel1", new ImageIcon(...), componente, pista); // Idem pero añade una pista cuando se deja el cursor en la pestaña
tbp.insertTab("Panel1", new ImageIcon(...), componente, pista, pos); // Inserta el componente en la posición pos del panel
```

#### Eliminar Componentes

```
tbp.remove(pos); // Elimina el componente en pos
```

```
tbp.removeTabAt(pos); // Ídem
tbp.removeAll(); // Elimina todos los componentes
```

#### Modificar Componentes

```
tbp.setComponentAt(pos, componente); // Sustituye el componente que había en la posición pos por el proporcionado
```

#### Modificar Parámetros de las Pestañas

Método	Descripción
tbp.setBackgroundAt(pos, color)	Pone como color de fondo el indicado.
tbp.setForegroundAt(pos, color)	Pone como color de texto el indicado.
tbp.setEnabledAt(pos, booleano)	Activa/desactiva el componente.
tbp.setIconAt(pos, icono)	Cambia el ícono de la pestaña.
tbp.setMnemonicAt(pos, carácter)	Permite acceder al componente con ALT+carácter.
tbp.setTitleAt(pos, título)	Pone el texto indicado en la pestaña.
tbp.setToolTipTextAt(pos, pista)	Pone la pista en la pestaña pos.

#### Seleccionar Componentes

```
tbp.setSelectedComponent(componente); // Selecciona (activa) la pestaña en la que está el componente (emplea equals)
tbp.setSelectedIndex(pos); // Activa el componente que está en la posición pos
```

#### Eventos

Para controlar cuando el usuario selecciona un nuevo componente, usamos el evento `ChangeEvent` y su receptor `ChangeListener`:

```
tbp.addChangeListener(new ChangeListener() {
 @Override
 public void stateChanged(ChangeEvent e) {
 System.out.println("Seleccionado " + tbp.getSelectedIndex());
 }
});
```

#### Ejemplo

```
import java.awt.GridLayout;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class EjemploTabbedPane extends JFrame {
 private JTabbedPane tbp = new JTabbedPane();
 private JPanel pnlUsuario = new JPanel(new GridLayout(1, 1));
 private JPanel pnlPago = new JPanel(new GridLayout(1, 1));
 private JPanel pnlContacto = new JPanel(new GridLayout(1, 1));
 private JPanel pnlCompra = new JPanel(new GridLayout(1, 1));

 public EjemploTabbedPane() {
 this.setSize(400, 300);
 this.setLocation(200, 200);
 this.setTitle("Usuario");
 tbp.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
 tbp.setTabPlacement(JTabbedPane.TOP);

 pnlUsuario.add(new JLabel("Usuario"));
 pnlUsuario.setName("Usuario");
 pnlPago.add(new JLabel("Pago"));
 pnlPago.setName("Pago");
 pnlContacto.add(new JLabel("Contacto"));
 pnlContacto.setName("Contacto");
 pnlCompra.add(new JLabel("Compra"));
 pnlCompra.setName("Compra");

 tbp.addTab("Usuario", new ImageIcon(getClass().getResource("/resources/customer_30px.png")), pnlUsuario, "Gestión de Usuarios");
 tbp.addTab("Pago", new ImageIcon(getClass().getResource("/resources/card_payment_30px.png")), pnlPago, "Gestión de Pagos");
 tbp.addTab("Contacto", new ImageIcon(getClass().getResource("/resources/headset_30px.png")), pnlContacto, "Contáctenos");
 tbp.addTab("Compra", new ImageIcon(getClass().getResource("/resources/buy_30px.png")), pnlCompra, "Portal de compras");

 tbp.setSelectedComponent(pnlUsuario);

 tbp.addChangeListener(new ChangeListener() {
 @Override
 public void stateChanged(ChangeEvent e) {
 EjemploTabbedPane.this.setTitle(tbp.getSelectedComponent().getName());
 }
 });
 }

 this.add(tbp);
}
```

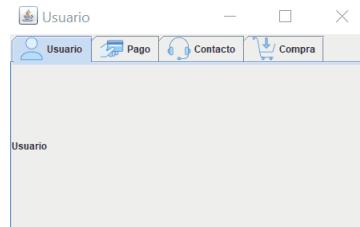
```

 this.setVisible(true);
 this.setDefaultCloseOperation(EXIT_ON_CLOSE);
 }

 public static void main(String[] args) {
 new EjemploTabbedPane();
 }
}

```

#### La salida:



#### JSplitPane

Un `JSplitPane` contiene dos componentes separados por un divisor. Arrastrando el divisor, el usuario puede especificar qué cantidad de área pertenece a cada componente.

##### Orientación

- **Horizontal:** `JSplitPane.HORIZONTAL_SPLIT`
- **Vertical:** `JSplitPane.VERTICAL_SPLIT`

##### Constructores

```

JSplitPane jsp = new JSplitPane(); // Crea un panel dividido en horizontal y con dos botones como componentes
JSplitPane jsp = new JSplitPane(JSplitPane.VERTICAL_SPLIT); // Crea un panel con la orientación indicada SIN COMPONENTES
JSplitPane jsp = new JSplitPane(JSplitPane.VERTICAL_SPLIT, cmp1, cmp2); // Crea un panel con la orientación indicada con dos componentes
JSplitPane jsp = new JSplitPane(JSplitPane.VERTICAL_SPLIT, true); // Se redibujan los componentes según se cambian de tamaño (SIN COMPONENTES)
JSplitPane jsp = new JSplitPane(JSplitPane.VERTICAL_SPLIT, true, cmp1, cmp2); // Idem pero con dos componentes

```

##### Especificar Componentes

```

jsp.setTopComponent(cmp); // Sitúa el primer componente
jsp.setBottomComponent(cmp); // Sitúa el segundo componente

```

##### Modificar el Separador

```

jsp.setDividerLocation(100); // A 100 pixels del inicio del panel
jsp.setDividerLocation(0.5); // En medio del panel (SÓLO FUNCIONA CUANDO LA VENTANA YA ES VISIBLE)
jsp.setDividerSize(10); // Asigna un tamaño de 10 pixels al separador

```

##### Otros Métodos

```

jsp.setOneTouchExpandable(true); // Permite ampliar/contraer los componentes con un solo toque
jsp.setContinuousLayout(true); // Hace que se redibujen los componentes según se redimensiona

```

#### Ejemplo

```

import javax.swing.JFrame;
import static javax.swing.JFrame.EXIT_ON_CLOSE;
import javax.swing.JLabel;
import javax.swing.JSplitPane;

public class PruebaJSplitPane extends JFrame {
 private JSplitPane jspIzquierda = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
 private JSplitPane jspDerecha = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
 private JSplitPane jsp = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, jspIzquierda, jspDerecha);

 public PruebaJSplitPane() {
 this.setSize(500, 400);
 this.setLocation(200, 200);

 jspDerecha.setTopComponent(new JLabel("Arriba Der"));
 jspDerecha.setBottomComponent(new JLabel("Abajo Der"));

 jspIzquierda.setTopComponent(new JLabel("Arriba Izq"));
 jspIzquierda.setBottomComponent(new JLabel("Abajo Izq"));
 jspIzquierda.setDividerLocation(230);

 jsp.setOneTouchExpandable(true);
 jsp.setContinuousLayout(true);
 this.add(jsp);

 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setVisible(true);
 jsp.setDividerLocation(0.5);
 }

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

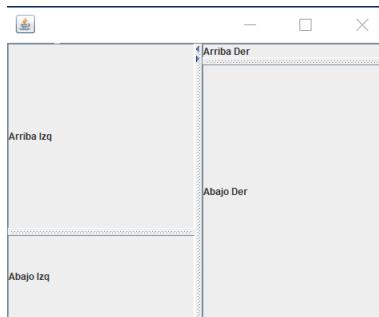
```

```

 PruebaJSplitPane p = new PruebaJSplitPane();
 }
}

```

**La salida:**



### 5.25. Tablas de Datos JTable

#### JTable

Un objeto `JTable` permite mostrar información de forma tabular. Los datos de la tabla se obtienen a través de una clase que implemente `TableModel` (por defecto, una instancia de la clase `DefaultTableModel`) que proporciona métodos para asignar y modificar la información a mostrar. Adicionalmente se puede modificar el aspecto de las tablas mediante una clase que implemente `TableCellRenderer` (o que extienda `DefaultTableCellRenderer`). Con este interfaz se puede especificar el aspecto de cada columna mediante objetos que implementen `CellRenderer`. Por otro lado, es posible proporcionar editores personalizados para las celdas mediante el interfaz `TableCellEditor` (o extendiendo `DefaultTableCellEditor`). Las tablas son un componente bastante complejo así que veremos paso a paso sus características más importantes.

##### Creación básica de una tabla

Por defecto, el modelo de la tabla almacena la información en dos matrices o vectores:

- **títulos:** Es una matriz o un vector de objetos sobre los que se va a ejecutar el método `toString()` y el resultado se mostrará como encabezado de las columnas
- **datos:** Es una matriz de matrices o un vector de vectores en los que, cada elemento de la matriz (o vector) representa una fila de la tabla y es, a su vez otra matriz (o vector) cuyos elementos representan los valores a mostrar en dicha fila

Si partimos de estos datos, podemos emplear los constructores del objeto `JTable` para crearla.

```

import javax.swing.JFrame;
import static javax.swing.JFrame.EXIT_ON_CLOSE;
import javax.swing.JScrollPane;
import javax.swing.JTable;

public class PruebaJTable extends JFrame {
 private Object[][] datos = {
 {"Austria", "Vienna", "https://www.flagsimporter.com/pub/media/catalog/category/AUSTRIA35-2.jpg", 8858775, new GregorianCalendar(2019,0,1).getTime()},
 {"Belgium", "Brussels", "https://www.flagsimporter.com/pub/media/catalog/category/BELGIUM35-2.jpg", 11455519, new GregorianCalendar(2019,1,21).getTime()},
 ...
 {"United Kingdom of Great Britain and Northern Ireland", "London", "https://www.flagsimporter.com/pub/media/catalog/category/UK35-2.jpg", 66647112,
 };
 private Object[] columnas = {"Country", "Capital", "Flag", "Population", "Date", "Upodated"};
 private JTable tbl = new JTable(datos, columnas);
 private JScrollPane jsp = new JScrollPane(tbl);

 public PruebaJTable() {
 this.setSize(400, 200);
 this.setLocation(200, 200);
 this.add(jsp);
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setVisible(true);
 }

 public static void main(String[] args) {
 PruebaJTable p = new PruebaJTable();
 }
}

```

**La salida**

Country	Capital	Flag	Population	Date	Updated
Austria	Vienna	<a href="https://www.flaticon.com/free-icon/flag-austria_315714">https://www.flaticon.com/free-icon/flag-austria_315714</a>	8858775	Tue Jan 01 00:	true
Belgium	Brussels	<a href="https://www.flaticon.com/free-icon/flag-belgium_315715">https://www.flaticon.com/free-icon/flag-belgium_315715</a>	11455519	Thu Feb 21 00:	true
Bulgaria	Sofia	<a href="https://www.flaticon.com/free-icon/flag-bulgaria_315716">https://www.flaticon.com/free-icon/flag-bulgaria_315716</a>	7000039	Sun Mar 03 00:	true
Croatia	Zagreb	<a href="https://www.flaticon.com/free-icon/flag-croatia_315717">https://www.flaticon.com/free-icon/flag-croatia_315717</a>	4075248	Mon Dec 23 00:	false
Cyprus	Nicosia	<a href="https://www.flaticon.com/free-icon/flag-cyprus_315718">https://www.flaticon.com/free-icon/flag-cyprus_315718</a>	8750000	Tue Jan 15 00:	true
Czech Republic	Prague	<a href="https://www.flaticon.com/free-icon/flag-czech-republic_315719">https://www.flaticon.com/free-icon/flag-czech-republic_315719</a>	58060001	Fri Nov 23 00:	false
Denmark	Copenhagen	<a href="https://www.flaticon.com/free-icon/flag-denmark_315720">https://www.flaticon.com/free-icon/flag-denmark_315720</a>	5800000	Thu Jan 02 00:	true
Estonia	Tallinn	<a href="https://www.flaticon.com/free-icon/flag-estonia_315721">https://www.flaticon.com/free-icon/flag-estonia_315721</a>	1324820	Wed Jan 03 00:	true
Finland	Helsinki	<a href="https://www.flaticon.com/free-icon/flag-finland_315722">https://www.flaticon.com/free-icon/flag-finland_315722</a>	5517919	Tue Jan 01 00:	true
France	Paris	<a href="https://www.flaticon.com/free-icon/flag-france_315723">https://www.flaticon.com/free-icon/flag-france_315723</a>	67012883	Sat Jan 13 00:	false
Germany	Berlin	<a href="https://www.flaticon.com/free-icon/flag-germany_315724">https://www.flaticon.com/free-icon/flag-germany_315724</a>	83019213	Tue Jan 01 00:	true
Greece	Athens	<a href="https://www.flaticon.com/free-icon/flag-greece_315725">https://www.flaticon.com/free-icon/flag-greece_315725</a>	10724599	Tue Jan 01 00:	true
Hungary	Budapest	<a href="https://www.flaticon.com/free-icon/flag-hungary_315726">https://www.flaticon.com/free-icon/flag-hungary_315726</a>	9722756	Mon Jun 11 00:	false
Ireland	Dublin	<a href="https://www.flaticon.com/free-icon/flag-ireland_315727">https://www.flaticon.com/free-icon/flag-ireland_315727</a>	4904240	Tue Jan 01 00:	true
Italy	Rome	<a href="https://www.flaticon.com/free-icon/flag-italy_315728">https://www.flaticon.com/free-icon/flag-italy_315728</a>	60359546	Tue Jan 01 00:	true
Latvia	Riga	<a href="https://www.flaticon.com/free-icon/flag-latvia_315729">https://www.flaticon.com/free-icon/flag-latvia_315729</a>	1919965	Thu Jan 03 00:	true
Lithuania	Vilnius	<a href="https://www.flaticon.com/free-icon/flag-lithuania_315730">https://www.flaticon.com/free-icon/flag-lithuania_315730</a>	271184	Thu Jan 01 00:	true
Luxembourg	Luxembourg	<a href="https://www.flaticon.com/free-icon/flag-luxembourg_315731">https://www.flaticon.com/free-icon/flag-luxembourg_315731</a>	6138000	Thu Jan 01 00:	true
Malta	Valletta	<a href="https://www.flaticon.com/free-icon/flag-malta_315732">https://www.flaticon.com/free-icon/flag-malta_315732</a>	493550	Mon Jan 01 00:	false
Netherlands	Amsterdam	<a href="https://www.flaticon.com/free-icon/flag-netherlands_315733">https://www.flaticon.com/free-icon/flag-netherlands_315733</a>	17232163	Tue Jan 01 00:	true
Poland	Warsaw	<a href="https://www.flaticon.com/free-icon/flag-poland_315734">https://www.flaticon.com/free-icon/flag-poland_315734</a>	37972812	Tue Jan 01 00:	true
Portugal	Lisbon	<a href="https://www.flaticon.com/free-icon/flag-portugal_315735">https://www.flaticon.com/free-icon/flag-portugal_315735</a>	10276617	Tue Jan 01 00:	true
Romania	Bucharest	<a href="https://www.flaticon.com/free-icon/flag-romania_315736">https://www.flaticon.com/free-icon/flag-romania_315736</a>	19414458	Tue Jan 01 00:	true
Slovakia	Bratislava	<a href="https://www.flaticon.com/free-icon/flag-slovakia_315737">https://www.flaticon.com/free-icon/flag-slovakia_315737</a>	5450421	Tue Jan 01 00:	true
Slovenia	Ljubljana	<a href="https://www.flaticon.com/free-icon/flag-slovenia_315738">https://www.flaticon.com/free-icon/flag-slovenia_315738</a>	2080900	Tue Jan 01 00:	true
Spain	Madrid	<a href="https://www.flaticon.com/free-icon/flag-spain_315739">https://www.flaticon.com/free-icon/flag-spain_315739</a>	46937060	Tue Jan 01 00:	true

La tabla por defecto es editable, aunque los cambios que hagamos a la misma no son permanentes (si queremos modificar realmente los datos tendremos que hacerlo por código). Podemos cambiar este comportamiento indicando que no queremos editores para ningún tipo de dato con el método `tbl.setDefaultValueEditor(Object.class, null)`. Si los datos los obtenemos dinámicamente, podríamos asignarlos a la tabla con el método `setValueAt(Objeto, fila, columna)` pero es más simple pasar los datos a una matriz / vector y usar el método `setDataVector del objeto DefaultTableModel` que se crea al crear la tabla. Podríamos conseguir lo mismo que en el ejemplo con el siguiente código:

```
DefaultTableModel dtm = (DefaultTableModel) tbl.getModel();
dtm.setDataVector(datos, columnas);
```

#### Seleccionar elementos

Al igual que en las listas, en las tablas disponemos de tres modelos de selección que se especifican con el método `setSelectionMode` y que son los mismos vistos en la clase `JList`. Además para indicar que podemos seleccionar filas hay que emplear el método `setRowSelectionAllowed(true)` (por defecto) y, si queremos seleccionar columnas emplearemos `setColumnSelectionAllowed(true)` (es false por defecto). Las combinaciones posibles son:

<code>tbl.setColumnSelectionAllowed(false)</code>	<code>tbl.setColumnSelectionAllowed(true)</code>
<code>tbl.setSelectionMode(ListSelectionModel.SINGLE_SELECTION)</code>	<code>tbl.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION)</code>
<code>tbl.setRowSelectionAllowed(true)</code>	<code>tbl.setRowSelectionAllowed(true)</code>
<code>tbl.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION)</code>	<code>tbl.setSelectionMode(ListSelectionModel.MULTIPLE_SELECTION)</code>
<code>tbl.setRowSelectionAllowed(true)</code>	<code>tbl.setRowSelectionAllowed(true)</code>
<code>tbl.setSelectionMode(ListSelectionModel.SINGLE_SELECTION)</code>	<code>tbl.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION)</code>
<code>tbl.setRowSelectionAllowed(true)</code>	<code>tbl.setRowSelectionAllowed(false)</code>
<code>tbl.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION)</code>	<code>tbl.setSelectionMode(ListSelectionModel.SINGLE_SELECTION)</code>
<code>tbl.setRowSelectionAllowed(false)</code>	<code>tbl.setRowSelectionAllowed(true)</code>

tbl.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION)		

**NOTA:** También existe el método `setCellSelectionEnabled` que, si se activa activa a su vez la selección de filas y columnas y, si se desactiva, desactiva ambas selecciones. Si necesitamos saber qué elementos están seleccionados actualmente. Por ejemplo, con la siguiente selección:

Country	Capital	Flag	Population	Date	Updated
Austria	Vienna	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	8858775	Tue Jan 01 00:00:00 CET 2019	true
Belgium	Brussels	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	11455519	Thu Feb 21 00:00:00 CET 2019	true
Bulgaria	Sofia	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	700039	Sun Mar 03 00:00:00 CET 2019	true
Croatia	Zagreb	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	4076245	Mon Dec 23 00:00:00 CET 2018	false
Cyprus	Nicosia	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	875899	Tue Jan 15 00:00:00 CET 2019	true
Czech Republic	Prague	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	10849800	Fri Nov 23 00:00:00 CET 2018	false
Denmark	Copenhagen	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	5806801	Thu Jan 10 00:00:00 CET 2019	true
Estonia	Tallinn	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	1324820	Wed Jan 02 00:00:00 CET 2019	true
Finland	Helsinki	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	5517919	Tue Jan 01 00:00:00 CET 2019	true
France	Paris	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	67012883	Sat Jan 13 00:00:00 CET 2019	false
Germany	Berlin	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	83019213	Tue Jan 01 00:00:00 CET 2019	true
Greece	Athens	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	10724599	Tue Jan 01 00:00:00 CET 2019	true
Hungary	Budapest	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	9772756	Mon Jun 11 00:00:00 CEST 2018	false
Ireland	Dublin	<a href="https://www.flaticon.com">https://www.flaticon.com</a>	4904240	Tue Jan 01 00:00:00 CET 2019	true

tbl.getSelectedRow()	1
tbl.getSelectedRowCount()	5
tbl.getSelectedRows()	1, 6, 7, 8, 10
tbl.getSelectedColumn()	1
tbl.getSelectedColumnCount()	3
tbl.getSelectedColumns()	1, 2, 4

Para seleccionar celdas tenemos que combinar los siguientes métodos

```
tbl.setRowSelectionInterval(0,0); // Selecciona la primera fila
tbl.addRowSelectionInterval(2,3); // Añade a la selección las filas 3 y 4
tbl.setColumnSelectionInterval(4); // Sólo se selecciona la quinta columna de las filas anteriores
```

#### Responder a una selección de datos

Si queremos realizar alguna operación cuando el usuario selecciona algo (celda, fila, columna) debemos añadir un receptor de eventos `ListSelectionEvent` al modelo de selección de la tabla (que recuperamos con `getSelectionModel`):

```
tbl.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
 @Override
 public void valueChanged(ListSelectionEvent e) {
 if (!e.getValueIsAdjusting()){
 System.out.println("Valores Seleccionados:");
 for(int i:tbl.getSelectedRows()){
 for(int j:tbl.getSelectedColumns()) {
 System.out.println(tbl.getModel().getValueAt(i, j)+"\t");
 }
 System.out.println("");
 }
 }
 }
});
```

En el ejemplo anterior:

```
Valores Seleccionados:
Brussels https://www.flagsimporter.com/pub/media/catalog/category/BELGIUM35-2.jpg Thu Feb 21 00:00:00 CET 2019
Copenhagen https://www.flagsimporter.com/pub/media/catalog/category/DENMARK35-2.jpg Thu Jan 10 00:00:00 CET 2019
Tallinn https://www.flagsimporter.com/pub/media/catalog/category/ESTONIA35-2.jpg Wed Jan 02 00:00:00 CET 2019
Helsinki https://www.flagsimporter.com/pub/media/catalog/category/FINLAND35-2.jpg Tue Jan 01 00:00:00 CET 2019
Berlin https://www.flagsimporter.com/pub/media/catalog/category/GERMANY35-2.jpg Tue Jan 01 00:00:00 CET 2019
```

#### Modificar el aspecto de los datos

En el caso de las tablas la forma de modificar el aspecto de los datos es diferente a otros componentes de Swing. En este caso se especifica un renderizador por tipo de columna. El renderizador por defecto de un `JTable` muestra la información de la siguiente forma:

- Un `JLabel` con el texto en el caso de objetos de tipo `String`
- Un `JCheckBox` en el caso de valores `Boolean`
- Un `JLabel` con el número en el caso de un `Integer` alineado a la derecha
- Un `JLabel` con el número formateado en el caso de un `Double` o un `Float`
- Un `JLabel` con formato corto de fecha y hora para objetos `Date`
- Un `JLabel` con la imagen para objetos `Icon` y `ImageIcon`

- Un `JLabel` con el resultado de llamar al método `toString()` para `Object` (el resto)

¿Cómo sabe el renderizador qué tipo de objeto hay en cada columna? Llamando al método `getClass(columna)` del modelo. **PROBLEMA:** El método `getClass` del modelo por defecto retorna `Object.class` para todas las columnas. Por eso, nuestra tabla muestra los datos así. **SOLUCIÓN:** Crear nuestro propio `DefaultTableModel` y redefinir el método `getClass` para las columnas

```
import java.util.Date;
import java.util.GregorianCalendar;
import javax.swing.JFrame;
import static javax.swing.JFrame.EXIT_ON_CLOSE;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import javax.swing.table.DefaultTableModel;

public class PruebaJTable extends JFrame {
 private Object[][] datos = {...};
 private Object[] columnas = {"Country", "Capital", "Flag", "Population", "Date", "Updated"};
 private JTable tbl = new JTable();
 private JScrollPane jsp = new JScrollPane(tbl);

 public PruebaJTable() {
 this.setSize(600, 300);
 this.setLocation(200, 200);
 DefaultTableModel dtm = new DefaultTableModel() {
 @Override
 public Class<?> getColumnClass(int columnIndex) {
 switch (columnIndex) {
 case 3:
 return Double.class;
 case 4:
 return Date.class;
 case 5:
 return Boolean.class;
 }
 return super.getColumnClass(columnIndex);
 }
 };
 dtm.setDataVector(datos, columnas);
 tbl.setModel(dtm);
 tbl.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
 tbl.setRowSelectionAllowed(true);
 tbl.setColumnSelectionAllowed(true);
 tbl.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
 @Override
 public void valueChanged(ListSelectionEvent e) {
 if (!e.getValueIsAdjusting()) {
 System.out.println("Valores Seleccionados:");
 for(int i:tbl.getSelectedRows()){
 for(int j:tbl.getSelectedColumns()){
 System.out.println(tbl.getModel().getValueAt(i, j)+"\t");
 }
 }
 System.out.println("");
 }
 }
 });
 this.add(jsp);
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setVisible(true);
 }

 public static void main(String[] args) {
 PruebaJTable p = new PruebaJTable();
 }
}
```

La salida (observar los valores resaltados):

Country	Capital	Flag	Population	Date	Updated
Austria	Vienna	https://www.flaticon.com	8.858.775	01-ene-2019	✓
Belgium	Brussels	https://www.flaticon.com	11.455.519	21-feb-2019	✓
Bulgaria	Sofia	https://www.flaticon.com	7.000.039	03-mar-2019	✓
Croatia	Zagreb	https://www.flaticon.com	4.076.246	23-dic-2019	✓
Cyprus	Nicosia	https://www.flaticon.com	875.830	01-jun-2019	✓
Czech Republic	Praha	https://www.flaticon.com	10.642.000	25-noviembre-2018	✓
Danmark	København	https://www.flaticon.com	5.806.081	10-ene-2019	✓
Estonia	Tallinn	https://www.flaticon.com	1.324.820	02-ene-2019	✓
Finland	Helsinki	https://www.flaticon.com	5.517.919	01-ene-2019	✓
France	Paris	https://www.flaticon.com	67.012.883	13-ene-2018	✓
Germany	Berlin	https://www.flaticon.com	83.019.213	01-ene-2019	✓
Greece	Athens	https://www.flaticon.com	10.724.599	01-ene-2019	✓
Hungary	Budapest	https://www.flaticon.com	9.772.756	11-jun-2018	✓
Ireland	Dublin	https://www.flaticon.com	4.904.240	01-ene-2019	✓

¿Y si queremos crear nuestro propio renderizado? Tenemos que crear una clase que implemente `TableCellRenderer` (o extienda `DefaultTableCellRenderer`). Una vez creado podemos aplicarlo a todas las columnas de la clase indicada:

```
tbl.setDefaultRenderer(String.class, new MiRenderer());
```

o a una columna en particular. En este caso necesitamos un objeto  `TableColumn` que obtenemos del  `TableColumnModel`:

```
tbl.getColumnModel().getColumn(pos).setCellRenderer(new MiRender());
```

Veamos un ejemplo:

```
import java.awt.Color;
...
public class PruebaJTable extends JFrame {
 private Object[][] datos = { ... };
 private Object[] columnas = { ... };
 private JTable tbl = new JTable();
 private JScrollPane jsp = new JScrollPane(tbl);

 public PruebaJTable() {
 this.setSize(600, 300);
 this.setLocation(200, 200);
 DefaultTableModel dtm = new DefaultTableModel() {
 ...
 };
 dtm.setDataVector(datos, columnas);
 tbl.setModel(dtm);
 tbl.setRowHeight(60); // Especificamos el ALTO de la fila
 tbl.getColumnModel().getColumn(2).setCellRenderer(new FlagRender()); // Asignamos el renderizador
 ...
 this.add(jsp);

 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setVisible(true);
 }

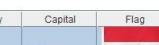
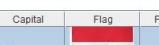
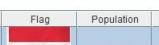
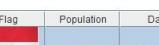
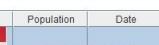
 class FlagRender implements TableCellRenderer {
 @Override
 public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {
 JLabel lbl = new JLabel();
 try {
 ImageIcon imIc = new ImageIcon(new URL(value.toString()));
 Image imag = imIc.getImage();
 imag = imag.getScaledInstance(80, 60, Image.SCALE_SMOOTH);
 imIc.setImage(imag);
 lbl.setIcon(imIc);
 lbl.setHorizontalAlignment(JLabel.CENTER);
 } catch (MalformedURLException ex) {
 Logger.getLogger(PruebaJTable.class.getName()).log(Level.SEVERE, null, ex);
 }
 lbl.setOpaque(true);
 if (isSelected) {
 lbl.setBackground(table.getSelectionBackground());
 } else {
 lbl.setBackground(Color.WHITE);
 }
 return lbl;
 }
 }
}

public static void main(String[] args) {
 PruebaJTable p = new PruebaJTable();
}
```

La salida:



A screenshot of a Java Swing application window titled "PruebaJTable". The window contains a JTable with the following data:

Country	Capital	Flag	Population	Date	Updated
Austria	Vienna		8.858.775	01-ene-2019	<input type="checkbox"/>
Belgium	Brussels		11.455.519	21-feb-2019	<input checked="" type="checkbox"/>
Bulgaria	Sofia		7.000.039	03-mar-2019	<input checked="" type="checkbox"/>
Croatia	Zagreb		4.076.246	23-dic-2019	<input type="checkbox"/>
Cyprus	Nicosia		875.899	15-ene-2019	<input checked="" type="checkbox"/>
Czech Republic	Prague		10.649.800	23-nov-2018	<input type="checkbox"/>

#### Ordenar la tabla

Mediante el método `setAutoCreateRowSorter(true)` podemos pedirle a Swing que cree un modelo de ordenación por defecto para los campos de la tabla. Este modelo comproueba el tipo de datos retornado por `getClass` para la columna y ordena la misma primero comprobando si implementa el interfaz `Comparable` y, si no, llamando al método `toString()` y ordenando en base a ese valor. Para ordenar una columna en ascendente basta con hacer clic en su nombre. Al pulsar sucesivamente se irá cambiando entre ordenación ascendente / descendente:

Country	Capital	Flag	Population	Date	Updated
Germany	Berlin		83.019.213	01-ene-2019	<input checked="" type="checkbox"/>
France	Paris		67.012.883	13-ene-2018	<input type="checkbox"/>
United Kingdom	London		66.647.112	01-ene-2019	<input checked="" type="checkbox"/>
Italy	Rome		60.359.546	01-ene-2019	<input checked="" type="checkbox"/>
Spain	Madrid		46.937.060	01-ene-2019	<input checked="" type="checkbox"/>

#### Permitir la modificación de una tabla

Al igual que cada clase tiene un visualizador, de la misma forma, cada clase tiene su propio editor. Dicho editor se implementa mediante la clase `DefaultCellEditor` y por defecto es un `JTextField` (excepto para valores booleanos que es un `JCheckBox`). Si queremos modificar el editor para una clase o una columna hay que crear una clase que implemente `TableCellEditor` (la clase `DefaultCellEditor` tiene un constructor que permite crear un editor con un `JTextField`, un `JCheckBox` o un `JComboBox`). Aunque visualmente podemos modificar una celda, el cambio en el modelo de datos corre por nuestra cuenta. Por ejemplo, si modificamos el contenido de la primera celda (haciendo doble clic o pulsando F2):

Country	Capital	Flag	Population	Date	Updated
Austria	Vienna		8.858.775	01-ene-2019	<input checked="" type="checkbox"/>
Belgium	Brussels		11.455.519	21-feb-2019	<input checked="" type="checkbox"/>
Bulgaria	Sofia		7.000.039	03-mar-2019	<input checked="" type="checkbox"/>
Croatia	Zagreb		4.076.246	23-dic-2019	<input type="checkbox"/>

Si mostramos el contenido de los datos, veremos que sigue apareciendo Austria (en lugar de Australia).

Para saber que el usuario ha modificado una celda se lanza el evento `TableModelEvent` en el modelo de datos. Podemos capturarlo y añadir el código necesario para actualizar los datos:

```
tbl.getModel().addTableModelListener(new TableModelListener() {
 @Override
 public void tableChanged(TableModelEvent e) {
 datos[e.getFirstRow()][e getColumn()] = tbl.getValueAt(e.getFirstRow(), e.getColumn());
 }
});
```

¿Cómo hacer que las celdas no sean editables? Lo tenemos que hacer en el modelo de datos sobrescribiendo el método `isCellEditable`:

```
DefaultTableModel dtm = new DefaultTableModel() {
 @Override
 public Class<?> getColumnClass(int columnIndex) {
 switch (columnIndex) {
 case 3:
 return Double.class;
 case 4:
 return Date.class;
 case 5:
 return Boolean.class;
 }
 return super.getColumnClass(columnIndex);
 }

 @Override
 public boolean isCellEditable(int row, int column) {
 return column==4 || column==5;
 }
};
```

Como se puede ver, ahora sólo se pueden editar las dos últimas columnas:

Country	Capital	Flag	Population	Date	Updated
Austria	Vienna		8.858.775	01-ene-2019	<input checked="" type="checkbox"/>
Belgium	Brussels		11.455.519	21-feb-2019	<input checked="" type="checkbox"/>
Bulgaria	Sofia		7.000.039	03-mar-2019	<input checked="" type="checkbox"/>
Croatia	Zagreb		4.076.246	23-dic-2019	<input type="checkbox"/>

#### Editores Personalizados

El último problema que tiene nuestra aplicación es que los editores personalizados no son suficientes en todos los casos. Podemos crear nuestros propios editores creando una clase que implemente `TableCellRenderer`. Dado que esta clase debe lanzar eventos cuando se modifican los datos, es mejor extender de la clase `AbstractCellRenderer` que ya hace eso por nosotros. Los editores se aplican a las columnas, por lo que deberemos previamente seleccionar la columna deseada del modelo de columnas. Por defecto, un `JTable` emplea un `DefaultTableCellEditor` que emplea un `JCheckBox` para los valores booleanos y un `JTextField` para el resto (en

este caso se emplea el método `toString()` para obtener el valor a mostrar). Esto implica que, si el valor es un texto o un valor lógico funciona perfectamente pero no es válido para el resto (por ejemplo no podemos editar la fecha). La clase `DefaultTableCellEditor` tiene un constructor que, además de esos controles, admite un `JComboBox` así que puede ser una buena opción para cuando necesitemos una lista de valores (de nuevo sólo funcionará bien con cadenas). Para el caso del campo de Fecha podemos crear un editor personalizado. De este editor necesitaremos implementar dos métodos `getTableCellEditorComponent` que retorna el editor para una celda y un valor específico `getCellEditorValue` que se llama tras la edición y debe retornar el valor a asignar al modelo de datos (en nuestro caso una fecha). En nuestro caso emplearemos un `JFormattedTextField` como componente y su valor como dato a retornar.

```
...
tbl.getColumnModel().getColumn(4).setCellEditor(new DateEditor()); // El campo fecha
...

class DateEditor extends AbstractCellEditor implements TableCellEditor{
 private JFormattedTextField ftx;

 @Override
 public Object getCellEditorValue() {
 return (ftx.getValue());
 }

 @Override
 public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int row, int column) {
 ftx = new JFormattedTextField(DateFormat.getDateInstance(DateFormat.SHORT));
 datos[row][column] = (Date) value;
 ftx.setValue(value);
 return ftx;
 }
}
```

Ahora ya podemos editar las fechas:

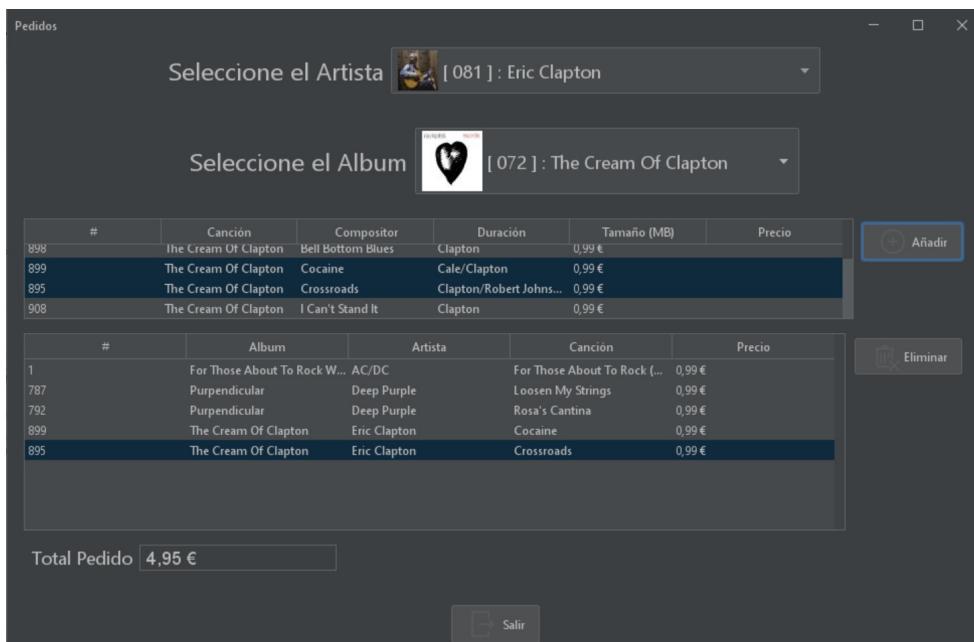
Country	Capital	Flag	Population	Date	Updated
Austria	Vienna		8.858.775	01-ene-2019	<input checked="" type="checkbox"/>
Belgium	Brussels		11.455.519	21-feb-2019	<input checked="" type="checkbox"/>
Bulgaria	Sofia		7.000.039	03-mar-2019	<input checked="" type="checkbox"/>
Croatia	Zagreb		4.076.246	23-dic-2019	<input type="checkbox"/>

#### 5.26. Actividad 31 - Uso de JTable

##### Instrucciones

###### [Classroom](#)

Se desea diseñar una aplicación de Pedidos que, al finalizar la aplicación, deberá tener un aspecto similar al siguiente:



Para hacer el ejercicio partiremos de la aplicación Chinook a la que añadiremos los ficheros `alumnos.dat`, `artistas.dat` y `pistas.dat`. Adicionalmente se proporcionan imágenes de todos los artistas y álbumes en el archivo `imagenes.zip`.

**NOTA:** Las imágenes de los artistas están identificadas por su id, mientras que los álbumes son ALBId. Por ejemplo, el artista con id 42 tendrá la imagen `42.jpg` y el álbum con el mismo id será `ALB42.jpg`.

##### Crear el modelo

Deberemos definir las clases `Artista` (`id entero, nombre cadena`), `Album` (`id entero, idArtista entero, titulo cadena`) y `Pista` (`id entero, nombre cadena, idAlbum entero, compositor cadena, ms entero, bytes entero, precio double`).

Las tres clases tendrán un constructor, métodos `set/get` y un método `equals` basado en el `id`. Además, la clase `Pista` tendrá los siguientes métodos:

- `String getDuracion()`: Retornará la duración de la canción en el formato mm:ss. Por ejemplo, si una canción dura 590.000 ms, deberá retornar 09:50.
- `String getTamaño()`: Retornará el tamaño de la canción en MB con dos decimales. Por ejemplo, si una canción ocupa 11.170.334 bytes deberá retornar 10,65 MB.

## Crear el DAO

Se deberán definir los siguientes DAOs:

### AlbumArtistaPistaDAO / AlbumArtistaPistaFich

Por simplicidad vamos a crear un único interfaz y clase para las tres clases.

Métodos del interfaz:

- `List<Album> getAlbumes(Integer idArtista)`
- `List<Album> getAlbumes()`
- `Album getAlbum(Integer id)`
- `List<Artista> getArtistas()`
- `Artista getArtista(Integer id)`
- `List<Pista> getPistasAlbum(Integer idAlbum)`

En la clase se definirán las propiedades `albumes`, `pistas` y `artistas` y, en el constructor, se añadirán los álbumes, las pistas y los artistas de los ficheros correspondientes a cada una de ellas.

- En la misma clase, el método `getAlbumes(Integer idArtista)` deberá retornar todos los álbumes del artista indicado ordenados alfabéticamente por título.
- El método `getAlbumes()` retornará todos los álbumes ordenados alfabéticamente por título.
- El método `getAlbum(Integer id)` retornará el álbum con el id indicado o un valor `null` si no existe.
- El método `getArtistas()` retornará todos los artistas ordenados alfabéticamente por nombre.
- El método `getArtista(Integer id)` retornará el artista con el id indicado o un valor `null` si no existe.
- El método `getPistasAlbum(Integer idAlbum)` deberá retornar todas las pistas del álbum indicado ordenadas alfabéticamente por nombre.

### PedidoDAO / PedidoList

El interfaz dispondrá de los siguientes métodos:

- `void nuevaPista(Pista p)`
- `void eliminarPista(Integer id)`
- `Double getImportePedido()`
- `List<Pista> getPistas()`

En la clase se definirá una propiedad llamada `pistas` para almacenar todas las pistas del pedido actual.

- El método `nuevaPista(Pista pista)` añadirá la pista a la propiedad `pistas`. **NOTA:** Si ya existe una pista en `pistas` con el mismo `id` que la que se quiere añadir, no se añadirá.
- El método `eliminarPista(Integer idPista)` eliminará la pista con el `idPista` indicado de la propiedad `pistas`.
- El método `getImportePedido()` retornará la suma de los precios de todas las pistas almacenadas hasta el momento.
- El método `getPistas()` retornará las pistas.

## Servicio

Se añadirán los daos al servicio y se definirán en el mismo todos los métodos anteriores (lo único los métodos `nuevaPista`, `eliminarPista` y `getPistas` se llamarán en el servicio `nuevaPistaPedido`, `eliminarPistaPedido` y `getPistasPedido`).

## Aplicación

La aplicación, como siempre, empleará la información del servicio. Tendremos como propiedades los títulos y los datos de las tablas tal y como se vio en el ejemplo de clase (como `Vector<String>` y `Vector<Vector<Object>>` respectivamente). Necesitamos 4 propiedades dado que son dos tablas distintas.

### Paso 1

En el constructor y, tras llamar a `initComponents()`:

- Se añadirán los títulos a las tablas correspondientes
- Se añadirán todos los artistas del servicio al `JComboBox` de artistas
- Se pondrá como importe del pedido 0.0 (deberá ser un `JFormattedTextField`).

- Se cambiará el renderizado de los `JComboBox` de artista y álbum para que se muestren tal y como se ve en el ejemplo

Al seleccionar un artista se llamará a un método `cargarAlbumes()` que realizará las siguientes tareas:

- Obtendrá el artista seleccionado actualmente (`getSelectedItem()`) en la lista desplegable
- Si no hemos obtenido un valor `null` (siempre es bueno poner esta condición al controlar que se ha seleccionado algo en los `JComboBox` para evitar excepciones)
- Eliminaremos todos los elementos del `JComboBox` de álbumes
- Recuperaremos todos los álbumes de ese artista desde el servicio
- Los añadiremos al `JComboBox`

**Comprobación:** Si ejecutamos, al iniciar el formulario y al seleccionar un artista, deberían aparecer sus álbumes en la lista.

## Paso 2

Al seleccionar un álbum se llamará a un método `cargarPistas()` que realizará las siguientes tareas:

- Obtendrá el álbum seleccionado de la lista desplegable
- Si no es `null`
- Recuperará todas las pistas del álbum a través del servicio
- Eliminará los datos del vector de datos de la tabla de pistas
- Por cada pista del álbum
  - Creará un `Vector<Object>` llamado `vctFila`
  - Añadirá la información de la pista a dicho vector
  - Añadirá la fila al vector de datos de la tabla de pistas
- Creará un `DefaultTableModel` con el vector de datos recién creado y los títulos de la tabla de pistas
- Lo asignará como modelo (`setModel()`) a la tabla de pistas

**Comprobación:** Si ejecutamos, al iniciar el formulario y al seleccionar un artista o un álbum deberían aparecer las pistas del álbum en la tabla superior.

## Paso 3

Al pulsar el botón de Añadir:

- Si no hay ninguna pista seleccionada en la tabla superior de pistas (`getSelectedRows().length == 0`)
- Se dará un mensaje de error adecuado
- Si hay pistas seleccionadas
- Almacenaremos en una variable `pistasAlbum` todas las pistas del álbum seleccionado actualmente
- Almacenaremos en una matriz de enteros las posiciones de las pistas que ha seleccionado el usuario (`getSelectedRows()`)
- Ahora recorreremos la matriz de posiciones y, con cada una de ellas:
  - Recuperaremos la pista que está en esa posición de la variable `pistasAlbum` creada anteriormente
  - Añadiremos dicha pista al pedido a través del servicio (`nuevaPistaPedido()`)
- Tras ello llamará al método `cargarPedido()`

El método `cargarPedido()`:

- Recuperará todas las pistas del pedido a través del servicio (`getPistasPedido()`)
- Eliminará los datos del vector de datos de la tabla de pedidos
- Por cada pista del pedido
- Creará un `Vector<Object>` llamado `vctFila`
- Añadirá la información de la pista a dicho vector
- Añadirá la fila al vector de datos de la tabla de pedidos
- Creará un `DefaultTableModel` con el vector de datos recién creado y los títulos de la tabla de pedidos
- Lo asignará como modelo (`setModel()`) a la tabla de pedidos
- Asignará al campo `importePedido` el importe del pedido del servicio (`getImportePedido()`)

**Comprobación:** Si ejecutamos, al seleccionar pistas y pulsar sobre Añadir deberán aparecer en la tabla inferior.

## Paso 5

Al pulsar el botón de Eliminar:

- Si no hay ninguna pista seleccionada en la tabla inferior de pedido (`getSelectedRows().length == 0`)
- Se dará un mensaje de error adecuado
- Si hay pistas seleccionadas
- Almacenaremos en una variable `pistasPedido` todas las pistas del pedido
- Almacenaremos en una matriz de enteros las posiciones de las pistas que ha seleccionado el usuario (`getSelectedRows()`)
- Ahora recorreremos la matriz de posiciones y, con cada una de ellas:
  - Recuperaremos la pista que está en esa posición de la variable `pistasPedido` creada anteriormente

- Eliminaremos dicha pista del pedido a través del servicio (`eliminarPistaPedido`)
- Tras ello llamará al método `cargarPedido()`

**Comprobación:** Si ejecutamos, al seleccionar pistas y pulsar sobre Eliminar deberán aparecer en la tabla inferior.

#### Paso 6

El botón Salir simplemente finalizará el proceso.

#### 5.27. Árboles JTree

##### JTree: Mostrando Información Jerarquizada

Cuando necesitamos mostrar información jerarquizada, un árbol (`JTree`) puede ser la solución. Cada elemento de un árbol se sitúa en una fila distinta y se denomina nodo (debe implementar el interfaz `Node`). El árbol parte de un único nodo llamado el nodo raíz (`Root Node`).

Un nodo sólo puede tener un padre, pero puede tener de 0 a n hijos. Un nodo que tiene hijos se denomina una rama (`branch`) mientras que un nodo terminal, se denomina hoja (`leaf`). Los nodo rama pueden expandirse/contraerse haciendo clic en ellos (por defecto están contraídos excepto el raíz). Un nodo se identifica por su fila o por el camino de todos los nodos que llevan a un nodo específico se almacena en un objeto `TreePath`.

Al igual que otros componentes Swing, un `JTree` dispone de:

- Un modelo de datos (objeto que implemente `TreeModel` o extienda `DefaultTreeModel`)
- Un modelo de selección (objeto que implemente `TreeSelectionModel` o extienda `DefaultTreeSelectionModel`)
- Un visualizador (objeto que implemente `TreeCellRenderer` o extienda `DefaultTreeCellrenderer`)
- Un editor (objeto que implemente `TreeCellEditor` o extienda `DefaultTreeCellEditor`)

#### Creación de un árbol

Aunque podemos crear árboles a partir de diferentes estructuras o de un modelo de datos, la forma más simple de crear un árbol es mediante un nodo raíz al que le vamos añadiendo sus descendientes. Para poder crear y añadir nodos a otros nodos, vamos a emplear la clase `DefaultMutableTreeNode` que implementa el interfaz `Node` y nos permite crear/modificar/eliminar nodos dinámicamente, así como añadir nodos hijos a otros nodos.

Un objeto `DefaultMutableTreeNode` puede almacenar un objeto de cualquier tipo y, por defecto, a la hora de mostrarse se llamará al método `toString()` de dicho objeto.

Vamos a trabajar con la clase `Empleado`:

```
public class Empleado {
 private int id;
 private String nombre;
 private String apellidos;
 private Date fechaContratacion;
 private Sexo sexo;
 private String puesto;
 private double salario;
 private String observaciones;
 private String foto;

 public enum Sexo {
 VARON, MUJER
 }

 public Empleado(int id, String nombre, String apellidos, Date fechaContratacion,
 Sexo sexo, String puesto, double salario, String observaciones) {
 this.id = id;
 this.nombre = nombre;
 this.apellidos = apellidos;
 this.fechaContratacion = fechaContratacion;
 this.sexo = sexo;
 this.puesto = puesto;
 this.salario = salario;
 this.observaciones = observaciones;

 // Generamos una URL para una foto de avatar genérica
 if (sexo == Sexo.VARON) {
 this.foto = "https://randomuser.me/api/portraits/men/" + (id % 100) + ".jpg";
 } else {
 this.foto = "https://randomuser.me/api/portraits/women/" + (id % 100) + ".jpg";
 }
 }

 // Getters y setters
 public int getId() {
 return id;
 }

 public void setId(int id) {
 this.id = id;
 }

 public String getNombre() {
 return nombre;
 }

 public void setNombre(String nombre) {
 this.nombre = nombre;
 }

 public String getApellidos() {
```

```

 return apellidos;
 }

 public void setApellidos(String apellidos) {
 this.apellidos = apellidos;
 }

 public Date getFechaContratacion() {
 return fechaContratacion;
 }

 public void setFechaContratacion(Date fechaContratacion) {
 this.fechaContratacion = fechaContratacion;
 }

 public Sexo getSexo() {
 return sexo;
 }

 public void setSexo(Sexo sexo) {
 this.sexo = sexo;
 }

 public String getPuesto() {
 return puesto;
 }

 public void setPuesto(String puesto) {
 this.puesto = puesto;
 }

 public double getSalario() {
 return salario;
 }

 public void setSalario(double salario) {
 this.salario = salario;
 }

 public String getObservaciones() {
 return observaciones;
 }

 public void setObservaciones(String observaciones) {
 this.observaciones = observaciones;
 }

 public String getFoto() {
 return foto;
 }

 public void setFoto(String foto) {
 this.foto = foto;
 }

 @Override
 public String toString() {
 SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
 return apellidos + ", " + nombre + "(" + sdf.format(fechaContratacion) + ")";
 }
}

YCategoria:

public class Categoria {
 private String categoria;
 private ImageIcon logo;

 public Categoria(String categoria, String logo) {
 this.categoria = categoria;
 this.logo = new ImageIcon(logo);
 }

 public String getCategoria() {
 return categoria;
 }

 public void setCategoria(String categoria) {
 this.categoria = categoria;
 }

 public ImageIcon getLogo() {
 return logo;
 }

 public void setLogo(ImageIcon logo) {
 this.logo = logo;
 }

 @Override
 public String toString() {
 return categoria;
 }
}

```

```
}
```

Y el código inicial

```
public class PruebaJTree extends JFrame {
 private JTree arbol;
 private List<Empleado> listaEmpleados;
 private Categoría[] categorías = {
 new Categoría("Dirección", "direccion.png"),
 new Categoría("Administración", "administracion.png"),
 new Categoría("Ventas", "ventas.png")
 };

 public PruebaJTree() {
 this.setSize(500, 400);
 this.setLocation(200, 200);
 inicializarEmpleados();
 crearArbol();
 this.add(this.arbol);
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setVisible(true);
 }

 private void inicializarEmpleados() {
 listaEmpleados = new ArrayList<>();

 // Añadimos varios empleados de ejemplo
 listaEmpleados.add(new Empleado(1, "Juan", "López Marín",
 new GregorianCalendar(2000, 1, 23).getTime(),
 Empleado.Sexo.VARON, "Dirección", 2575.34,
 "Licenciado en Gestión y Administración de Empresas\n\nFundador de la empresa"));

 listaEmpleados.add(new Empleado(12, "María", "Gómez Pérez",
 new GregorianCalendar(2005, 5, 15).getTime(),
 Empleado.Sexo.MUJER, "Administración", 1850.75,
 "Máster en Contabilidad y Finanzas"));

 listaEmpleados.add(new Empleado(23, "Carlos", "Martínez Ruiz",
 new GregorianCalendar(2010, 3, 7).getTime(),
 Empleado.Sexo.VARON, "Ventas", 1950.25,
 "Especialista en Marketing Digital"));

 listaEmpleados.add(new Empleado(34, "Laura", "Fernández García",
 new GregorianCalendar(2012, 8, 12).getTime(),
 Empleado.Sexo.MUJER, "Ventas", 1750.50,
 "Grado en Marketing y Publicidad"));

 listaEmpleados.add(new Empleado(45, "Pedro", "Sánchez Navarro",
 new GregorianCalendar(2015, 10, 3).getTime(),
 Empleado.Sexo.VARON, "Administración", 1650.30,
 "Técnico Superior en Administración y Finanzas"));

 listaEmpleados.add(new Empleado(56, "Ana", "Rodríguez Vázquez",
 new GregorianCalendar(2018, 2, 20).getTime(),
 Empleado.Sexo.MUJER, "Ventas", 1550.45,
 "Grado en Comercio Internacional"));

 listaEmpleados.add(new Empleado(65, "Sonia", "López Marín",
 new GregorianCalendar(2011, 11, 4).getTime(),
 Empleado.Sexo.MUJER, "Ventas", 1040.5, ""));
 }

 private void crearArbol() {
 DefaultMutableTreeNode raiz = new DefaultMutableTreeNode("Empleados");

 for (Categoría c : categorías) {
 DefaultMutableTreeNode dtmCat = new DefaultMutableTreeNode(c);

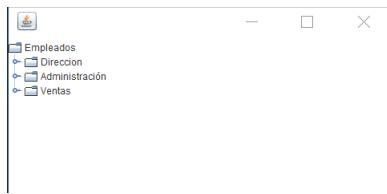
 for (Empleado e : listaEmpleados) {
 if (e.getPuesto().equals(c.getCategoría())) {
 dtmCat.add(new DefaultMutableTreeNode(e));
 }
 }

 raiz.add(dtmCat);
 }

 arbol = new JTree(raiz);
 }

 public static void main(String[] args) {
 PruebaJTree p = new PruebaJTree();
 }
}
```

La salida:



### Seleccionando elementos

A la hora de decidir qué elementos se pueden seleccionar, podemos emplear el método `setSelectionMode` del modelo de selección por defecto (`getSelectionMode`). Las opciones son:

TreeSelectionModel.DISCONTIGUOUS_TREE_SELECTION	Permite seleccionar cualquier elemento (por defecto)	
TreeSelectionModel.CONTIGUOUS_TREE_SELECTION	Seleccionar elementos juntos	
TreeSelectionModel.SINGLE_TREE_SELECTION	Sólo un elemento	

Por ejemplo el código para permitir seleccionar un único nodo:

```
this.arbol.getSelectionModel().setSelectionMode(TreeSelectionModel.SINGLE_TREE_SELECTION);
```

¿Cómo podemos saber cuál es el elemento seleccionado? Podemos emplear los métodos:

- `getSelectionPath()` (devuelve el `TreePath` al primer nodo seleccionado)
- `getSelectionPaths()` (una matriz con los `TreePath` seleccionados)
- `getSelectionRows()` (una matriz con las filas seleccionadas)

A partir de un `TreePath` podemos obtener el objeto seleccionado con `getLastPathComponent()`. También podemos recuperar cualquiera de los nodos padre con `getPathComponent(pos)` o saber cuántos hay con `getPathCount()`, etc. **NOTA:** los métodos que retornan un nodo, devolverán un `DefaultMutableTreeNode`. En este nodo, el objeto del usuario (lo que hemos pasado al nodo) lo podemos recuperar con `getUserObject()`.

Si queremos saber cuándo cambia la selección podemos añadir un receptor de eventos `TreeSelectionEvent` al modelo de selección. Este evento, entre otros métodos, nos permite obtener el camino al elemento seleccionado (`getPath()`). Por ejemplo, el siguiente código nos mostrará los datos de los empleados seleccionados:

```
public class PruebaJTree extends JFrame {
 private JTree arbol;
 private List<Empleado> listaEmpleados;
 private Categoria[] categorias = {
 ...
 };

 public PruebaJTree() {
 this.setSize(500, 400);
 this.setLocation(200, 200);
 inicializarEmpleados();
 crearArbol();
 this.add(this.arbol);

 configurarSeleccion();

 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setVisible(true);
 }

 private void inicializarEmpleados() {
 ...
 }

 private void crearArbol() {
 ...
 }

 private void configurarSeleccion() {
 this.arbol.getSelectionModel().setSelectionMode(TreeSelectionModel.SINGLE_TREE_SELECTION);

 this.arbol.getSelectionModel().addTreeSelectionListener(new TreeSelectionListener() {
 @Override
 public void valueChanged(TreeSelectionEvent e) {
 DefaultMutableTreeNode tn = (DefaultMutableTreeNode) e.getPath().getLastPathComponent();

 if (tn.getUserObject() instanceof Empleado) {
 Empleado emp = (Empleado) tn.getUserObject();
 System.out.println(emp.getApellidos() + ", " + emp.getNombre() +
 " [" + emp.getPuesto() + "]");

 System.out.println("Salario: " + emp.getSalario() + "€");
 System.out.println("Fecha contratación: " + new SimpleDateFormat("dd/MM/yyyy").format(emp.getFechaContratacion()));
 if (!emp.getObservaciones().isEmpty()) {
 System.out.println("Observaciones: " + emp.getObservaciones());
 }
 System.out.println("-----");
 }
 }
 });
 }

 public static void main(String[] args) {
 PruebaJTree p = new PruebaJTree();
 }
}
```

Al seleccionar un empleado (y sólo en este caso):

López Marin, Juan [Dirección]

#### Modificar el aspecto del árbol

La idea es la de siempre. Añadir un renderizador (`TreeCellRenderer`) que retorne el componente a mostrar en cada caso:

```
public class PruebaJTree extends JFrame {
 private JTree arbol;
 private List<Empleado> listaEmpleados;
 private Categoria[] categorias = {
 ...
 };

 public PruebaJTree() {
 ...
 }

 private void inicializarEmpleados() {
 ...
 }

 private void crearArbol() {
 ...
 arbol.setCellRenderer(new Render());
 }

 private void configurarSeleccion() {
 ...
 }

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

```

 PruebaJTree p = new PruebaJTree();
 }

}

class Render implements TreeCellRenderer {
 @Override
 public Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
 boolean expanded, boolean leaf, int row, boolean hasFocus) {
 DefaultMutableTreeNode dtm = (DefaultMutableTreeNode) value;
 JLabel lbl = new JLabel(dtm.getUserObject().toString());
 lbl.setOpaque(true);

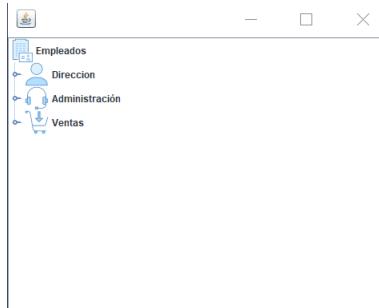
 if (row != 0) {
 if (dtm.getUserObject() instanceof Categoría) {
 lbl.setIcon(((Categoría) dtm.getUserObject()).getLogo());
 } else if (dtm.getUserObject() instanceof Empleado) {
 try {
 ImageIcon im = new ImageIcon(new URL(((Empleado) dtm.getUserObject()).getFoto()));
 Image imag = im.getImage().getScaledInstance(40, 40, Image.SCALE_SMOOTH);
 im.setImage(imag);
 lbl.setIcon(im);
 lbl.setBorder(new EmptyBorder(2, 0, 2, 0));
 } catch (Exception ex) {
 Logger.getLogger(PruebaJTree.class.getName()).log(Level.SEVERE, null, ex);
 }
 }
 } else {
 // Icono para el nodo raíz - asumimos que existe este recurso
 try {
 lbl.setIcon(new ImageIcon("empleados.png"));
 } catch (Exception e) {
 // Si no existe el recurso, no mostramos icono
 System.out.println("No se pudo cargar el icono para el nodo raíz");
 }
 }

 if (selected) {
 lbl.setBackground(new Color(116, 167, 250));
 } else {
 lbl.setBackground(Color.WHITE);
 }

 return lbl;
 }
}

```

La salida:



### Árboles Modificables en Java Swing (JTree)

Un `JTree` puede ser modificado dinámicamente para añadir, eliminar o cambiar nodos. Para ello, necesitamos trabajar con un modelo de árbol que permita estas operaciones, como `DefaultTreeModel` junto con nodos mutables (`DefaultMutableTreeNode`).

#### Creación de un árbol modificable

Para crear un árbol modificable, seguiremos estos pasos:

1. Crear un modelo de árbol (`DefaultTreeModel`) con un nodo raíz
2. Usar nodos mutables (`DefaultMutableTreeNode`) para permitir cambios
3. Implementar métodos para añadir, eliminar y editar nodos

#### Ejemplo básico

```

import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class ArbolModificable extends JFrame {
 private JTree arbol;
 private DefaultTreeModel modelo;
 private DefaultMutableTreeNode raiz;

 public ArbolModificable() {
 // Configuración básica de la ventana
 setTitle("Árbol Modificable");
 }
}

```

```

setSize(400, 500);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Crear el árbol con un nodo raíz
raiz = new DefaultMutableTreeNode("Categorías");
modelo = new DefaultTreeModel(raiz);
arbol = new JTree(modelo);

// Añadir algunos nodos iniciales
DefaultMutableTreeNode deportes = new DefaultMutableTreeNode("Deportes");
DefaultMutableTreeNode musica = new DefaultMutableTreeNode("Música");
DefaultMutableTreeNode cine = new DefaultMutableTreeNode("Cine");

modelo.insertNodeInto(deportes, raiz, 0);
modelo.insertNodeInto(musica, raiz, 1);
modelo.insertNodeInto(cine, raiz, 2);

// Añadir algunos hijos a los nodos
modelo.insertNodeInto(new DefaultMutableTreeNode("Fútbol"), deportes, 0);
modelo.insertNodeInto(new DefaultMutableTreeNode("Baloncesto"), deportes, 1);
modelo.insertNodeInto(new DefaultMutableTreeNode("Rock"), musica, 0);
modelo.insertNodeInto(new DefaultMutableTreeNode("Pop"), musica, 1);
modelo.insertNodeInto(new DefaultMutableTreeNode("Acción"), cine, 0);
modelo.insertNodeInto(new DefaultMutableTreeNode("Comedia"), cine, 1);

// Expandir el árbol para ver todos los nodos
expandirArbol();

// Crear panel de botones para modificar el árbol
 JPanel panelBotones = new JPanel();
 JButton btnAnadir = new JButton("Añadir");
 JButton btnEliminar = new JButton("Eliminar");
 JButton btnEditar = new JButton("Editar");

panelBotones.add(btnAnadir);
panelBotones.add(btnEliminar);
panelBotones.add(btnEditar);

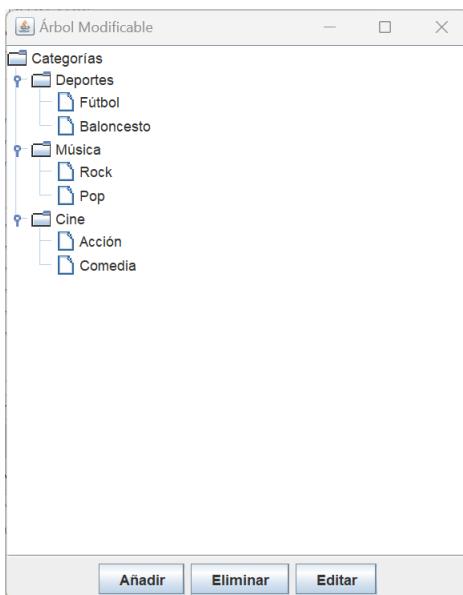
// Añadir componentes a la ventana
add(new JScrollPane(arbol), BorderLayout.CENTER);
add(panelBotones, BorderLayout.SOUTH);

setLocationRelativeTo(null);
setVisible(true);
}

// Método para expandir todos los nodos del árbol
private void expandirArbol() {
 for (int i = 0; i < arbol.getRowCount(); i++) {
 arbol.expandRow(i);
 }
}
}

```

La salida:



#### Añadir nodos

Para añadir un nodo podemos emplear el método:

```
modelo.insertNodeInto(nuevoNodo, nodoPadre, posicion);
```

El proceso sería el siguiente:

1. Crear un nuevo `DefaultMutableTreeNode` con el contenido deseado
2. Usar `insertNodeInto` del modelo, especificando el nodo padre y la posición
3. Opcionalmente, hacer visible el nuevo nodo con `scrollPathToVisible`

El código:

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package org.zabalburu.weatherapp;

import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class ArbolModificable extends JFrame {
 private JTree arbol;
 private DefaultTreeModel modelo;
 private DefaultMutableTreeNode raiz;

 public ArbolModificable() {
 ...
 panelBotones.add(btnAnadir);
 panelBotones.add(btnEliminar);
 panelBotones.add(btnEditar);

 // Añadir acción al botón Añadir
 btnAnadir.addActionListener((e) -> anadirNodo());
 ...
 }

 // Método para expandir todos los nodos del árbol
 private void expandirArbol() {
 ...
 }

 // Método para añadir un nuevo nodo
 private void anadirNodo() {
 // Obtener el nodo seleccionado
 DefaultMutableTreeNode nodoSeleccionado =
 (DefaultMutableTreeNode) arbol.getLastSelectedPathComponent();

 // Si no hay nodo seleccionado, usar la raíz
 if (nodoSeleccionado == null) {
 nodoSeleccionado = raiz;
 }

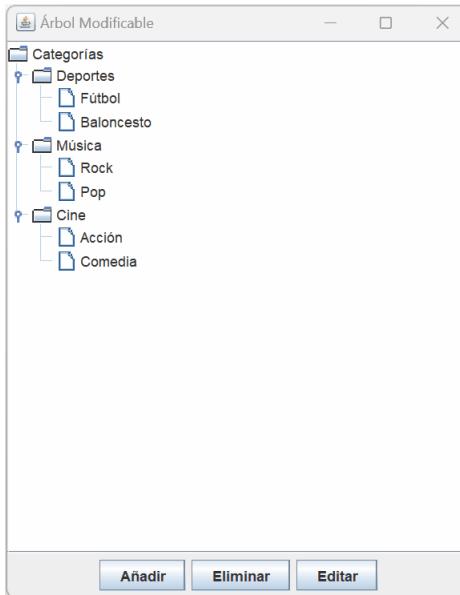
 // Pedir el nombre del nuevo nodo
 String nuevoNombre = JOptionPane.showInputDialog(
 this,
 "Nombre del nuevo elemento:",
 "Añadir elemento",
 JOptionPane.QUESTION_MESSAGE
);

 // Si se proporcionó un nombre, añadir el nodo
 if (nuevoNombre != null && !nuevoNombre.trim().isEmpty()) {
 DefaultMutableTreeNode nuevoNodo = new DefaultMutableTreeNode(nuevoNombre);
 modelo.insertNodeInto(nuevoNodo, nodoSeleccionado, nodoSeleccionado.getChildCount());

 // Hacer visible el nuevo nodo
 TreePath path = new TreePath(nuevoNodo.getPath());
 arbol.scrollPathToVisible(path);
 arbol.setSelectionPath(path);
 }
 }

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

La salida:



### 3. Eliminar nodos

```
// Eliminar un nodo
modelo.removeNodeFromParent(nodoAEliminar);
```

Para eliminar un nodo:

1. Obtener el nodo a eliminar (normalmente el seleccionado)
2. Usar `removeNodeFromParent` del modelo para eliminarlo
3. El nodo raíz no debe eliminarse

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package org.zabalburu.weatherapp;

import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class ArbolModificable extends JFrame {
 private JTree arbol;
 private DefaultTreeModel modelo;
 private DefaultMutableTreeNode raiz;

 public ArbolModificable() {
 ...

 // Añadir acción al botón Eliminar
 btnEliminar.addActionListener(e -> eliminarNodo());
 ...
 }

 // Método para expandir todos los nodos del árbol
 private void expandirArbol() {
 ...
 }

 // Método para eliminar un nodo
 private void eliminarNodo() {
 DefaultMutableTreeNode nodoSeleccionado =
 (DefaultMutableTreeNode) arbol.getLastSelectedPathComponent();

 if (nodoSeleccionado != null && nodoSeleccionado != raiz) {
 // Confirmar eliminación
 int respuesta = JOptionPane.showConfirmDialog(
 this,
 "¿Está seguro de eliminar '" + nodoSeleccionado.getUserObject() + "'?",
 "Confirmar eliminación",
 JOptionPane.YES_NO_OPTION
);

 if (respuesta == JOptionPane.YES_OPTION) {
 modelo.removeNodeFromParent(nodoSeleccionado);
 }
 } else {
 JOptionPane.showMessageDialog(
 this,
 "Seleccione un nodo para eliminar (no puede eliminar la raíz)",
 "Error",
 JOptionPane.ERROR_MESSAGE
);
 }
 }
}
```

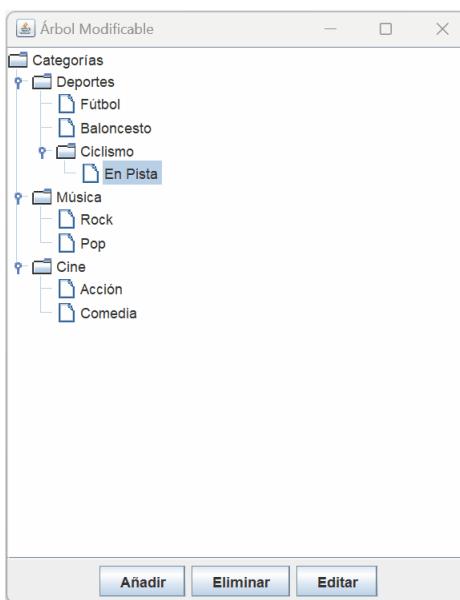
```

 JOptionPane.ERROR_MESSAGE
);
}
}

public static void main(String[] args) {
 ...
}
}

```

La salida:



#### 4. Editar nodos

```
// Editar el contenido de un nodo
nodoSeleccionado.setUserObject(nuevoValor);
modelo.nodeChanged(nodoSeleccionado);
```

Para editar un nodo:

1. Obtener el nodo a editar
2. Cambiar su contenido con `setUserObject`
3. Notificar al modelo del cambio con `nodeChanged`

```
package org.zabalburu.weatherapp;

import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import java.awt.event.*;

public class ArbolModificable extends JFrame {
 private JTree arbol;
 private DefaultTreeModel modelo;
 private DefaultMutableTreeNode raiz;

 public ArbolModificable() {
 ...

 // Añadir acción al botón Eliminar
 btnEliminar.addActionListener(e -> editarNodo());
 ...

 }

 // Método para expandir todos los nodos del árbol
 private void expandirArbol() {
 ...
 }

 // Método para editar un nodo
 private void editarNodo() {
 DefaultMutableTreeNode nodoSeleccionado =
 (DefaultMutableTreeNode) arbol.getLastSelectedPathComponent();

 if (nodoSeleccionado != null) {
 String nombreActual = nodoSeleccionado.getUserObject().toString();
 String nuevoNombre = JOptionPane.showInputDialog(
 this,
 "Editar nombre:",
 nombreActual
);
 }
 }
}
```

```

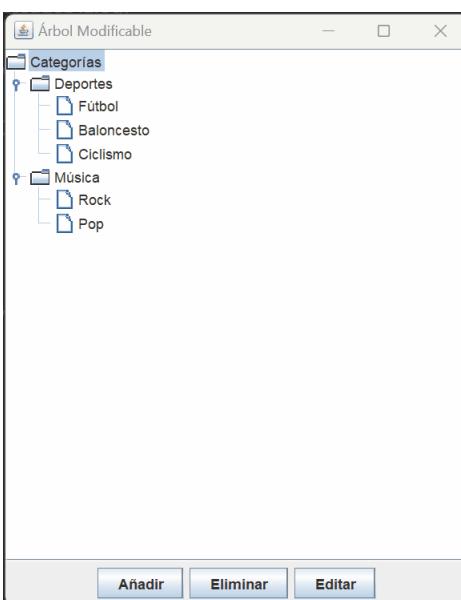
);
}

if (nuevoNombre != null && !nuevoNombre.trim().isEmpty()) {
 nodoSeleccionado.setUserObject(nuevoNombre);
 modelo.nodeChanged(nodoSeleccionado);
}
} else {
 JOptionPane.showMessageDialog(
 this,
 "Seleccione un nodo para editar",
 "Error",
 JOptionPane.ERROR_MESSAGE
);
}
}

public static void main(String[] args) {
 ...
}
}

```

La salida:



**NOTA:** Evidentemente en el ejemplo sólo estamos trabajando con los **nodos** del árbol. En una aplicación real habría que añadir, editar y eliminar además los datos del repositorio correspondiente

#### Expandir y contraer nodos

```

// Expandir todos los nodos
for (int i = 0; i < arbol.getRowCount(); i++) {
 arbol.expandRow(i);
}

// Expandir un nodo específico
TreePath path = new TreePath(nodo.getPath());
arbol.expandPath(path);

// Contraer un nodo
arbol.collapsePath(path);

```

#### Escuchar cambios en el árbol

```

modelo.addTreeModelListener(new TreeModelListener() {
 @Override
 public void treeNodesChanged(TreeModelEvent e) {
 System.out.println("Nodo modificado");
 }

 @Override
 public void treeNodesInserted(TreeModelEvent e) {
 System.out.println("Nodo añadido");
 }

 @Override
 public void treeNodesRemoved(TreeModelEvent e) {
 System.out.println("Nodo eliminado");
 }

 @Override
 public void treeStructureChanged(TreeModelEvent e) {
 System.out.println("Estructura modificada");
 }
});

```

```
});
```

## 5.28. Actividad 32 - Creación Formulario Complejo

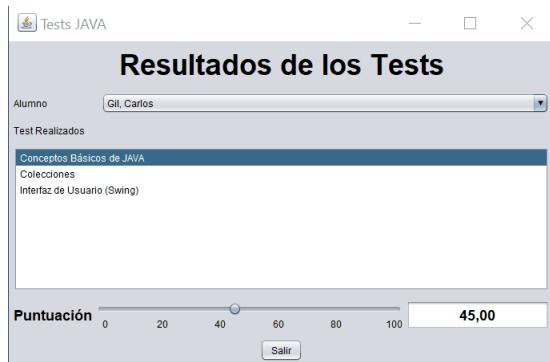
### Instrucciones

#### Classroom

Se pide crear una aplicación que permita **valorar los resultados de una serie de alumnos en unos tests**.

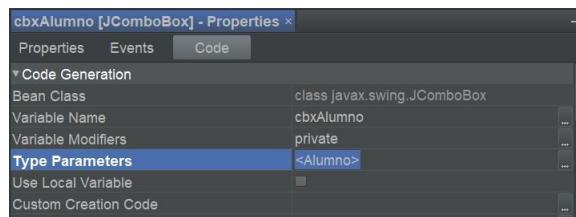
1. Definir las siguientes clases (en el paquete `tests.modelo`):
  2. **Alumno**: id, apellidos, nombre
  3. **Test**: id, titulo
  4. **AlumnoTest**: idAlumno, idTest, resultado con sus constructores, métodos set/get adecuados y, al menos, un método equals basado en el id.
5. Crear (en el paquete `tests.dao`) los siguientes interfaces con los métodos indicados:
  6. AlumnoDAO:
    7. List<Alumno> getAlumnos()
    8. Alumno getAlumno(int id)
    9. TestDAO:
      10. List<Test> getTests()
      11. Test getTest(int id)
    12. AlumnoTestDAO:
      13. List<AlumnoTest> getAlumnosTests()
      14. AlumnoTest getAlumnoTest(int idAlumno, int idTest)
  15. Definir la clase **AlumnoList** (también en el paquete `tests.dao`) que implementará `AlumnoDAO` y
  16. Definirá una propiedad de tipo `ArrayList<Alumno>` para almacenar los alumnos
  17. En el constructor, **añadirá alumnos de prueba** a dicha lista
  18. Definirá los métodos del interfaz para que funcionen sobre la lista indicada:
    19. `getAlumnos()`: Retornará la lista de alumnos. ADICIONAL: Hacer que el método `getAlumnos()` retorne los alumnos ordenados alfabéticamente por sus apellidos.
    20. `getAlumno(id)`: Buscará el alumno de la lista con ese id. Si existe, lo retornará y si no, retornará null
    21. Crear, de manera similar, la clase **TestList** (en este caso con un `ArrayList<Test>`)
    22. Definir la clase **AlumnoTestList** en la cual:
      23. Se creará un `ArrayList<AlumnoTest>` de objetos `AlumnoTest`
      24. En el constructor se añadirán datos a dicha lista (de alumnos y de tests que existan en las clases `AlumnoList` y `TestList`)
      25. El método `getAlumnosTests()` retornará dicha lista
      26. El método `getAlumnoTest(idAlumno, idTest)` retornará el objeto `AlumnoTest` con ese `idAlumno` e `idTest` si existe y, si no existe, retornará un valor null.
  6. Definir la clase **Servicio** en el paquete `test.servicio`. En esta clase:
    - Haremos que la clase sea un **singleton**. Es decir que todas las clases que usen el servicio empleen el mismo (y único) objeto de tipo `Servicio`. Para ello:
      - Se definirá una propiedad estática de tipo `Servicio` a la que se asignará un nuevo servicio (`private static Servicio servicio = new Servicio()`)
      - Se creará un constructor **PRIVADO** que **no hará nada** (de este modo nadie más puede crear objetos de tipo `Servicio`)
      - Se definirá un método estático llamado `getServicio` que retornará el servicio creado. De este modo, las clases que quieran hacer uso del servicio emplearán el método `Servicio.getServicio()` que retornará siempre el mismo servicio.
      - Se definirá una propiedad de tipo `AlumnoDAO` y se le asignará un nuevo objeto de tipo `AlumnoList`
      - Se definirá una propiedad de tipo `TestDAO` y se le asignará un nuevo objeto de tipo `TestList`
      - Se definirá una propiedad de tipo `AlumnoTestDAO` y se le asignará un nuevo objeto de tipo `AlumnoTestList`
      - Se definirán los siguientes métodos:
        - `List<Alumno> getAlumnos()` : Retornará la lista de alumnos desde la propiedad `AlumnoDAO`
        - `Alumno getAlumno(idAlumno)` : Retornará el alumno con ese `idAlumno` desde el método correspondiente de la propiedad `AlumnoDAO`
        - `List<Test> getTests()` : Retornará la lista de los tests desde la propiedad `TestDAO`
        - `Test getTest(idTest)` : Retornará el test con el `idTest` indicado desde el método correspondiente de la propiedad `TestDAO`

- `AlumnoTest getAlumnoTest (idAlumno, idTest)` : Retornará el `AlumnoTest` con ese `idAlumno` y de `test` mediante el correspondiente método de la propiedad `AlumnoTestDAO`.
- `List<Test> getTestsAlumno (idAlumno)`: Este método:
  - Definirá un `ArrayList<Test>` de objetos de tipo `Test` vacío
  - Obtendrá todos los objetos `AlumnoTest` del objeto `AlumnoTestDAO`
  - Recorrerá dicha lista y, con cada objeto `AlumnoTest` de la misma:
  - Si se `idAlumno` es igual al que nos han pasado:
    - Buscará el test con el `idTest` del objeto `AlumnoTest` en el que estamos
    - Añadirá el test encontrado a la lista
- Retornará dicha lista
- Crear un formulario similar al siguiente:



Se deberá tener en cuenta lo siguiente:

- La lista desplegable será un `JComboBox` de objetos de tipo `Alumno`, lo que implica que, si estás empleando el diseñador de Swing, hay que modificar la propiedad `TypeParameter` en la pestaña `Code` de las propiedades del `ComboBox` para que emplee `<Alumno>` en lugar de `<String>`:
- 



- Hay que eliminar la información de la propiedad `model` del mismo objeto dado que los datos los asignaremos por código.
- Algo similar hay que hacer con la lista de tests. En este caso `TypeParameter` será de tipo `<Test>` dado que vamos a almacenar objetos de tipo `Test` en la lista (también hay que eliminar el contenido de la propiedad `model`)
- **En la lista de test** sólo se podrá seleccionar un test
- La ventana deberá tener una propiedad de tipo `Servicio` para poder acceder a los datos de la aplicación
- En el constructor de la ventana:
- Tras `initComponents` se llamará a un método `cargarAlumnos()` que:
  - Por cada alumno de la aplicación (método `getAlumnos()` del servicio)
  - Lo añadirá a la lista desplegable (método `addItem()` del `JComboBox`).
  - NOTA: Recordar que, por defecto, el `JComboBox` muestra lo que retorna el método `toString()` de los objetos que almacena (en este caso `Alumno`)

[✓] Si ejecutamos la aplicación deberíamos ver todos los alumnos en la lista

- Añadir al `JComboBox` el receptor de eventos que permite saber que se ha seleccionado un nuevo alumno (`ActionEvent` - ver apuntes `JComboBox`). Cuando esto sucede vamos a añadir los tests realizados por el alumno a la lista de test. Para ello (ver apuntes `JList`):
- Creamos un modelo de datos para la lista creando un objeto vacío de tipo `DefaultListModel<AlumnoTest>`
- Obtenemos el alumno actualmente seleccionado (ver apuntes `JComboBox`)
- Si no hay ningún alumno seleccionado (el alumno es `null`) retornaremos sin más.
- Obtenemos todos los tests que ha hecho el alumno como objetos `AlumnoTest` llamando al método `getTestsAlumno()` pasándole el id del alumno seleccionado y, por cada uno de ellos:
  - Lo añadimos al modelo (`addElement()`)
  - Asignamos el modelo de datos a la lista (`setModel()`)
  - Seleccionamos el primer elemento de la lista (ver apuntes `JList`)

[✓] Si ejecutamos la aplicación deberíamos ver todos los test que ha hecho cada alumno cuando lo seleccionamos en la lista desplegable

- El siguiente paso es detectar cuando se ha seleccionado un test de la lista para mostrar su puntuación (ver apuntes `JList`). Dado que el evento que tenemos que controlar se asigna al modelo de selección de la lista y no a la propia lista, los siguientes pasos **NO SE PUEDEN HACER DESDE EL DISEÑADOR**. Hay que hacerlos por código y debemos situar dicho código **DETRÁS** de `initComponents()` y **ANTES** de `cargarAlumnos()`. Los pasos son:
- Obtener el modelo de selección de la lista (`getSelectionModel()`) y almacenarlo en una variable de tipo `ListSelectionModel`
- Añadir a dicha variable un objeto de una clase anónima que implemente `ListSelectionListener` (`addListSelectionListener()`) que, en el método `valueChanged()`, llame a un nuevo método en el que :
  - Obtengamos el alumno seleccionado en la lista desplegable
  - Obtengamos el test seleccionado en la lista de tests
  - Si no hay ningún test seleccionado (es `null`) retornar
  - Si lo hay, busque el objeto `AlumnoTest` (método `getAlumnoTest()` del servicio) con el id del alumno y del test seleccionados
  - Actualice el `JSlider` y el `JFormattedTextField` con la puntuación de dicho objeto.

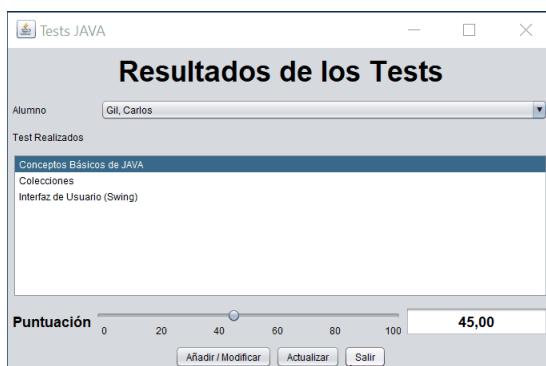
[✓] Si ejecutamos la aplicación deberíamos ver que, al seleccionar un test de un alumno nos aparece su puntuación en los dos sitios

- Añadir un evento de acción al botón `Salir` para que, al pulsarlo, finalice la aplicación.

#### Reto

Vamos a añadir a la aplicación la posibilidad de añadir / modificar la nota de los tests. Para ello vamos a añadir a la clase `AlumnoTestList` el método:

- `modificarTestAlumno(Integer idAlumno, Integer idTest, Integer puntuación)`: Este método comprobará si existe el objeto `AlumnoTest` para ese alumno y test.
- Si **existe**, modificará su puntuación con la recibida en el método
- Si **no existe** (el alumno no tiene la nota para ese test), añadirá un **nuevo** objeto `AlumnoTest` con la información recibida
- Añadiremos un método similar al servicio que llame al método recién creado. El funcionamiento final de la aplicación será similar al siguiente:



En el formulario de `Añadir / Modificar Test`:

- Al seleccionar cualquiera de las dos listas desplegables se llamará al mismo método en el que:
- Se obtendrá el alumno y el test seleccionado
- Si alguno de ellos es `null`, se retornará
- Se buscará el objeto `AlumnoTest` de ese alumno / test con el método correspondiente del servicio
  - Si **es nulo** (no hay nota para ese alumno y test), se asignará `null` al `JFormattedTextField` de la puntuación
  - Si **no lo es**, se asignará la puntuación del objeto al `JFormattedTextField`
- Al pulsar el botón de `Guardar` se llamará al método creado en el paso anterior del servicio con los datos del formulario
- El botón `Volver`, hará un `dispose()` del formulario

En el formulario principal

- El botón `Añadir / Modificar` creará un nuevo formulario de añadir y lo mostrará
- El botón `Actualizar`:
- Guardará la posición del alumno seleccionado actualmente (`getSelectedIndex()`)
- Seleccionará el elemento que está en la posición `-1` (no seleccionamos nada)
- Volverá a seleccionar el alumno de la posición guardada previamente

## 6. Acceso a Bases de Datos

### 6.1. Introducción

#### Java Data Base Connectivity (JDBC)

JDBC es un conjunto de interfaces y clases que proporciona acceso a bases de datos desde Java. Para interactuar con la BBDD concreta se emplean drivers proporcionados por el fabricante y encargados de traducir sentencias SQL a la sintaxis concreta del RDBMS.

De forma similar a las colecciones, la mayor parte de JDBC son interfaces y, son los fabricantes de las BBDD los que diseñan las clases que implementan dichos interfaces para trabajar con ellas. Es por esto que, para poder utilizar una determinada BBDD, necesitaremos una serie de clases que obtendremos del fabricante de la misma.

Lo primero que necesitamos es obtener una Conexión a la BBDD. La conexión tiene que ser un objeto que implemente el interfaz `java.sql.Connection` (un objeto `com.microsoft.sqlserver.jdbc.SQLServerConnection` en el caso de SQLServer o un objeto `oracle.jdbc.driver.OracleConnection` en el caso de Oracle).

Lo primero es obtener las librerías JDBC para Oracle y/o para SQLServer.

##### Conexión mediante DriverManager

En este caso, lo primero es cargar el driver de la BBDD mediante el método `Class.forName("driver")` y, tras ello, emplear la clase `DriverManager` para obtener la conexión. El driver es una clase que implementa `java.sql.Driver` y es `oracle.jdbc.driver.OracleDriver` para Oracle y `com.microsoft.sqlserver.jdbc.SQLServerDriver` para SQLServer.

Una vez cargado(s) el(los) driver(s), la clase `java.sql.DriverManager` obtiene una conexión con la BBDD adecuada a través de una URL específica de la BBDD.

- En el caso de Oracle es `jdbc:oracle:thin:@servidor:puerto:SID` donde servidor es el equipo donde está el Oracle, puerto es el puerto en el que escucha y SID el identificador de la BBDD
- En el caso de SQLServer es `jdbc:sqlserver://servidor:puerto;databaseName=bbdd` donde bbdd es el nombre de la BBDD y servidor y puerto el equipo y puerto en el que está

Un ejemplo:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class PuebaConexion {
 public static void main(String[] args) {
 Connection cnnSQLServer = null;
 Connection cnnOracle = null;
 try {
 Class.forName("oracle.jdbc.driver.OracleDriver");
 Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
 cnnSQLServer = DriverManager.getConnection("jdbc:sqlserver://localhost:1234;databaseName=pubs","sa","tiger");
 cnnOracle = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl","sa","tiger");
 System.out.println("Conectado!!!");
 } catch (ClassNotFoundException ex) {
 Logger.getLogger(PuebaConexion.class.getName()).log(Level.SEVERE, null, ex);
 } catch (SQLException ex) {
 Logger.getLogger(PuebaConexion.class.getName()).log(Level.SEVERE, null, ex);
 }
 }
}
```

##### Conexión mediante DataSource (método preferido)

La forma recomendada para establecer una conexión con una BBDD es mediante un objeto de una clase que implemente el interfaz `javax.sql.DataSource`. Cada fabricante proporciona su propia implementación de dicho interface (por ejemplo, `oracle.jdbc.pool.OracleDataSource` para Oracle o `com.microsoft.sqlserver.jdbc.SQLServerDataSource` para SQLServer)

En este caso, cada `DataSource` dispone de métodos específicos para cada uno de los elementos necesarios para la conexión:

```
import com.microsoft.sqlserver.jdbc.SQLServerDataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
import oracle.jdbc.pool.OracleDataSource;

public class PuebaConexion {

 public static void main(String[] args) {
 Connection cnnSQLServer = null;
 Connection cnnOracle = null;
 try {
 SQLServerDataSource dsSQLServer = new SQLServerDataSource();
 OracleDataSource dsOracle = new OracleDataSource();

 dsSQLServer.setUser("sa");
 dsSQLServer.setPassword("tiger");
 dsSQLServer.setServerName("localhost");
 dsSQLServer.setPortNumber(1234);
 dsSQLServer.setDatabaseName("Northwind");
 cnnSQLServer = dsSQLServer.getConnection();
 }
 }
}
```

```

 dsOracle.setDriverType("thin");
 dsOracle.setServerName("localhost");
 dsOracle.setDatabaseName("ORCL");
 dsOracle.setPortNumber(1521);
 dsOracle.setUser("sa");
 dsOracle.setPassword("tiger");
 cnnOracle = dsOracle.getConnection();

 System.out.println("Conectado!");
 } catch (SQLException ex) {
 Logger.getLogger(PuebaConexion.class.getName()).log(Level.SEVERE, null, ex);
 }
}
}

```

#### HelperClass

Dado que, en una aplicación normalmente basta con establecer una única conexión con la BBDD, una buena idea es crear una clase auxiliar que implemente el patrón Singleton y nos proporcione dicha conexión. Por ejemplo, para Oracle y empleando un DataSource podría ser:

```

import java.sql.Connection;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
import oracle.jdbc.pool.OracleDataSource;

public class Conexion {
 private static Connection cnn = null;

 private Conexion() {}

 public static Connection getConexion() {
 if (cnn==null){
 try {
 OracleDataSource dsOracle = new OracleDataSource();
 dsOracle.setDriverType("thin");
 dsOracle.setServerName("localhost");
 dsOracle.setDatabaseName("ORCL");
 dsOracle.setPortNumber(1521);
 dsOracle.setUser("sa");
 dsOracle.setPassword("tiger");
 cnn = dsOracle.getConnection();
 } catch (SQLException ex) {
 Logger.getLogger(Conexion.class.getName()).log(Level.SEVERE, null, ex);
 }
 }
 return cnn;
 }

 public static void cerrarConexion(){
 if (cnn != null){
 try {
 cnn.close();
 } catch (SQLException ex) {}
 cnn = null;
 }
 }
}

```

Cuando se necesite una conexión con la BBDD basta con ejecutar `Conexion.getConexion()`. Si la conexión ya existe, se reutiliza y si no existe se crea una nueva en ese momento.

#### 6.2. Ejecución de Consultas DDL y DML

##### Consultas en JDBC

Una vez que tenemos una conexión establecida con la BBDD podemos empezar a hacer consultas sobre las mismas. Para cualquier tipo de consulta necesitamos obtener un objeto que implemente el interfaz `java.sql.Statement` que es el que nos permite lanzar consultas contra la BBDD a través de la conexión.

Podemos obtener un objeto `Statement` mediante el comando `createStatement()` del objeto conexión.

Una vez obtenida la conexión, podemos ejecutar consultas de manipulación de datos (DDL y DML excepto SELECT) mediante el método `executeUpdate(sql)` de dicho objeto. Si la consulta es una SELECT entonces emplearemos el método `executeQuery(sql)` y, si no sabemos de qué tipo es, podemos emplear `execute(sql)`.

La diferencia es que, dado que las consultas no SELECT no devuelven resultados, el método `executeUpdate` retorna un entero (que debería indicar el resultado de la ejecución de la consulta aunque cada implementación en cada BBDD decide qué retornar), mientras que las SELECT devuelven información por lo que el método `executeQuery` retornará un objeto `java.sql.ResultSet` que nos va a permitir acceder a dicha información (lo veremos más adelante).

**NOTA:** Es recomendable cerrar todos los objetos creados en la aplicación (`Connection, Statement, ResultSet...`). Toda operación de BBDD puede lanzar una excepción de tipo `java.sql.SQLException` que deberá ser controlada.

##### Consultas NO SELECT

Como ya hemos comentado, el método `executeUpdate` nos permite ejecutar consultas NO SELECT, es decir consultas de creación, modificación y eliminación (CREATE, ALTER, DROP) de objetos (tablas, vistas, usuarios...) y consultas DML de creación, modificación y eliminación de registros (INSERT, UPDATE y DELETE). Las consultas se enviarán a la BBDD para su ejecución, así que habrá que tener en cuenta las peculiaridades de cada una de ellas y las sentencias y tipos de objetos disponibles.

Como se ha comentado, el método `executeUpdate` retorna un entero pero, en JAVA, no se especifica qué valor debe retornar. Lo habitual es que, en consultas DDL, retorne un valor indicando si la consulta ha tenido éxito o no y en consultas DML retorne el número de registros afectados por la operación.

Veamos un ejemplo en el que vamos a crear unas tablas para gestionar los conocimientos en lenguajes de programación de una serie de programadores. Vamos a crear las tablas PROGRAMADOR, LENGUAJE y una tabla intermedia llamada PROGRAMADORENGUAJE:

#### Ejemplo para SQLServer

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class CreacionTablasSQLServer {

 public static void main(String[] args) {
 try {
 Connection cnn = Conexion.getConexion();
 Statement stmt = cnn.createStatement();
 // Eliminamos las tablas si existen (si no, dará una excepción pero seguimos adelante)
 // Primero la tabla referenciada (no se puede eliminar una tabla mientras referencia a otra)
 try {
 stmt.executeQuery("DROP TABLE PROGRAMADORENGUAJE");
 } catch (SQLException ex) {}
 try {
 stmt.executeQuery("DROP TABLE PROGRAMADOR");
 } catch (SQLException ex) {}
 try {
 stmt.executeQuery("DROP TABLE LENGUAJE");
 } catch (SQLException ex) {}
 // Tabla PROGRAMADOR
 stmt.executeUpdate("CREATE TABLE PROGRAMADOR (" +
 "ID INTEGER PRIMARY KEY," +
 "APELLODOS VARCHAR(120)," +
 "NOMBRE VARCHAR(50)," +
 "FECHA_ALTA DATE)");
 // Tabla LENGUAJE
 stmt.executeUpdate("CREATE TABLE LENGUAJE (" +
 "ID INTEGER PRIMARY KEY," +
 "NOMBRE VARCHAR(255))");
 // Tabla PROGRAMADORENGUAJE
 stmt.executeUpdate("CREATE TABLE PROGRAMADORENGUAJE (" +
 "ID INTEGER PRIMARY KEY," +
 "IDPROGRAMADOR INTEGER REFERENCES PROGRAMADOR," +
 "IDLENGUAJE INTEGER REFERENCES LENGUAJE," +
 "NIVEL INTEGER");
 // Inserción de datos
 stmt.executeUpdate("INSERT INTO PROGRAMADOR VALUES(1,'López Martín', 'Ana', '2017-02-01')");
 stmt.executeUpdate("INSERT INTO PROGRAMADOR VALUES(2,'Ginés Montes', 'Luis', '2018-12-13')");
 stmt.executeUpdate("INSERT INTO LENGUAJE VALUES(1,'JAVA')");
 stmt.executeUpdate("INSERT INTO LENGUAJE VALUES(2,'C#')");
 stmt.executeUpdate("INSERT INTO LENGUAJE VALUES(3,'Javascript')");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(1,1,1,80)");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(2,1,2,50)");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(3,1,3,80)");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(4,2,3,85)");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(5,2,2,90)");
 stmt.close();
 Conexion.cerrarConexion();
 } catch (SQLException ex) {
 Logger.getLogger(PuebaConexion.class.getName()).log(Level.SEVERE, null, ex);
 }
 }
}
```

#### Ejemplo para Oracle

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class CreacionTablasSQLServer {

 public static void main(String[] args) {
 try {
 Connection cnn = Conexion.getConexion();
 Statement stmt = cnn.createStatement();
 // Eliminamos las tablas si existen (si no, dará una excepción pero seguimos adelante)
 // Primero la tabla referenciada (no se puede eliminar una tabla mientras referencia a otra)
 try {
 stmt.executeQuery("DROP TABLE PROGRAMADORENGUAJE");
 } catch (SQLException ex) {}
 try {
 stmt.executeQuery("DROP TABLE PROGRAMADOR");
 } catch (SQLException ex) {}
 try {
 stmt.executeQuery("DROP TABLE LENGUAJE");
 } catch (SQLException ex) {}
 // Tabla PROGRAMADOR
 stmt.executeUpdate("CREATE TABLE PROGRAMADOR (" +
 "ID INTEGER PRIMARY KEY," +
 "APELLODOS VARCHAR2(120)," +
 "NOMBRE VARCHAR2(50)," +
```

```

 "FECHA_ALTA DATE);
 // Tabla LENGUAJE
 stmt.executeUpdate("CREATE TABLE LENGUAJE (" +
 "ID INTEGER PRIMARY KEY," +
 "NOMBRE VARCHAR2(255))");
 // Tabla PROGRAMADORENGUAJE
 stmt.executeUpdate("CREATE TABLE PROGRAMADORENGUAJE (" +
 "ID INTEGER PRIMARY KEY," +
 "IDPROGRAMADOR INTEGER REFERENCES PROGRAMADOR," +
 "IDL LENGUAJE INTEGER REFERENCES LENGUAJE," +
 "NIVEL INTEGER)");
 // Inserción de datos
 stmt.executeUpdate("INSERT INTO PROGRAMADOR VALUES(1,'López Martín', 'Ana', DATE '2017-02-01')");
 stmt.executeUpdate("INSERT INTO PROGRAMADOR VALUES(2,'Ginés Montes', 'Luis', DATE '2018-12-13')");
 stmt.executeUpdate("INSERT INTO LENGUAJE VALUES(1,'JAVA')");
 stmt.executeUpdate("INSERT INTO LENGUAJE VALUES(2,'C#')");
 stmt.executeUpdate("INSERT INTO LENGUAJE VALUES(3,'Javascript')");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(1,1,1,80)");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(2,1,2,50)");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(3,1,3,80)");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(4,2,3,85)");
 stmt.executeUpdate("INSERT INTO PROGRAMADORENGUAJE VALUES(5,2,2,90)");
 stmt.close();
 Conexion.cerrarConexion();
} catch (SQLException ex) {
 Logger.getLogger(PueblaConexion.class.getName()).log(Level.SEVERE, null, ex);
}
}
}

```

### 6.3. Actividad 33 Creación de Tablas

#### Instrucciones

##### Classroom

A partir de la Actividad 28 (Funkos) hecha en la lección anterior **vamos a modificarla para que la información se almacene en una BBDD**. Para ello se pide:

- Añadir una clase `util.Conexion` que retorne una **conexión con la BBDD de Oracle**
- Añadir otra clase `util.CreacionTablas` que:
- Elimine la tabla para poder ejecutar la clase tantas veces como se quiera. Si la tabla no existe, el proceso deberá continuar y no dar errores
- Añada las sentencias necesarias para crear la tabla `Funko` con los campos indicados en el ejercicio (el campo `id` será la **clave primaria**).
- Lea los datos del fichero `funkos.dat` y los añada (mediante `INSERT`) a la BBDD
- Se deberán cerrar todos los objetos utilizados.
- Una vez hecho, comprobar que los datos se han guardado adecuadamente en la BBDD

### 6.4. Ejecución de Consultas de Selección

#### Consultas SELECT en JDBC

Los comandos que retoman resultados (SELECT), pueden ser ejecutados mediante el método `executeQuery(sentencia)` del interfaz.

Este método retorna un objeto de una clase que implementa el interfaz `java.sql.ResultSet` (`oracle.jdbc.Resultset` o `com.microsoft.sqlserver.jdbc.SQLServerResultSet`)

Un objeto `ResultSet` representa un conjunto de datos organizados en filas y columnas. Por defecto, el `ResultSet` retorna un conjunto de datos de sólo lectura y que sólo se pueden recorrer hacia adelante.

##### Recorrer un ResultSet

Cuando se ejecuta una consulta de selección y se obtiene un `ResultSet` el sistema se posiciona delante de la primera fila del mismo. Para leer una fila se debe emplear el método `next()` que almacena la siguiente fila en el `ResultSet` si existe. En el caso de que hayamos podido leer la fila (hay más datos), el método retorna un valor `true` y cuando no quedan más datos retornará un valor `false`. En este último caso el sistema queda posicionado detrás de la última fila.

Una vez tenemos una fila en el objeto, accederemos a los datos de la misma mediante alguno de los métodos `getTipo(nombreCampo)` o `getTipo(nºcolumna)`, en este último caso se empieza a contar las columnas desde 1. Existe un método de este tipo, por cada tipo de dato (`int, char, byte, float, double, long, String, Date, etc.`).

Además, disponemos de métodos auxiliares que nos permiten saber en qué posición estamos y el estado del `ResultSet` como se puede ver en la siguiente tabla:

Comando	<code>rst.getString("nombre")</code>	<code>rst.isClosed()</code>	<code>rst.isBeforeFirst()</code>	<code>rst.isAfterLast()</code>	<code>rst.getRow()</code>	retorna
<code>ResultSet rst = stmt.executeQuery("Select * From PROGRAMADOR");</code>	SQLException	false	true	false	0	
<code>rst.next();</code>	Ana	false	false	false	1	true
<code>rst.next();</code>	Luis	false	false	false	2	true
<code>rst.next();</code>	SQLException	false	false	true	0	false
<code>rst.close();</code>	SQLException	true	SQLException	SQLException	SQLException	

Por tanto, el código para recorrer todos los programadores de la BBDD sería:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ListadoProgramadores {

 public static void main(String[] args) {
 try {
 Connection cnn = Conexion.getConexion();
 Statement stmt = cnn.createStatement();
 ResultSet rst;
 rst = stmt.executeQuery("Select * From PROGRAMADOR Order By APELLIDOS, NOMBRE");
 System.out.println("Listado de Programadores");
 System.out.println("=====");
 while (rst.next()) {
 System.out.printf("%-30s%D%n",rst.getString("apellidos"), " "+rst.getString("nombre"),
 rst.getDate("fecha_alta"));
 }
 rst.close();
 stmt.close();
 Conexion.cerrarConexion();
 } catch (SQLException ex) {
 Logger.getLogger(PuebaConexion.class.getName()).log(Level.SEVERE, null, ex);
 }
 }
}
```

La salida:

```
Listado de Programadores
=====
Ginés Montes, Luis 12/13/18
López Martín, Ana 02/01/18
```

Veamos otro ejemplo en el que se muestra los conocimientos de un programador en diferentes lenguajes controlando los posibles errores:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class BuscarProgramador {

 public static void main(String[] args) {
 try {
 Connection cnn = Conexion.getConexion();
 Statement stmt = cnn.createStatement();
 ResultSet rst;
 String apellidos = JOptionPane.showInputDialog(null, "Apellidos");
 rst = stmt.executeQuery("Select * From PROGRAMADOR WHERE APELLIDOS='" + apellidos + "'");
 if (rst.next()) {
 String programador = rst.getString("apellidos") + ", " + rst.getString("nombre");
 rst = stmt.executeQuery("Select l.nombre, nivel "
 + "FROM LENGUAJE L, PROGRAMADORLENGUAJE PL "
 + "WHERE l.id = pl.idlenguaje and pl.idprogramador=" + rst.getInt("id"));
 System.out.println("PROGRAMADOR: " + programador);
 while (rst.next()){
 System.out.printf("\t%-20s%3d\n",rst.getString("nombre"), rst.getInt("nivel"));
 }
 } else {
 JOptionPane.showMessageDialog(null, "No hay ningún programador con ese apellido",
 "No Enccontrado", JOptionPane.WARNING_MESSAGE);
 }
 rst.close();
 stmt.close();
 Conexion.cerrarConexion();
 } catch (SQLException ex) {
 Logger.getLogger(PuebaConexion.class.getName()).log(Level.SEVERE, null, ex);
 }
 }
}
```

La salida, suponiendo que el programador existe:

```
PROGRAMADOR: Ginés Montes, Luis
Javascript 85
C# 90
```

## 6.5. Actividad 34 - ResultSets

### Instrucciones

#### [Classroom](#)

- Añadir a la actividad anterior una clase FunkoOracle que implemente FunkoDAO y que \*\*defina todos los métodos de dicho interfaz basándose en consultas a la

BBDD.

- Modificar la clase de servicio para que, en lugar de FunkoFich emplee FunkoOracle y comprobar que la aplicación funciona correctamente.

## 6.6. Sentencias Preparadas PreparedStatement

### Sentencias Preparadas en JDBC (PreparedStatement)

Las consultas que hemos realizado hasta el momento y que necesitan datos adicionales que tenemos almacenados en variables tienen varios problemas:

- Son complicadas de escribir. La expresión no es fácil de leer, hay que andar concatenando la misma con las variables donde está la información y, si hay parámetros de cadena o de fecha debemos añadir manualmente las comillas simples o los símbolos necesarios dentro de la expresión.
- Pueden tener problemas de compatibilidad con campos de tipo fecha. Dado que, cada BBDD gestiona los literales de fecha de una forma, no se puede emplear la misma consulta con parámetros de fecha para todos los casos. Eso nos obligaría a reescribir esas consultas si hacemos un cambio de BBDD.
- Son susceptibles a ataques de inyección de código SQL. Si pedimos al usuario un dato para ejecutar una consulta, ésta puede proporcionar un dato especialmente preparado para intentar modificar dicha consulta.

#### Ejemplo de inyección de código

Supongamos que queremos asegurar el acceso a nuestra BBDD pidiendo un nombre de usuario y una contraseña. Así que mostramos un formulario al usuario pidiendo ambos campos y escribimos el siguiente código para controlar el acceso a la información:

```
String sql = "Select * From Usuarios Where nombre=''' + nombre + ''' and password=''' + password + '''";
ResultSet rst = stmt.executeQuery(sql);
if (rst.next()){
 JOptionPane.showMessageDialog("Bienvenido!");
 ...
} else {
 JOptionPane.showMessageDialog("Usuario / contraseña erróneos");
}
```

¿Qué pasa si cuando le pedimos los datos al usuario escribimos lo siguiente?:

- nombre --> lo dejamos en blanco
- password --> ' OR 1='1 La consulta quedaría de la siguiente forma Select \* From Usuarios Where nombre='' and password='' OR 1='1' Si evaluamos la consulta veremos que siempre es verdadera por lo que entrariamos al sistema sin conocer los datos de ningún usuario.

#### PreparedStatement

Las sentencias preparadas solucionan todos estos problemas. Una sentencia preparada es una clase que implementa el interfaz `java.sql.PreparedStatement` (hijo de `Statement`) y permite especificar parámetros en una consulta SQL a través de marcadores de posición.

A diferencia de las sentencias normales, las sentencias preparadas se crean en el momento de crear el objeto:

```
PreparedStatement pst = cnn.prepareStatement(SQL);
```

Dentro de la instrucción SQL podemos emplear el marcador de posición (?) en aquellos lugares de la sentencia en los que vamos a añadir información dinámica (sólo se puede añadir información dinámica como valores de los parámetros, es decir, no se pueden emplear marcadores en una sentencia preparada para indicar el nombre de la tabla o el nombre de algún campo).

Una vez definidos los parámetros de la consulta y, antes de ejecutarla, se le deben asignar valores mediante:

```
pst.setType(posición, valor)
```

Donde hay un método para cada tipo de datos (`String, Date, int...`) y posición es la posición del marcador al que estamos asignando el valor.

Una vez asignados los valores ejecutamos la consulta con:

```
ResultSet rst = pst.executeQuery(); // SELECT
pst.executeUpdate(); // Resto
```

La instrucción preparada se encarga de evitar el problema de SQL Injection escapando los caracteres especiales. Por ejemplo, para el ataque anterior se generaría la siguiente sentencia SQL:

```
Select * From Usuarios where nombre=null and password='\' OR 1='1'
```

Con lo que no funcionaría el ataque dado que las comillas son escapadas en la sentencia.

#### Comparación entre Statement y PreparedStatement

Statement	PreparedStatement
Statement stmt = cnn.createStatement();	PreparedStatement pst = cnn.prepareStatement(SQL);
	pst.setTipo(1,valor); pst.setTipo(2,valor); ...
ResultSet rst = stmt.executeQuery(sql); o Integer rdo = stmt.executeUpdate(sql);	ResultSet rst = pst.executeQuery(); o Integer rdo = pst.executeUpdate();
stmt.close();	pst.close();

¿Cuándo usar uno u otro? En general, usaremos `PreparedStatement` cuando la consulta tenga datos variables y `Statement` cuando no.

En el siguiente ejemplo vamos a ver cómo podemos obtener el nivel en cada lenguaje de un Programador cuyos apellidos pedimos al usuario controlando los posibles errores:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class PuebaPreparedStatement {
 public static void main(String[] args) throws ClassNotFoundException {
 try {
 Connection cnn = Conexion.getConexion();
 PreparedStatement pstProgramador = cnn.prepareStatement(
 "Select * From Programador Where Apellidos=?");
 PreparedStatement pstLenguajes = cnn.prepareStatement(
 "Select L.nombre, PL.nivel From ProgramadorLenguaje PL, Lenguaje L "+
 "Where PL.idLenguaje = L.id and PL.idProgramador = ? " +
 "Order By L.nombre");
 String apellidos = JOptionPane.showInputDialog("Apellidos");
 pstProgramador.setString(1, apellidos);
 ResultSet rstProgramador = pstProgramador.executeQuery();
 if (rstProgramador.next()){
 pstLenguajes.setInt(1, rstProgramador.getInt("id"));
 ResultSet rstLenguajes = pstLenguajes.executeQuery();
 System.out.println(rstProgramador.getString("apellidos")+", "+
 rstProgramador.getString("nombre")+"\n\n");
 System.out.printf("%-30s%5s\n", "Lenguaje", "Nivel");
 System.out.print("%-30s%5s\n", "=====", "=====");
 while (rstLenguajes.next()){
 System.out.printf("%-30s%5d\n", rstLenguajes.getString("nombre"),
 rstLenguajes.getInt("nivel"));
 }
 rstProgramador.close();
 rstLenguajes.close();
 pstProgramador.close();
 pstLenguajes.close();
 Conexion.cerrarConexion();
 } else {
 JOptionPane.showMessageDialog(null, "No existe ningún programador con esos apellidos",
 "Error", JOptionPane.ERROR_MESSAGE);
 }
 Conexion.cerrarConexion();
 } catch (SQLException ex) {}
 }
}
```

La salida:

López Martín, Ana

Lenguaje	Nivel
=====	=====
C#	50
JAVA	80
Javascript	80

## 6.7. Actividad 35 - PreparedStatement

### Instrucciones

#### Classroom

Vamos a modificar la aplicación de `Pedidos` (Actividad 31) para que trabaje contra la BBDD de `Chinook` en Oracle. Para ello hay que crear una clase `AlbumArtistaPistaOracle` que implemente `AlbumArtistaPistaDAO` y que ejecute todos sus métodos contra la BBDD empleando sentencias preparadas. Tras ello habrá que modificar la clase `Servicio` para que en vez de emplear `AlbumArtistaPistaFich` emplee `AlbumArtistaPistaOracle`, es decir, sustituir `new AlbumArtistaPistaFich()` por `new AlbumArtistaPistaOracle()`. Al emplear interfaces, con este simple cambio la aplicación debería funcionar exactamente igual que antes aunque ahora los datos se cogerán de la BBDD.

## 6.8. Actividad 36

### Instrucciones

#### Classroom

Se desea diseñar una aplicación para llevar el control de las **oficinas** con las que trabajamos. Para ello se creará una clase `Oficina` que tendrá un campo por cada campo de la tabla `OFICINAS` y, además métodos `set/get` para todos los campos, constructor sin argumentos y otro que reciba todos los datos. Además, el método `toString` retornará la cadena `idOficina - ciudad [región]`. Además, se definirá una clase `Empleado` con los campos que se ven en la tabla, métodos `set/get` y constructores. En este caso, el método `toString` retornará simplemente el **nombre del empleado**. Para gestionar la aplicación, vamos a emplear el patrón DAO para lo cual se definirá una clase con los siguientes métodos:

- Método `getOficinas` que retornará una lista con todas las oficinas de la base de datos ordenadas en base al campo `idOficina`.
- Método `getDirector` que recibirá un `idOficina` y deberá retornar el `Empleado` que es el **director de dicha oficina**. Si la oficina no tiene director (el campo es un **cero**, retornará un valor **nulo**).

- Método `getEmpleados` que retornará una lista con todos los empleados de la oficina ordenados en base al `nombre`.
- Método `modificarOficina` que recibirá un objeto `Oficina` y modificará con sus datos la oficina correspondiente en la base de datos. El método no retornará nada.

La aplicación debe permitir consultar y modificar los datos de una oficina (excepto el idde oficina) y tendrá el siguiente (en modo consulta):



Los controles `Objetivo` y `Ventas` deben mostrar la información correctamente **formateada en castellano**. El cuadro desplegable de `Región` tendrá los valores `Norte`, `Sur`, `Este`, `Oeste` y `Centro`. La tabla mostrará todos los empleados de la oficina y los campos `cuenta` y `ventas` deberán mostrarse con **formato monetario**. Nuestra aplicación podrá estar en dos estados `CONSULTA` o `MODIFICACION` (se controlarán mediante constantes). El estado inicial será `CONSULTA`. Además, la aplicación dispondrá de una variable de instancia para almacenar la lista de todas las oficinas

Todos los accesos a los datos se harán a través de la clase `DAO` creada anteriormente. La aplicación :

- Al inicializar la ventana:
- Se almacenarán todas las oficinas, regiones y empleados en las listas desplegables correspondientes
- Se seleccionará el primer elemento de la lista de oficinas
- Se llamará al método `mostrar`
- El método `mostrar`:
- Asignará a los controles los valores correspondientes a la oficina seleccionada (en el caso del `director` y la `región` se seleccionarán los elementos correspondientes en base a la oficina).
- Permitirá o no editar los controles de la oficina en función del estado (el campo `código` nunca será editable).
- Habilitará / deshabilitará los botones de acuerdo al estado.
- Asignará los empleados de la oficina a la tabla
- Cada vez que se **seleccione una nueva oficina** de la lista desplegable se llamará al método `mostrar`
- El botón `modificar` cambiará a modo `MODIFICACION` y el botón `cancelar` a modo `CONSULTA`. Ambos llamarán al método `mostrar`
- El botón `guardar`:
- Comprobará que los campos `Ciudad` y `Dirección` tengan datos.
  - De no ser así, se dará un mensaje de **error** adecuado
  - Si hay datos, modificará la oficina actual con la información de los campos del formulario y, con dicha oficina, llamará al método `modificarOficina` de la clase de acceso a datos.
  - Tras ello, pasará a modo `CONSULTA` y llamará a `mostrar`
- El botón `Salir` finalizará la ejecución del programa

## 6.9. ResultSets Navegables y Actualizables

### ResultSet Avanzado en JDBC

Como ya sabemos un `ResultSet` es un interface de JAVA que dispone de los métodos necesarios para poder acceder a la información devuelta por una sentencia `SELECT` en una BBDD.

Por defecto, en un `ResultSet` sólo podemos ir hacia adelante y sólo podemos leer la información, pero este funcionamiento se puede modificar especificando el tipo de `ResultSet` y la concurrencia:

Los tipos se definen mediante constantes y puede ser alguno de los siguientes:

Constante	Descripción
<code>ResultSet.TYPE_FORWARD_ONLY</code>	Sólo se puede ir hacia adelante (método <code>next()</code> ). Es la opción por defecto
<code>ResultSet.TYPE_SCROLL_INSENSITIVE</code>	Podemos movernos en todas direcciones ( <code>beforeFirst()</code> , <code>first()</code> , <code>last()</code> , <code>next()</code> , <code>previous()</code> , <code>afterLast()</code> , <code>absolute(pos)</code> ). Los cambios hechos por otros usuarios a los datos que estamos gestionando en el <code>ResultSet</code> no son visibles para nosotros
<code>ResultSet.TYPE_SCROLL_SENSITIVE</code>	Podemos movernos en todas direcciones ( <code>beforeFirst()</code> , <code>first()</code> , <code>last()</code> , <code>next()</code> , <code>previous()</code> , <code>afterLast()</code> , <code>absolute(pos)</code> ). Los cambios hechos por otros usuarios a los datos que estamos gestionando en el <code>ResultSet</code> son visibles para nosotros

La concurrencia también se define con constantes:

Constante	Descripción
ResultSet.CONCUR_READ_ONLY	Sólo se puede leer la información. Por defecto
ResultSet.CONCUR_UPDATABLE	Se puede modificar la información del ResultSet (y, por tanto, de la BBDD)

**NOTA:** Las características indicadas pueden no estar implementadas (o no totalmente) por la BBDD. **NOTA:** En Oracle para que un ResultSet pueda ser actualizable, en la SELECT no se debe poner \*.

A la hora de trabajar con un ResultSet hay dos conceptos importantes: **visibilidad** que determina si se pueden ver o no los cambios hechos en el ResultSet internamente o externamente (por otros usuarios) y **detección** que permite que el ResultSet sea informado de los cambios realizados por otros usuarios.

En el tema de la visibilidad depende del tipo de objeto creado, dado que si se emplea un ResultSet de tipo SCROLL\_SENSITIVE, tras realizar un updateRow() se ejecuta automáticamente un refreshRow() con lo que las actualizaciones son visibles. La siguiente tabla muestra los efectos que tienen los diferentes tipos de ResultSet sobre la visibilidad:

Se puede ver	Forward Only	Scroll Insensitive	Scroll Sensitive
Inserciones propias	NO	NO	NO
Actualizaciones propias	SI	SI	SI
Eliminaciones propias	NO	SI	SI
Inserciones de otros	NO	NO	NO
Actualizaciones de otros	NO	NO	SI
Eliminaciones de otros	NO	NO	NO

### Uso

A la hora de especificar el tipo y concurrencia de un ResultSet depende de si empleamos una sentencia preparada o no.

```
Statement stmt = cnn.createStatement();
ResultSet rst = stmt.executeQuery(SQL, tipo, concurrencia);

PreparedStatement pst = cnn.prepareStatement(SQL, tipo, concurrencia);
```

### Modificación de la información

En caso de tener un ResultSet modificable, la forma de añadir un nuevo registro a la BBDD sería la siguiente:

```
rst.moveToInsertRow(); // Es necesario situarse en la llamada fila de inserción
rst.updateTipo(campo,valor); // Modificamos los campos con los valores deseados especificando el tipo de datos
...
rst.insertRow(); // Ejecutamos la inserción
```

En el caso de una modificación es similar:

```
rst.updateTipo(campo,valor); // Modificamos los campos con los valores deseados especificando el tipo de datos
...
rst.updateRow(); // Ejecutamos la modificación
```

En ambos casos, se puede cancelar la operación con:

```
rst.cancelRowUpdates();
```

Para eliminar un registro, lo leemos y lo eliminamos:

```
rst.next();
rst.deleteRow();
```

Disponemos de los métodos rowInserted(), rowUpdated() y rowDeleted() para saber si la fila actual ha sido insertada, modificada o eliminada pero, en el caso de Oracle no se proporciona mecanismos de detección de cambios, por lo que retornan siempre false.

### Ejemplo

Veamos un ejemplo simple con la tabla de Programadores:

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class FrmProgramador extends javax.swing.JFrame {
 private Statement stmt;
 private ResultSet rst;

 private enum Estado {
 NUEVO, MODIFICACION, CONSULTA
 };
 private Estado estado = Estado.CONSULTA;

 public FrmProgramador() {
```

```

initComponents();
try {
 stmt = Conexion.getConexion().createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
 ResultSet.CONCUR_UPDATABLE);
 rst = stmt.executeQuery("Select id,apellidos,nombre,fecha_alta From Programador");
 rst.first();
 mostrar();
} catch (SQLException ex) {
 Logger.getLogger(FrmProgramador.class.getName()).log(Level.SEVERE, null, ex);
}
}

@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {
 // Código generado automáticamente
 ...
}// </editor-fold>

private void btnPrimeroActionPerformed(java.awt.event.ActionEvent evt) {
 try {
 rst.first();
 mostrar();
 } catch (SQLException ex) {
 Logger.getLogger(FrmProgramador.class.getName()).log(Level.SEVERE, null, ex);
 }
}

private void btnAnteriorActionPerformed(java.awt.event.ActionEvent evt) {
 try {
 rst.previous();
 mostrar();
 } catch (SQLException ex) {
 Logger.getLogger(FrmProgramador.class.getName()).log(Level.SEVERE, null, ex);
 }
}

private void btnSiguienteActionPerformed(java.awt.event.ActionEvent evt) {
 try {
 rst.next();
 mostrar();
 } catch (SQLException ex) {
 Logger.getLogger(FrmProgramador.class.getName()).log(Level.SEVERE, null, ex);
 }
}

private void btnUltimoActionPerformed(java.awt.event.ActionEvent evt) {
 try {
 rst.last();
 mostrar();
 } catch (SQLException ex) {
 Logger.getLogger(FrmProgramador.class.getName()).log(Level.SEVERE, null, ex);
 }
}

private void btnNuevoActionPerformed(java.awt.event.ActionEvent evt) {
 estado = Estado.NUEVO;
 mostrar();
}

private void btnModificarActionPerformed(java.awt.event.ActionEvent evt) {
 estado = Estado.MODIFICACION;
 mostrar();
}

private void btnCancelActionPerfomed(java.awt.event.ActionEvent evt) {
 estado = Estado.CONSULTA;
 mostrar();
}

private void btnSalirActionPerfomed(java.awt.event.ActionEvent evt) {
 System.exit(0);
}

private void btnGuardarActionPerfomed(java.awt.event.ActionEvent evt) {
 if (txtApellidos.getText().isEmpty()) {
 JOptionPane.showMessageDialog(this, "Debe especificar los apellidos del programador", "Faltan datos", JOptionPane.ERROR_MESSAGE);
 txtApellidos.requestFocus();
 } else if (txtNombre.getText().isEmpty()) {
 JOptionPane.showMessageDialog(this, "Debe especificar el nombre del programador", "Faltan datos", JOptionPane.ERROR_MESSAGE);
 txtNombre.requestFocus();
 } else {
 try {
 int id;
 if (estado == Estado.NUEVO) {
 if (rst.last()) {
 id = rst.getInt("id") + 1;
 } else {
 id = 1;
 }
 rst.moveToInsertRow();
 rst.updateInt("id", id);
 }
 }
 }
}

```

```

 rst.updateString("apellidos", txtApellidos.getText());
 rst.updateString("nombre", txtNombre.getText());
 rst.updateDate("fecha_alta", new java.sql.Date((Date) ftxFechaAlta.getValue()).getTime());
 if (estado == Estado.MODIFICACION) {
 rst.updateRow();
 JOptionPane.showMessageDialog(this, "Registro MODIFICADO!");
 } else {
 rst.insertRow();
 JOptionPane.showMessageDialog(this, "Registro AÑADIDO!");
 rst.last();
 }
 estado = Estado.CONSULTA;
 mostrar();
 } catch (SQLException ex) {
 Logger.getLogger(FrmProgramador.class.getName()).log(Level.SEVERE, null, ex);
 }
}
}

private void btnEliminarActionPerformed(java.awt.event.ActionEvent evt) {
 try {
 if (JOptionPane.showConfirmDialog(this,
 "Esta operación no puede deshacerse\nnEstá seguro de eliminar el programador",
 "Confirmación",
 JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION) {
 int actual = rst.getRow();
 rst.deleteRow();
 stmt.executeUpdate("Select id,apellidos,nombre,fecha_alta From Programador");
 if (!rst.absolute(actual)) {
 rst.last();
 }
 JOptionPane.showMessageDialog(this, "Registro ELIMINADO!");
 mostrar();
 }
 } catch (SQLException ex) {
 JOptionPane.showMessageDialog(this, "ERROR:" + ex.getMessage());
 }
}
}

private void mostrar() {
 try {
 int pos = rst.getRow();
 if (rst.first()) {
 rst.absolute(pos);
 } else {
 estado = Estado.NUEVO;
 }
 if (estado != Estado.NUEVO) {
 ftxID.setValue(rst.getInt("id"));
 txtApellidos.setText(rst.getString("apellidos"));
 txtNombre.setText(rst.getString("nombre"));
 ftxFechaAlta.setValue(rst.getDate("fecha_alta"));
 } else {
 ftxID.setValue(null);
 txtApellidos.setText("");
 txtNombre.setText("");
 ftxFechaAlta.setValue(new Date());
 }
 txtApellidos.setEditable(estado != Estado.CONSULTA);
 txtNombre.setEditable(estado != Estado.CONSULTA);
 ftxFechaAlta.setEditable(estado != Estado.CONSULTA);
 btnNuevo.setEnabled(estado == Estado.CONSULTA);
 btnModificar.setEnabled(estado == Estado.CONSULTA);
 btnEliminar.setEnabled(estado == Estado.CONSULTA);
 btnCancelar.setEnabled(estado != Estado.CONSULTA);
 btnGuardar.setEnabled(estado != Estado.CONSULTA);
 btnPrimero.setEnabled(estado == Estado.CONSULTA && !rst.isFirst());
 btnAnterior.setEnabled(estado == Estado.CONSULTA && !rst.isFirst());
 btnSiguiente.setEnabled(estado == Estado.CONSULTA && !rst.isLast());
 btnUltimo.setEnabled(estado == Estado.CONSULTA && !rst.isLast());
 } catch (SQLException ex) {
 Logger.getLogger(FrmProgramador.class.getName()).log(Level.SEVERE, null, ex);
 }
}
}

public static void main(String args[]) {
 java.awt.EventQueue.invokeLater(new Runnable() {
 public void run() {
 new FrmProgramador().setVisible(true);
 }
 });
}
}

// Declaración de componentes de la interfaz
private javax.swing.JButton btnAnterior;
// ...
private javax.swing.JTextField txtNombre;
}

```

La salida:



## 6.10. Transacciones

### Transacciones en JDBC

Por defecto cada vez que se ejecuta una instrucción de manipulación de datos, se ejecuta un commit implícito en la BBDD.

Eso lo determina el método `setAutoCommit(boolean)` de la conexión:

Si desactivamos esta opción, podemos gestionar el proceso de actualización con:

- `commit()`: Se almacenan los cambios en la BBDD
- `rollback()`: Se deshacen los cambios
- `rollback(Savepoint s)`: Se deshacen hasta el punto indicado (primero se crea el Savepoint con el método `setSavepoint()` de la conexión y luego se hace el rollback hasta él). NO está disponible para Oracle con el driver thin.

#### Ejemplo

En este ejemplo vamos a diseñar una aplicación en la que vamos a provocar un error en medio del proceso de inserción. Primero lo haremos sin tener en cuenta las transacciones:

```
public class Transacciones {

 public static void main(String[] args) {
 Connection cnn = Conexion.getConexion();
 Statement stmt;
 ResultSet rst;
 try {
 stmt = cnn.createStatement();
 try {
 stmt.executeUpdate("Drop Table Transaccion");
 } catch (SQLException ex) {
 }
 stmt.executeUpdate("Create Table Transaccion(id integer primary key)");
 stmt.executeUpdate("Insert into Transaccion values(1)");
 stmt.executeUpdate("Insert into Transaccion values(2)");
 stmt.executeUpdate("Insert into Transaccion values(1)");
 stmt.executeUpdate("Insert into Transaccion values(4)");
 stmt.close();
 } catch (SQLException ex) {
 System.out.println("ERROR : " + ex.getMessage());
 Conexion.cerrarConexion();
 }
 try {
 cnn = Conexion.getConexion();
 stmt = cnn.createStatement();
 rst = stmt.executeQuery("Select * From Transaccion");
 while(rst.next()){
 System.out.println(rst.getInt("id"));
 }
 rst.close();
 stmt.close();
 Conexion.cerrarConexion();
 } catch (SQLException ex) {
 System.out.println("ERROR : " + ex.getMessage());
 }
 }
}
```

La salida:

```
ORA-00001: restricción única (SCOTT.SYS_C002354) violada
1
2
```

Como se puede ver el error se produce al intentar insertar un registro cuya clave ya existe. En este caso, la inserción de los dos registros anteriores se ha realizado.

Si desactivamos las transacciones deberemos explícitamente llamar a `rollback` o `commit` cuando sea necesario:

```
public class Transacciones {

 public static void main(String[] args) {
 Connection cnn = Conexion.getConexion();
 Statement stmt;
 ResultSet rst;
 try {

```

```

stmt = cnn.createStatement();
try {
 stmt.executeUpdate("Drop Table Transaccion");
} catch (SQLException ex) {
 ex.printStackTrace();
}
stmt.executeUpdate("Create Table Transaccion(id integer primary key)");
cnn.setAutoCommit(false);
stmt.executeUpdate("insert into Transaccion values(1)");
stmt.executeUpdate("insert into Transaccion values(2)");
stmt.executeUpdate("insert into Transaccion values(1)");
stmt.executeUpdate("insert into Transaccion values(4)");
cnn.commit();
stmt.close();
Conexion.cerrarConexion();
} catch (SQLException ex) {
 System.out.println("ERROR : " + ex.getMessage());
 try {
 cnn.rollback();
 } catch (SQLException ex1) {}
 Conexion.cerrarConexion();
}
try {
 cnn = Conexion.getConexion();
 stmt = cnn.createStatement();
 rst = stmt.executeQuery("Select * From Transaccion");
 while(rst.next()){
 System.out.println(rst.getInt("id"));
 }
 rst.close();
 stmt.close();
 Conexion.cerrarConexion();
} catch (SQLException ex) {
 System.out.println("ERROR : " + ex.getMessage());
}
}
}

```

La salida en este caso:

```
ORA-00001: restriccción única (SCOTT.SYS_C002354) violada
```

**NOTA:** Si la transacción no se confirma explícitamente (mediante un commit) no se realiza. Esto implica que es obligatorio poner commit para que los cambios tengan lugar. Por tanto, en nuestro código aunque no pusieramos rollback el resultado sería el mismo.

## 6.11. Procedimientos Almacenados

### Procedimientos Almacenados en JDBC

Muchos SGBD permiten la creación de procedimientos almacenados que no son sino conjuntos de instrucciones SQL que se almacenan en el servidor y que pueden ser ejecutadas desde los clientes. Los procedimientos son dependientes del SGBD utilizado y se suelen crear a partir de las herramientas del mismo.

En cualquier caso, es posible crear procedimientos almacenados desde Java (siempre que lo admita el SGBD) ejecutando `executeUpdate` con la cadena que genera el procedimiento almacenado (en Oracle y SQL Server por ejemplo tendría el formato siguiente: `CREATE PROCEDURE nombre AS instrucionSQL`). También hay funciones almacenadas (`CREATE FUNCTION función RETURN TIPO IS var AS instrucionSQL`).

Una vez creado el procedimiento almacenado (desde Java o desde el propio SGBD) es posible ejecutarlo desde Java. Para ello hay que crear un objeto `CallableStatement` mediante el comando:

```
CallableStatement cs = con.prepareCall("{call nombreproc}");
```

Los procedimientos almacenados pueden tener parámetros de entrada (`IN`), salida (`OUT`) o entrada/salida (`INOUT`).

Los parámetros se especifican como en las instrucciones preparadas:

```
{ call procedure(?,?) } ó { call ?= funcion(?) }
```

Los parámetros `IN` e `INOUT` DEBEN inicializarse antes de ejecutar el comando igual que en las sentencias preparadas:

```
cs.setString(1,"Valor")
```

Además, los parámetros `OUT` e `INOUT` (y el valor devuelto por una función) deben registrarse para poder emplearse mediante:

```
cs.registerOutParameter(int pos, int tipo[,int decimales])
```

Donde `tipo` es una constante definida en la clase `Types`. Por ejemplo:

```
cs.registerOutParameter(1, java.sql.Types.STRING)
```

#### Ejemplo

Vamos a crear una función que reciba como parámetro un número de departamento (`p_dep`). La función:

- Si el departamento no existe, lanzará una excepción
- Si existe, almacenará su nombre en el parámetro `p_nomdep`
- Si hay empleados en ese departamento:
- Almacenará el número de empleados en el parámetro `p_numemp`
- Retornará el salario medio de los empleados del departamento en `v_media`

- Si no hay empleados:
- Almacenará el número de empleados totales en `p_numemp`
- Almacenará 0 en el número de departamento `p_dep`
- Retornará el salario medio de todos los empleados de la empresa en `v_media`

La función:

```
create or replace FUNCTION SALARIOSEMPJAVA
(p_dep in out emp.deptno%TYPE,
 p_nomdep OUT DEPT.DNAME%TYPE,
 p_numemp OUT NUMBER
) RETURN NUMBER AS
 v_media number;
BEGIN
select dname into p_nomdep from dept where deptno=p_dep;
 select avg(sal),count(*) into v_media,p_numemp
 from emp
 where emp.DEPTNO=p_dep;
 if v_media is null then
 p_dep := 0;
 select avg(sal),count(*) into v_media,p_numemp
 from emp;
 end if;
RETURN v_media;

exception
when no_data_found then
 if p_nomdep is null then
 raise_application_error(-20001,'Departamento con número ' || p_dep || ' NO EXISTE!');
 end if;
END SALARIOSEMPJAVA;
```

El programa JAVA:

```
public class PruebaCallableStatement {

 public static void main(String[] args) {
 try {
 CallableStatement cs = Conexion.getConexion()
 .prepareCall("{ ? = call SALARIOSEMPJAVA(?, ?, ?)}");
 cs.registerOutParameter(1, java.sql.Types.DOUBLE);
 cs.registerOutParameter(2, java.sql.Types.INTEGER);
 cs.registerOutParameter(3, java.sql.Types.VARCHAR);
 cs.registerOutParameter(4, java.sql.Types.INTEGER);
 for(int d=10; d<50; d+=10){
 cs.setInt(2, d);
 cs.execute();
 System.out.println("Departamento : " + cs.getInt(2));
 System.out.println("Nombre : " + cs.getString(3));
 System.out.println("Nº Empleados : " + cs.getInt(4));
 System.out.printf("Sueldo Medio : %.2f\n", cs.getDouble(1));
 System.out.println("=====");
 }
 } catch (SQLException ex) {
 System.out.println("ERROR: " + ex.getMessage());
 }
 }
}
```

La salida:

```
Departamento : 10
Nombre : ACCOUNTING
Nº Empleados : 3
Sueldo Medio : 2916,67
=====
Departamento : 20
Nombre : RESEARCH
Nº Empleados : 5
Sueldo Medio : 2175,00
=====
Departamento : 30
Nombre : SALES
Nº Empleados : 6
Sueldo Medio : 1566,67
=====
Departamento : 0
Nombre : OPERATIONS
Nº Empleados : 14
Sueldo Medio : 2073,21
=====
ERROR: ORA-20001: Departamento con número 50 NO EXISTE!
ORA-06512: en "SCOTT.SALARIOSEMPJAVA", linea 24
ORA-06512: en linea 1
```

Como se puede ver, al seleccionar el departamento 40 retorna un 0 como departamento, lo que implica que el departamento existe pero no tiene empleados, así que la cuenta y la media es sobre toda la empresa. Por otro lado, al seleccionar el departamento 50 salta la excepción personalizada dado que dicho departamento no existe.

#### 6.12. Actividad ODS

## Programación

### Acceso a Bases de Datos

large-arrow-white

Previo

Acceso a Bases de Datos

Volver a la lista de las lecciones

Continuar

large-arrow-white

### Actividad ODS

- Tarea
- Calificaciones
- Estado de presentación
- Estadísticas
- Escala de calificaciones
- Establecer baremo
- Establecer competencias
- Finalización
- Visibilidad
- Ejemplos
- 
- Añadir
- Eliminar
- 

### Instrucciones

Se pide desarrollar una aplicación con información recopilada de Internet del consumo energético y coste aproximado debido al consumo en standby de una serie de equipos informáticos en una empresa. Posibles elementos a incluir: Ordenadores, Portátiles, Pantallas, Racks de distribución, etc. La aplicación deberá almacenar la información en la BBDD de Oracle y deberá disponer de un formulario en el que se podrá seleccionar el número de unidades de cada uno de los elementos indicados y deberá mostrar el consumo total por cada hora que están en stand by. Además, asumiendo que en la empresa trabajan 8 horas deberá calcular el consumo diario en stand by (16 horas), al mes y al año. Opcionalmente se puede poner un coste económico tras buscar el precio Kwh en alguna web.

- Editar
- 

editar

### Tarea

- Tipo: Ejercicio escrito online (Ensayo)
- Máxima puntuación individual: 100
- Calificación: Cuenta para la media
- Categoría: Ningunos

### Programa

- Fecha límite:
- 17 Abr, 9:59 pm
- No dada

### Calificación

- Fecha límite: 98, Enviado: 0
- calificadas: 0

editar

### Opciones

Biblioteca: Personal

Intentos máximos: 1

Permitir entregas fuera de plazo: No

Grados de publicación: Instant

Permita que los estudiantes comenten las entregas de otros estudiantes.: No

Desactivar fecha de entrega: No

large-arrow-white

Previo

Volver a la lista de las lecciones

Continuar

large-arrow-white

## 7. Capa de Persistencia de JAVA JPA

### 7.1. Introducción a JPA

#### ORM y JPA en Java

Las opciones vistas hasta el momento para gestionar el almacenamiento de la información de objetos en bases de datos tienen varios problemas. En el caso de emplear JDBC puro, no existe un mecanismo simple para gestionar los objetos y sus relaciones en la base de datos. En el caso de trabajar directamente con objetos, el problema lo tenemos a la hora de gestionar las relaciones entre clases.

Para facilitar la transición entre clases y tablas de bases de datos se fueron definiendo una serie de frameworks o entornos de trabajo que proporcionaban un mecanismo para realizar el mapeo entre clases y tablas de la base de datos (ORM - Object Relational Mapping). Probablemente el más utilizado de estos mecanismos sea Hibernate.

Posteriormente en la versión empresarial de JAVA (J2EE en su versión 3) se introdujo JPA (Java Persistence API) que permitía especificar la relación entre el modelo orientado a objetos y el modelo relacional mediante anotaciones en el código y unos ficheros de configuración. Hibernate es compatible con JPA a través de Hibernate Annotations.

El elemento básico en un ORM es la entidad. Una entidad es una clase con anotaciones que se asocia a una tabla de una BBDD. Una anotación es una forma de añadir metadatos (información sobre los datos) al código fuente Java para que estén disponibles para la aplicación cuando se compila o se ejecuta. Por ejemplo, la anotación `@Override` avisa al compilador de que el método sobreescribe un método de la clase padre. Esto permite al compilador que dicho método exista en la clase padre y evitar así posibles errores si no hemos definido el método hijo correctamente.

#### Entidades

Las entidades permiten definir las clases que deben ser persistidas en la base de datos. Las entidades, sus campos y sus relaciones se definen mediante anotaciones en el código de la clase.

La anotación `javax.persistence.Entity` es la que define una clase como persistente. La clase se mapea a una tabla y, cada instancia de la clase, a una fila de dicha tabla. Las anotaciones en el código deben ir precedidas del símbolo `@`.

Definir una clase como persistente implica cumplir las siguientes reglas:

- La clase debe llevar la anotación `Entity`
- Debe incluir un constructor sin argumentos de tipo `public` o `protected`
- La tabla referencia debe disponer de una clave primaria y la entidad el campo correspondiente anotado con `@Id`.
- Ni la clase ni ninguno de sus métodos puede ser `final`
- Las propiedades persistentes no pueden ser públicas y, por tanto, deben ser accedidas mediante métodos de acceso (`set/get`)

Las propiedades (los campos) que pueden ser persistidas deben ser de alguno de los siguientes tipos:

- Tipos primitivos (`int, float, double...`)
- Cadenas `java.lang.String`
- Wrappers de los tipos básicos (`java.lang.Integer, java.lang.Double, etc`)
- Clases de fecha (`util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp`)
- Tipos de datos definidos por el usuario (deben implementar `java.io.Serializable`)
- Matrices `byte[], Byte[], char[]` y `Character[]`
- Enumeraciones
- Otras entidades o colecciones de entidades
- Clases incrustadas

La anotación `@Entity` se pone al nivel de la clase y se mapea con la tabla que tenga el mismo nombre. Opcionalmente se le puede pasar un atributo `name` cuando el nombre de la entidad no coincide con el nombre de la tabla:

Asociamos la entidad Cliente a la tabla cliente de la BBDD

```
@Entity
public class Cliente {
 ...
}
```

Asociamos la entidad Cliente a la tabla customer de la BBDD

```
@Entity(name="customer")
public class Cliente {
 ...
}
```

**NOTA:** Si necesitamos ser más específicos con la tabla a emplear (nombre, esquema...) podemos añadir, a nivel de clase, la anotación `@Table`

#### Campos o propiedades persistentes

A la hora de definir qué campos de la clase queremos persistir en la base de datos, podemos optar por persistir los campos directamente o hacerlo a través de los métodos de acceso. Eso implica que a la hora de almacenar/recuperar un campo, se almacene/escriba directamente el valor de la propiedad o se ejecute el método `set/get` para almacenar/recuperar el valor.

**NOTA:** No se deben mezclar campos y propiedades persistentes en la misma clase. Es decir, podemos poner anotaciones sobre los campos o sobre los métodos get asociados a los mismos, pero no en ambos sitios.

Para indicar que un campo no debe ser persistido, se emplea la anotación `javax.persistence.Transient` (también vale con definir el campo con el modificador transient):

```
private transient String comentarios;

o

@Transient
private String comentarios;
```

**NOTA:** Las propiedades estáticas o finales se definen automáticamente como no persistentes.

Los campos o propiedades persistentes se pueden anotar con la anotación `javax.persistence.Basic` que es la anotación por defecto y, por tanto, no es necesario indicarla. Podemos incluir el atributo fetch (valores lazy o eager por defecto) para indicar si queremos que la información del campo se recupere de la base de datos en cuanto se acceda a la tabla por primera vez (por defecto) o cuando se acceda al campo por primera vez.

Otra anotación que puede ser útil es `javax.persistence.Column` que permite especificar las características de la propiedad o campo en la base de datos. Los elementos de la anotación pueden ser:

- `columnDefinition`: el código SQL que queremos que se use para definir el campo. Opcional.
- `insertable`: si se debe incluir en las instrucciones INSERT. Por defecto es true
- `length`: La longitud del campo (si es una cadena). Por defecto 255.
- `name`: El nombre de la columna en la tabla. Por defecto es el nombre del campo de la clase
- `nullable`: Si el campo puede ser null. Por defecto true
- `default`: Valor por defecto
- `scale`: El número de decimales en los valores con decimales. Por defecto 0. Es necesario especificarlo si queremos controlar exactamente cuántos decimales queremos
- `precision`: El número de dígitos del campo (incluidos decimales). Por defecto 0. Es necesario especificarlo si queremos controlar exactamente cuántos dígitos queremos
- `table`: El nombre de la tabla en la que se encuentra el campo. Por defecto en la tabla de la entidad
- `unique`: Si el campo debe ser único (también se puede especificar con la anotación `javax.persistence.UniqueConstraint`)
- `updatable`: Si el campo debe emplearse en las instrucciones UPDATE. Por defecto true.

Una columna con definición, longitud, valor por defecto y no modificable (la anotación `@Basic` es opcional)

```
@Basic
@Column(columnDefinition="VARCHAR2(20)", length=30, default='Bilbao', updatable=false)
private String poblacion;
```

Una propiedad numérica de 5 enteros y dos decimales y con otro nombre de columna

```
@Column(name="PVENTA", precisión=2, scale=5)
public getPrecio() {
 return precio;
}
```

**NOTA:** Estas dos anotaciones no podrían ir en la misma entidad puesto que una es de campo y la otra de propiedad.

#### Clave primaria

Como ya se ha indicado, toda entidad (y por tanto la tabla) debe incluir una clave principal. Dicho campo debe ir anotado con la anotación `@Id`.

Si la clave debe ser generada automáticamente, se puede emplear la anotación `@GeneratedValue` que tiene dos atributos: `strategy` (cómo se van a obtener los valores de la clave) y `generator` (el objeto, en su caso, que se empleará en dicha obtención):

strategy	generator	Descripción
GenerationType.AUTO		Es JPA quien elige el tipo de generador a usar en función de la BBDD (por ejemplo SEQUENCE en Oracle o IDENTITY en SQL Server)
GenerationType.SEQUENCE	nombreSecuencia	Se emplea un objeto de BBDD de tipo secuencia. Si no se especifica generator se asume que la secuencia es hibernate_sequence, si se especifica indica el nombre de la secuencia a emplear. Por ejemplo, en Oracle crearemos la secuencia con la siguiente instrucción: <code>create sequence hibernate_sequence INCREMENT BY 1 START WITH 1</code>
GenerationType.IDENTITY		El campo id emplea un tipo de datos autoincrementable propio de la BBDD. Por ejemplo, en SQL Server: <code>create table tabla (id integer identity(1,1) primary key, ...)</code>
GenerationType.TABLE	nombreTabla	Se emplea una tabla para almacenar el siguiente id. Si no se especifica generator se asume que la tabla es hibernate_sequences, si se especifica indica el nombre de la tabla a emplear. Por ejemplo, en Oracle crearemos la tabla con la siguiente instrucción: <code>create table hibernate_sequences (sequence_name varchar2(100), next_val number(4))</code>

#### 7.2. Primeros Pasos Maven e Hibernate

#### JPA con Hibernate

JPA en sí mismo es una especificación (un API) que los desarrolladores de soluciones deben implementar, por lo que hay diferentes productos compatibles con el mismo (EclipseLink, Hibernate, Apache OpenJPA...). De todos ellos, el más usado es Hibernate y es el que emplearemos este curso y el siguiente.

Así que, para poder trabajar con JPA necesitamos las librerías de alguna implementación de la misma. En nuestro caso vamos a añadir las siguientes dependencias al pom.xml para emplear Hibernate:

```
<?xml version="1.0" encoding="UTF-8"?>
<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
<modelVersion>4.0.0</modelVersion>
<groupId>net.zabalburu.jpa</groupId>
<artifactId>PruebaJPA</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>1.8</maven.compiler.source>
 <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies>
 <dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-entitymanager</artifactId>
 <version>5.4.15.Final</version>
 </dependency>
 <dependency>
 <groupId>org.hibernate.validator</groupId>
 <artifactId>hibernate-validator</artifactId>
 <version>6.1.5.Final</version>
 </dependency>
 <dependency>
 <groupId>com.oracle.database.jdbc</groupId>
 <artifactId>jdbc8</artifactId>
 <version>19.3.0.0</version>
 </dependency>
</dependencies>
</project>
```

Cargamos las dependencias entitymanager (el gestor de entidades que incluye el núcleo de hibernate) y validator (opcional, nos permite añadir validaciones a los datos a la hora de insertarlos / modificarlos).

### Fichero de persistencia (persistence.xml)

Para poder emplear la persistencia es necesario indicar a JAVA con qué BBDD nos vamos a conectar y qué entidades vamos a usar. Esta información se almacena en lo que se denomina Unidad de Persistencia que se define (puede haber varias) en el fichero persistence.xml que debe ubicarse en la carpeta src/main/resources/META-INF del proyecto. Esta carpeta habrá que crearla manualmente.

**NOTA:** NetBeans dispone de asistentes que permiten crear el fichero automáticamente pero, en nuestro caso, vamos a ver el proceso manual para que pueda servir para cualquier entorno de desarrollo.

Básicamente en un fichero de persistencia se puede definir:

- La configuración del EntityManager que es la clase encargada de gestionar las entidades
- El origen de los datos
- Extensiones del desarrollador de la implementación específica de JPA
- Las entidades
- Información para el mapeado de las clases (entidades) a los campos. Actualmente, el método preferido es usar anotaciones.

La estructura básica del fichero será similar a la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
<persistence-unit name="nombreUP" transaction-type="RESOURCE_LOCAL">
 <class>entidad1</class>
 <class>entidad2</class>
 <class>...</class>
 <properties>
 <property name="javax.persistence.jdbc.url" value="url"/>
 <property name="javax.persistence.jdbc.password" value="pwd"/>
 <property name="javax.persistence.jdbc.driver" value="driver"/>
 <property name="javax.persistence.jdbc.user" value="usuario"/>
 <!-- Otras propiedades de JPA y/o del desarrollador de la implementación -->
 </properties>
</persistence-unit>
<persistence-unit name="PU2" transaction-type="RESOURCE_LOCAL">
 ...
</persistence-unit>
...
</persistence>
```

En este fichero:

- Se define una unidad de persistencia con el nombre `nombreUP` y se indica que las transacciones se gestionan localmente (`RESOURCE_LOCAL`). Este es el valor estándar para JAVA SE. En el caso de JAVA EE (el año próximo) se emplea `Java Transaction API` (indicado como `JTA`)
- En el caso de J2EE se especifica posteriormente el atributo `jta-data-source` que indica el `DataSource` con el que nos vamos a conectar a la BBDD (se define en el

servidor). Dado que usamos JAVA SE no vamos a usar esta opción este año.

- Luego se definen las clases. Esta indicación es opcional, dado que se buscan todas las clases etiquetadas como `@Entity`. Para hacerlo obligatorio se puede añadir un elemento `<exclude-unlisted-clases/>`
- En el caso de J2SE se especifican ahora las propiedades de la conexión. Las básicas son:
  - `javax.persistence.jdbc.url`: La cadena de conexión a la BBDD (por ejemplo `jdbc:oracle:thin:@localhost:1521:ORCL` para Oracle, `jdbc:sqlserver://localhost:1234;databaseName=Northwind` para SQLServer)
  - `javax.persistence.jdbc.driver`: La clase del driver (por ejemplo `oracle.jdbc.OracleDriver` para Oracle, `com.microsoft.sqlserver.jdbc.SQLServerDriver` para SQL Server)
- `javax.persistence.jdbc.user`: El usuario (`scott`)
- `javax.persistence.jdbc.password`: La contraseña (`tiger`)
- Luego pueden ir propiedades propias de la implementación. En Hibernate hay decenas de propiedades. Algunas de las más interesantes son:
  - `hibernate.dialect`: El tipo de BBDD (`org.hibernate.dialect.Oracle8iDialect` para Oracle, `org.hibernate.dialect.SQLServerDialect` para SQLServer)
  - `hibernate.connection.autocommit`: Puede ser `true` o `false` (por defecto)
  - `hibernate.show_sql`: Muestra (`true`) o no (`false`, por defecto) las instrucciones SQL que ejecuta Hibernate
  - `hibernate.format_sql`: Muestra las consultas anteriores formateadas
  - `hibernate.hbm2ddl.auto`: Determina si los cambios en la estructura de las entidades implican un cambio automático en la BBDD. Es decir, si añado una nueva entidad en el proyecto o modifco algún campo de alguna entidad, si esos cambios deben implementarse en la BBDD automáticamente o manualmente. Las opciones son:
    - `none`: Por defecto. Cualquier cambio en las entidades debe hacerse manualmente en la BBDD. Es decir, hay que modificar, crear las tablas antes de ejecutar el código
    - `create`: Se eliminan todas las tablas asociadas a las entidades y se crean de nuevo. Si es necesario crear enumeraciones o tablas auxiliares para los ids autogenerados, también se crean
    - `update`: Similar al anterior, con la diferencia de que las tablas, si existen, no se eliminan. Es decir, comprueba las entidades y modifica / crea las tablas en función de las mismas
    - `validate`: Comprueba el esquema pero no hace cambios en la BBDD. Por ejemplo, si añadimos un campo a la Entidad y no lo añadimos en la BBDD o viceversa nos dará una excepción `Schema-validation`. Es decir, comprueba que haya correspondencia entre las entidades y las tablas de la BBDD
    - `drop`: Al finalizar la aplicación, se eliminan todas las tablas asociadas a las entidades

Un fichero básico para el ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://www.w3.org/2001/XMLSchema-instance">
 <persistence-unit name="JPAPU" transaction-type="RESOURCE_LOCAL">
 <properties>
 <property name="javax.persistence.jdbc.url"
 value="jdbc:oracle:thin:@localhost:1521:ORCL"/>
 <property name="javax.persistence.jdbc.password" value="tiger"/>
 <property name="javax.persistence.jdbc.driver" value="oracle.jdbc.OracleDriver"/>
 <property name="javax.persistence.jdbc.user" value="scott"/>
 <property name="hibernate.dialect" value="org.hibernate.dialect.Oracle8iDialect"/>
 <property name="hibernate.show_sql" value="true"/>
 <property name="hibernate.format_sql" value="true"/>
 <property name="hibernate.hbm2ddl.auto" value="update"/>
 </properties>
 </persistence-unit>
</persistence>
```

En este caso estamos indicando que queremos que nos cree / modifique las tablas en base a las entidades que definamos y que nos muestre las instrucciones SQL que se ejecutan en la BBDD.

## Creación de una Entidad

```
@Entity
@Table(name = "pemp")
public class PEmpleado implements Serializable{

 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 private Integer id;

 @Column(columnDefinition = "VARCHAR2(50)")
 @Basic(optional = false)
 private String apellidos;

 private String nombre;

 @Column(name="fecha_alta")
 private Date fechaAlta;

 public PEmpleado() {
 }

 public Integer getId() {
```

```

 return id;
 }

 public void setId(Integer id) {
 this.id = id;
 }

 public String getApellidos() {
 return apellidos;
 }

 public void setApellidos(String apellidos) {
 this.apellidos = apellidos;
 }

 public String getNombre() {
 return nombre;
 }

 public void setNombre(String nombre) {
 this.nombre = nombre;
 }

 public Date getFechaAlta() {
 return fechaAlta;
 }

 public void setFechaAlta(Date fechaAlta) {
 this.fechaAlta = fechaAlta;
 }

 @Override
 public boolean equals(Object obj) {
 if (this == obj) {
 return true;
 }
 if (obj == null) {
 return false;
 }
 if (getClass() != obj.getClass()) {
 return false;
 }
 final PEmpleado other = (PEmpleado) obj;
 if (!Objects.equals(this.id, other.id)) {
 return false;
 }
 return true;
 }

 @Override
 public String toString() {
 return apellidos + ", " + nombre;
 }
}

```

## Creando el EntityManager

Para poder trabajar con las entidades, necesitamos crear un objeto de tipo `EntityManager`. Esta clase es la que se encarga de gestionar las entidades y ejecutar las consultas y operaciones sobre la BBDD.

Para obtener un `EntityManager`, necesitamos un objeto `EntityManagerFactory` que creamos a partir de un objeto `Persistence`:

```

public class GestionProyectos {

 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();

 em.close();
 }
}

```

Si ejecutamos la aplicación veremos entre los mensajes las instrucciones sql de creación de la tabla y la secuencia que Hibernate crea a partir de la entidad (dado que hemos indicado `update` como opción en `hibernate.hbm2ddl.auto`)

```

may. 14, 2020 5:48:03 P. M. org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [name: JPAPU]
...
Hibernate:
create table pemp (
 id number(10,0) not null,
 apellidos VARCHAR2(50) not null,
 fecha_alta date,
 nombre varchar2(255),
 primary key (id)
)
Hibernate: create sequence hibernate_sequence start with 1 increment by 1
may. 14, 2020 5:48:05 P. M. org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService
...

```

Como podemos ver, se ha creado la tabla con nombre `pemp`, ha empleado nuestra definición de campo para `apellidos` (y lo ha puesto como `not null`). Por otro lado, ha definido `id` como clave primaria y ha creado una **secuencia** para asignar los valores a dicho `id` comenzando desde 1 e incrementándose en 1 cada vez.

### 7.3. Primeros Pasos EclipseLink

#### JPA con EclipseLink

JPA en sí mismo es una especificación (un API) que los desarrolladores de soluciones deben implementar, por lo que hay diferentes productos compatibles con el mismo (EclipseLink, Hibernate, Apache OpenJPA...). En este curso emplearemos EclipseLink dado que es el que viene con Netbeans.

##### Fichero de persistencia (persistence.xml)

Para poder emplear la persistencia es necesario indicar a JAVA con qué BBDD nos vamos a conectar y qué entidades vamos a usar. Esta información se almacena en lo que se denomina Unidad de Persistencia que se define (puede haber varias) en el fichero `persistence.xml` que debe ubicarse en la carpeta `src/main/resources/META-INF` del proyecto. Esta carpeta habrá que crearla manualmente.

**NOTA:** NetBeans dispone de asistentes que permiten crear el fichero automáticamente pero, en nuestro caso, vamos a ver el proceso manual para que pueda servir para cualquier entorno de desarrollo.

Básicamente en un fichero de persistencia se puede definir:

- La configuración del EntityManager que es la clase encargada de gestionar las entidades
- El origen de los datos
- Extensiones del desarrollador de la implementación específica de JPA
- Las entidades
- Información para el mapeado de las clases (entidades) a los campos. Actualmente, el método preferido es usar anotaciones.

La estructura básica del fichero será similar a la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://www.w3.org/2001/XMLSchema-instance">
 <persistence-unit name="nombreUP" transaction-type="RESOURCE_LOCAL">
 <class>entidad1</class>
 <class>entidad2</class>
 <class>...</class>
 <properties>
 <property name="javax.persistence.jdbc.url" value="url"/>
 <property name="javax.persistence.jdbc.password" value="pwd"/>
 <property name="javax.persistence.jdbc.driver" value="driver"/>
 <property name="javax.persistence.jdbc.user" value="usuario"/>
 <!-- Otras propiedades de JPA y/o del desarrollador de la implementación -->
 </properties>
 </persistence-unit>
 <persistence-unit name="PU2" transaction-type="RESOURCE_LOCAL">
 ...
 </persistence-unit>
 ...
</persistence>
```

En este fichero:

- Se define una unidad de persistencia con el nombre `nombreUP` y se indica que las transacciones se gestionan localmente (RESOURCE-LOCAL). Este es el valor estándar para JAVA SE. En el caso de JAVA EE (el año próximo) se emplea Java Transaction API (indicado como JTA)
- En el caso de J2EE se especifica posteriormente el atributo `jta-data-source` que indica el DataSource con el que nos vamos a conectar a la BBDD (se define en el servidor). Dado que usamos JAVA SE no vamos a usar esta opción este año.
- Luego se definen las clases. Esta indicación es opcional, dado que se buscan todas las clases etiquetadas como `@Entity`. Para hacerlo obligatorio se puede añadir un elemento `<exclude-unlisted-classes>`
- En el caso de J2SE se especifican ahora las propiedades de la conexión. Las básicas son:
  - `javax.persistence.jdbc.url`: La cadena de conexión a la BBDD (por ejemplo `jdbc:oracle:thin:@localhost:1521:ORCL` para Oracle, `jdbc:sqlserver://localhost:1234;databaseName=Northwind` para SQL Server)
  - `javax.persistence.jdbc.driver`: La clase del driver (por ejemplo `oracle.jdbc.OracleDriver` para Oracle, `com.microsoft.sqlserver.jdbc.SQLServerDriver` para SQL Server)
  - `javax.persistence.jdbc.user`: El usuario (`scott`)
  - `javax.persistence.jdbc.password`: La contraseña (`tiger`)
  - `javax.persistence.schema-generation.database.action`: Permite indicar si queremos crear / modificar las tablas si no existen:
- `none`: Por defecto. Cualquier cambio en las entidades debe hacerse manualmente en la BBDD. Es decir, hay que modificar, crear las tablas antes de ejecutar el código
- `create`: Se crean las tablas si no existen. Si es necesario crear enumeraciones o tablas auxiliares para los ids autogenerados, también se crean
- `drop-and-create`: Similar al anterior, pero se eliminan todas las tablas antes de crearlas.
- `create-or-extend-tables`: Si la tabla no existe, la crea. Si existe la actualiza si hay cambios.
- Luego pueden ir propiedades propias de la implementación.

Un fichero básico para el ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://eclipsesource.com/jpa/persistence_2_1.xsd">
 <persistence-unit name="JPAPU" transaction-type="RESOURCE_LOCAL">
 <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
 <properties>
 <property name="javax.persistence.jdbc.url" value="jdbc:oracle:thin:@localhost:1521:ORCL"/>
 <property name="javax.persistence.jdbc.user" value="scott"/>
 <property name="javax.persistence.jdbc.driver" value="oracle.jdbc.OracleDriver"/>
 <property name="javax.persistence.jdbc.password" value="tiger"/>
 <property name="javax.persistence.schema-generation.database.action" value="create"/>
 </properties>
 </persistence-unit>
</persistence>
```

En este caso estamos indicando que queremos que nos cree / modifique las tablas en base a las entidades que definamos y que nos muestre las instrucciones SQL que se ejecutan en la BBDD.

## Creación de una Entidad

```
@Entity
@Table(name = "pemp")
public class PEmpleado implements Serializable {

 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 private Integer id;

 @Column(columnDefinition = "VARCHAR2(50)")
 @Basic(optional = false)
 private String apellidos;

 private String nombre;

 @Column(name="fecha_alta")
 private Date fechaAlta;

 public PEmpleado() {
 }

 public Integer getId() {
 return id;
 }

 public void setId(Integer id) {
 this.id = id;
 }

 public String getApellidos() {
 return apellidos;
 }

 public void setApellidos(String apellidos) {
 this.apellidos = apellidos;
 }

 public String getNombre() {
 return nombre;
 }

 public void setNombre(String nombre) {
 this.nombre = nombre;
 }

 public Date getFechaAlta() {
 return fechaAlta;
 }

 public void setFechaAlta(Date fechaAlta) {
 this.fechaAlta = fechaAlta;
 }

 @Override
 public boolean equals(Object obj) {
 if (this == obj) {
 return true;
 }
 if (obj == null) {
 return false;
 }
 if (getClass() != obj.getClass()) {
 return false;
 }
 final PEmpleado other = (PEmpleado) obj;
 if (!Objects.equals(this.id, other.id)) {
 return false;
 }
 return true;
 }

 @Override
 public String toString() {
```

```

 return apellidos + ", " + nombre;
 }
}

```

**NOTA:** Deberemos añadir la entidad recién definida al fichero persistence.xml:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
<persistence-unit name="JPAPU" transaction-type="RESOURCE_LOCAL">
 <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
 <class>pruebabaja.PEmpleado</class>
 <properties>
 ...
 </properties>
</persistence-unit>
</persistence>

```

## Creando el EntityManager

Para poder trabajar con las entidades, necesitamos crear un objeto de tipo EntityManager. Esta clase es la que se encarga de gestionar las entidades y ejecutar las consultas y operaciones sobre la BBDD.

Para obtener un EntityManager, necesitamos un objeto EntityManagerFactory que creamos a partir de un objeto Persistence:

```

public class GestionProyectos {

 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();

 em.close();
 }
}

```

Podemos comprobar que se ha creado la tabla con nombre pemp, he empleado nuestra definición de campo para apellidos (y lo ha puesto como not null). Por otro lado, ha definido id como clave primaria y ha creado una secuencia para asignar los valores a dicho id comenzando desde 1 e incrementándose en 1 cada vez.

### 7.4. EntityManager Acceso y modificación de objetos

#### Creación y modificación de Objetos

Una vez obtenido el EntityManager, disponemos de diversos métodos para crear / modificar / eliminar objetos de la BBDD. Dado que, por defecto la opción autoCommit está desactivada, cualquier operación de modificación de objetos debe realizarse dentro de una transacción.

#### Añadir objetos

Para añadir objetos, basta con crear un objeto de la entidad deseada y llamar al método persist:

```

public class GestionProyectos {
 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 PEmpleado e = new PEmpleado();
 em.getTransaction().begin();
 e.setApellidos("López Marin");
 e.setNombre("Ana");
 e.setFechaAlta(new GregorianCalendar(2020, 4, 1).getTime());
 em.persist(e);
 e = new PEmpleado();
 e.setApellidos("De Miguel López");
 e.setNombre("Carlos");
 e.setFechaAlta(new GregorianCalendar(2020, 4, 10).getTime());
 em.persist(e);
 e = new PEmpleado();
 e.setApellidos("Simón Tercilla");
 e.setNombre("Natalia");
 e.setFechaAlta(new GregorianCalendar(2020, 6, 13).getTime());
 em.persist(e);
 em.getTransaction().commit();
 em.close();
 }
}

```

Si miramos la tabla en la BBDD:

#	ID	APELLIDOS	FECHA_ALTA	NOMBRE
1	1	López Marin	2020-05-01 00:00:00.000	Ana
2	2	De Miguel López	2020-05-10 00:00:00.000	Carlos
3	3	Simón Tercilla	2020-07-13 00:00:00.000	Natalia

## Modificar objetos

Para modificar un objeto, hay que emplear el método merge. Si el objeto existe (en base a su ID) se actualiza, si no existe se crea:

```
public class GestionProyectos {
 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 PEmpleado e = new PEmpleado();
 em.getTransaction().begin();
 e.setId(1);
 e.setApellidos("López Marín");
 e.setNombre("Ana María");
 e.setFechaAlta(new GregorianCalendar(2020, 4, 1).getTime());
 em.merge(e);
 e = new PEmpleado();
 e.setApellidos("Ginés Ruiz");
 e.setNombre("Luis");
 e.setFechaAlta(new GregorianCalendar(2020, 10, 21).getTime());
 em.merge(e);
 em.getTransaction().commit();
 em.close();
 }
}
```

Si miramos la tabla en la BBDD:

#	ID	APELLIDOS	FECHA_ALTA	NOMBRE
1	4	Ginés Ruiz	2020-11-21 00:00:00	Luis
2	1	López Marín	2020-05-01 00:00:00	Ana María
3	2	De Miguel López	2020-05-10 00:00:00	Carlos
4	3	Simón Teruel	2020-07-13 00:00:00	Natalia

## Buscar un objeto en base a su ID

Para localizar un objeto en base a su id, tenemos el método

```
find(Clase, id)
```

Donde Clase identifica la entidad a buscar e id el valor del campo ID de dicha entidad. Este método devuelve el objeto de tipo Clase con ese id si existe o un valor null si no existe.

## Eliminar objetos

Para modificar un objeto, hay que emplear el método remove. Basta con pasarle el objeto a eliminar que debemos recuperarlo a través del EntityManager para que lo gestione (por ejemplo, empleando el método find):

```
public class GestionProyectos {
 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 PEmpleado e = em.find(PEmpleado.class, 4);
 System.out.println(e + " [" + e.getId() + "] ");
 em.getTransaction().begin();
 em.remove(e);
 em.getTransaction().commit();
 em.close();
 }
}
```

Si miramos la tabla en la BBDD:

#	ID	APELLIDOS	FECHA_ALTA	NOMBRE
1	1	López Marín	2020-05-01 00:00:00	Ana María
2	2	De Miguel López	2020-05-10 00:00:00	Carlos
3	3	Simón Teruel	2020-07-13 00:00:00	Natalia

## 7.5. Consultas Básicas JPQL

### Consultas en JPA con JPQL

Para consultas en las tablas generadas con JPA se emplea el Java Persistence Query Language (JPQL) que permite realizar consultas estáticas y dinámicas sobre las tablas de la base de datos. JPQL es un lenguaje similar a SQL pero que trabaja sobre las entidades y no sobre las tablas.

Para gestionar las consultas disponemos de la interfaz `javax.persistence.Query`. Podemos obtener un objeto consulta a través del `EntityManager` con el método:

```
Query q = em.createQuery(instruccionJPQL);
```

Como ya se ha dicho, la consulta trabaja con entidades y propiedades y no sobre tablas y campos. Las entidades deben llevar alias obligatoriamente y las consultas pueden llevar condiciones, agruparse, ordenarse, etc. Veremos más en detalle la sintaxis JPQL más adelante.

Por otro lado las consultas pueden ser de tipo SELECT (recuperar objetos), UPDATE (modificar las propiedades de los objetos) y DELETE (eliminar objetos).

Si estamos empleando una select, podemos recuperar los objetos retornados con los siguientes métodos:

- `q.getSingleResult()`: Cuando la consulta sólo retorna un objeto. Lo retorna como un Object por lo que habrá que hacer un casting. Si no existe el objeto, se lanza una excepción `javax.persistence.NoResultException`.
- `q.getResultList()`: Retorna una lista con los objetos encontrados. Si no hay ningún objeto, retorna una lista vacía.

#### Ejemplo:

Retornar todos los empleados:

```
public class GestionProyectos {
 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 Query q = em.createQuery("Select e From PEmpleado e "+
 "Order By e.apellidos, e.nombre");
 List<PEmpleado> empleados = q.getResultList();
 System.out.println("Listado de Empleados\n");
 System.out.printf("%-5s%-30s%10s\n#", "Nombre", "Apellido", "Fecha Alta");
 System.out.printf("%-5s%-30s%10s\n==", "====", "=====", "=====");
 for(PEmpleado e : empleados){
 System.out.printf("%-5d%-30s%10tD\n", e.getId(), e.toString(), e.getFechaAlta());
 }
 em.close();
 }
}
```

La salida:

```
Listado de Empleados
```

#	Nombre	Apellido	Fecha Alta
2	De Miguel López, Carlos		05/10/20
1	López Marín, Ana María		05/01/20
3	Simón Tercilla, Natalia		07/13/20

### Parámetros

En una consulta JPQL se pueden emplear parámetros. Estos parámetros pueden ser:

- por nombre: Se asocia el parámetro con un nombre significativo. Sintaxis: `:nombre`
- por índice: Se asocia el parámetro a una posición. Sintaxis: `?índice`

Por claridad, se suelen emplear los parámetros por nombre.

Una vez definidos los parámetros en la consulta, antes de obtener los resultados se deben asignar valores a los parámetros con el método:

```
q.setParameter("nombre", valor);
q.setParameter(indice, valor);
```

#### Ejemplo:

Buscar un empleado en base a sus apellidos y su nombre:

```
public class GestionProyectos {
 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 String apellidos = JOptionPane.showInputDialog("Apellidos");
 String nombre = JOptionPane.showInputDialog("Nombre");
 Query q = em.createQuery("Select e From PEmpleado e "+
 "Where e.apellidos=:apellidos and e.nombre=:nombre");
 q.setParameter("apellidos", apellidos);
 q.setParameter("nombre", nombre);
 try {
 PEmpleado e = (PEmpleado) q.getSingleResult();
 JOptionPane.showMessageDialog(null,
 "Empleado : " + e.toString() +
 "\nId : " + e.getId() +
 "\nFecha Alta : " + DateFormat.getDateInstance().format(e.getFechaAlta()),
 e.toString(),
 JOptionPane.INFORMATION_MESSAGE);
 }
 }
}
```

```

 } catch (NoResultException ex) {
 JOptionPane.showMessageDialog(null,
 "No se ha encontrado el empleado",
 "Error",
 JOptionPane.ERROR_MESSAGE);
 }
 em.close();
 }
}

```

La salida:



## 7.6. Actividad 37 - Conceptos Básicos JPA

### Instrucciones

#### [Classroom](#)

Nos piden crear una aplicación para gestionar un sistema de noticias en la empresa mediante JPA. De cada Noticia se debe guardar la siguiente información:

- **id**: Debe ser autogenerado por el sistema y además ser el campo clave de la tabla correspondiente
- **titular**: Obligatorio y único (VARCHAR2 de 120)
- **fecha**: Obligatoria
- **sinopsis**: Una descripción de la noticia
- **URL**: La URL de la noticia si la hay

Se pide:

- Configurar la aplicación para que use JPA (con la opción de actualizar automáticamente la estructura de la BBDD) y almacene la información en la tabla `noti` de la BBDD.
- Diseñar una aplicación con un menú con las siguientes opciones:

Gestión de Noticias	1. Nueva Noticia	2. Quitar Noticia	3. Buscar Noticias	4. Ver Todas las Noticias	5. Salir
Opción [1-5]:					

#### Nueva Noticia

- Se pedirá el `titular`, la `fecha` (dd/mm/aaaa), la `sinopsis` y la `URL` de la noticia y se creará una entidad `Noticia` con dicha información.
- La cadena de fecha se intentará convertir en una fecha mediante un `DateFormat` de formato corto (`parse`)
- Si hay errores, se asignará la fecha del día
- Se persistirá la noticia en la BBDD
- Se preguntará al usuario si quiere introducir otra noticia

#### Quitar Noticia

- Se pedirá el `titular` de la noticia y se buscará en la BBDD
- Si la **noticia no existe**, se dará un mensaje de **error**
- Si existe
- Se mostrarán el resto de los campos de la noticia para pedir confirmación al usuario para su eliminación
- Si confirma se eliminará la noticia

#### Buscar Noticias

- Se pedirá el texto a buscar y se **buscarán todas las noticias que incluyan dicho texto en su `titular`** (podéis usar el operador `Like` exactamente igual que en Oracle)
- Si hay alguna noticia se mostrarán en una tabla en un `JOptionPane`
- Si no la hay, se dará un mensaje indicándolas

#### Ver todas las Noticias

- Se mostrarán todas las noticias **ordenadas en descendente** por `fecha`.

## 7.7. Relaciones entre Entidades OneToMany y ManyToOne

### Relaciones entre Entidades en JPA

Uno de los principales problemas a la hora de mapear una estructura basada en objetos a tablas es cómo gestionar la relación entre objetos.

Como ya sabemos las posibles opciones son 1 a 1 (OneToOne), 1 a muchos (OneToMany), muchos a 1 (ManyToOne) y muchos a muchos (ManyToMany). Aunque todos

estos tipos de relaciones pueden ser gestionados por Hibernate, nos centraremos en las relaciones de uno a varios dado que son las más importantes:

Vamos a pensar en la relación Empleado gestiona Proyecto. Es una relación de 1 a n en la que un Empleado puede gestionar múltiples proyectos, pero cada proyecto es gestionado por un único empleado.

A la hora de representar esta relación en JPA podemos definir dos campos:

- Una colección de Proyectos (`List<Proyecto>`) en la entidad Empleado que representa los proyectos que gestiona el empleado. En este caso tenemos una relación OneToMany de Empleados a Proyectos
- Un Empleado en la entidad Proyecto que es el responsable del proyecto. En este caso tenemos una relación de ManyToOne de Proyectos a Empleados

#### @OneToMany

A la hora de indicar una relación de uno a varios podemos añadir los siguientes atributos:

- `mappedBy`: Obligatorio. Representa el campo de la entidad relacionada que es el propietario de la relación. En nuestro ejemplo, el campo de la entidad Proyecto que representa el empleado que es responsable del proyecto
- `cascade`: Opcional. Indica qué operaciones en cascada deben realizarse en la entidad relacionada. Puede ser: REMOVE (las entidades relacionadas se eliminan al eliminar la entidad principal), MERGE (similar para modificaciones / inserciones), PERSIST (lo mismo para inserciones), DETACH (si se deja de gestionar la entidad principal se dejan de gestionar las relacionadas), REFRESH (si se hace un refresh de la entidad principal, se hace de las entidades relacionadas), ALL (todas las anteriores). Por defecto, ninguna de las anteriores.
- `fetch`: Opcional. Indica si, al recuperar una entidad se deben recuperar las entidades relacionadas (EAGER) o si dichas entidades se deben recuperar al acceder a ellas (LAZY). Por defecto: LAZY
- `orphanRemoval`: Opcional. Indica que no queremos hijos no relacionados con algún padre. Si tenemos Cascade.REMOVE, al eliminar un empleado se eliminarán sus proyectos pero, si ponemos la lista de proyectos del empleado a null, todos sus proyectos permanecerían con un valor null en el campo responsable. Sin embargo, si ponemos orphanRemoval a true, poner los proyectos gestionados por un empleado a null implicaría eliminar dichos proyectos. Por defecto: false
- `targetEntity`: Opcional. La entidad relacionada. Si la colección está definida con genéricos (por ejemplo `List<Proyecto>`) no hace falta. Por defecto: void.class

#### @ManyToOne

En la parte n de la relación indicamos la anotación @ManyToOne que puede tener los atributos cascade, fetch, targetEntity (suelen indicarse en la parte 1) y, además:

- `optional`: Si la relación es opcional, es decir, si pueden existir proyectos sin responsable (true). Por defecto true

#### @JoinColumn

Adicionalmente, puede ser necesario especificar el campo de la tabla que es la clave extranjera de la relación mediante la anotación @JoinColumn que tiene los siguientes atributos:

- `name`: El nombre de la columna en la tabla. Si no es específica se entiende que es NombreCampo\_ClaveEntidadReferenciada. Por defecto "".
- `referencedColumnName`: El nombre del campo referenciado. Si no se indica, la clave primaria de la entidad referenciada. Por defecto ""
- `columnDefinition`: La definición de la columna. Por defecto ""
- `insertable`: Si debe incluirse en las instrucciones INSERT. Por defecto true.
- `updatable`: Si debe incluirse en las instrucciones UPDATE. Por defecto true.
- `unique`: Si la clave es única. Por defecto: false
- `nullable`: Si puede ser nula. Por defecto: true
- `table`: En qué tabla está el campo. Si no se indica se asume que en la tabla primaria. Por defecto ""

#### Ejemplo

Vamos a modificar la entidad Empleado para que sea la parte uno de una relación con la entidad Proyecto:

```
@Entity
@Table(name = "pemp")
public class Empleado implements Serializable {

 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 private Integer id;
 ...
 @OneToMany(mappedBy = "responsable")
 private List<Proyecto> proyectosGestionados;
 ...
 public List<Proyecto> getProyectosGestionados() {
 return proyectosGestionados;
 }
 public void setProyectosGestionados(List<Proyecto> proyectosGestionados) {
 this.proyectosGestionados = proyectosGestionados;
 }
 ...
}
```

La entidad Proyecto:

```
@Entity
@Table(name="pproy")
public class Proyecto implements Serializable{

 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 private Long id;

 private String nombre;

 private Date fechaInicio;

 private Date fechaFin;

 @ManyToOne
 @JoinColumn(name="idResponsable")
 private Empleado responsable;

 public Proyecto() {
 }

 public String getNombre() {
 return nombre;
 }

 public void setNombre(String nombre) {
 this.nombre = nombre;
 }

 public Date getFechaInicio() {
 return fechaInicio;
 }

 public void setFechaInicio(Date fechaInicio) {
 this.fechaInicio = fechaInicio;
 }

 public Date getFechaFin() {
 return fechaFin;
 }

 public void setFechaFin(Date fechaFin) {
 this.fechaFin = fechaFin;
 }

 public Empleado getResponsable() {
 return responsable;
 }

 public void setResponsable(Empleado responsable) {
 this.responsable = responsable;
 }

 public Long getId() {
 return id;
 }

 public void setId(Long id) {
 this.id = id;
 }

 @Override
 public String toString() {
 return "Proyecto{" + "id=" + id + ", nombre=" + nombre + ", fechaInicio=" + fechaInicio + ", fechaFin=" + fechaFin + ", responsable=" + responsable
 }
}
```

Dado que tenemos la opción de modificación del DDL, vamos a actualizar la información en la BBDD simplemente activando el EntityManager:

```
public class GestionProyectos {

 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 em.close();
 }
}
```

En la salida:

Hibernate:

```
create table pproj (
 id number(19,0) not null,
 fechaFin date,
 fechaInicio date,
 nombre varchar2(255),
 idResponsable number(10,0),
 primary key (id)
)
```

Hibernate:

```
alter table pproy
 add constraint FKpbjdw7ptbfwl9pgakkundefqt
 foreign key (idResponsable)
 references pemp
```

Como se puede ver, se crea la tabla pproy con una clave extranjera sobre pemp con el nombre que hemos dado a la columna en el @JoinColumn (si no lo hubiéramos especificado, sería RESPONSABLE\_ID).

## 7.8. Trabajo con Entidades Relacionadas

### Añadir información

Si queremos añadir información de proyectos a la BBDD podemos persistir un objeto Proyecto asignándole previamente el Empleado correspondiente:

```
public class GestionProyectos {

 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 em.getTransaction().begin();
 Empleado e = em.find(Empleado.class, 1);
 Proyecto p = new Proyecto();
 p.setNombre("Desarrollo Aplicación Web para INDRA");
 p.setFechaInicio(new GregorianCalendar(2020,4,20).getTime());
 p.setFechaFin(new GregorianCalendar(2020,4,20).getTime());
 p.setResponsable(e);
 em.persist(p);
 p = new Proyecto();
 p.setNombre("Curso JPA Empleados");
 p.setFechaInicio(new GregorianCalendar(2020,6,1).getTime());
 p.setFechaFin(new GregorianCalendar(2020,7,1).getTime());
 p.setResponsable(e);
 em.persist(p);
 em.getTransaction().commit();
 for(Proyecto proy : e.getProyectosGestionados()){
 System.out.println(proy);
 }
 em.close();
 }
}
```

La salida:

```
Hibernate:
select
 empleado0_.id as id1_1_0,
 empleado0_.apellidos as apellidos2_1_0,
 empleado0_.fecha_alta as fecha_alta3_1_0,
 empleado0_.nombre as nombre4_1_0
from
 pemp empleado0_
where
 empleado0_.id=?
Hibernate:
select
 hibernate_sequence.nextval
from
 dual
Hibernate:
insert
into
 pproy
 (fechaFin, fechaInicio, nombre, idResponsable, id)
values
 (?, ?, ?, ?, ?)

Hibernate:
insert
into
 pproy
 (fechaFin, fechaInicio, nombre, idResponsable, id)
values
 (?, ?, ?, ?, ?)

Hibernate:
select
 proyectosg0_.idResponsable as idresponsable5_2_0_,
 proyectosg0_.id as id1_2_0_,
 proyectosg0_.id as id1_2_1_,
 proyectosg0_.fechafin as fechafin2_2_1_,
 proyectosg0_.fechainicio as fechainicio3_2_1_,
 proyectosg0_.nombre as nombre4_2_1_,
 proyectosg0_.idResponsable as idresponsable5_2_1_
from
 pproy proyectosg0_
where
 proyectosg0_.idResponsable=?
```

Proyecto{id=26, nombre=Curso JPA Empleados, fechaInicio=Wed Jul 01 00:00:00 CEST 2020, fechaFin=Sat Aug 01 00:00:00 CEST 2020, responsable=López Marin, Ana  
 Proyecto{id=22, nombre=Desarrollo Aplicación Web para INDRA, fechaInicio=Wed May 20 00:00:00 CEST 2020, fechaFin=Wed May 20 00:00:00 CEST 2020, responsable=}

Como se puede ver, al obtener la entidad `Empleado` ya podemos acceder a todos los proyectos que gestiona.

NOTA: Dado que, por defecto la lista de proyectos tiene como fetch tipo `LAZY`, los datos de los proyectos no se recuperan hasta que no se ejecuta el método `getProyectos()`.

Dado que ahora tenemos la relación establecida entre `Empleado` y `Proyecto` podemos **navegar de una entidad a otra a través de los campos relacionados**. Por ejemplo, dado un proyecto, podemos acceder a todos los proyectos del mismo empleado con el siguiente código:

```
public class GestionProyectos {

 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 em.getTransaction().begin();
 Query q = em.createQuery("Select p From Proyecto p "
 + "where upper(p.nombre) like :proyecto");
 String nombre = JOptionPane.showInputDialog("Nombre Proyecto");

 q.setParameter("proyecto", "%" + nombre.toUpperCase() + "%");
 List<Proyecto> lp = q.getResultList();
 if (lp.size() == 0) {
 JOptionPane.showMessageDialog(null,
 "No existe el proyecto");
 } else {
 Proyecto p = lp.get(0);
 String listado = "<html><h2>Responsable : " +
 p.getResponsable() + "</h2>" +
 "<table border=1><tr><th>#</th><th>Proyecto</th>" +
 "<th>Fecha Inicio</th><th>Fecha Fin</th></tr>" +
 "DateFormat df = DateFormat.getDateInstance();
 for (Proyecto proy : p.getResponsable().getProyectosGestionados()) {
 listado += "<tr><td>" + proy.getId() + "</td>" +
 "<td>" + proy.getNombre() + "</td>" +
 "<td>" + df.format(proy.getFechaInicio()) + "</td>" +
 "<td>" + df.format(proy.getFechaFin()) + "</td></tr>";
 }
 JOptionPane.showMessageDialog(null, listado);
 }
 em.close();
 }
 }
}
```

La ejecución :



Evidentemente, la modificación y eliminación de proyectos se realizaría de la misma forma.

### 7.9. Actividad 38 - Creación de relaciones

#### Instrucciones

En la aplicación de `Noticias` se quiere añadir la posibilidad de **agrupar dichas noticias en temas** (política, economía, deportes, etc). Se pide diseñar la entidad `Tema` y las relaciones necesarias para poder gestionar las noticias y sus temas. Para el tema añadir la opción de que, si se elimina un tema, se eliminan automáticamente todas las noticias de dicho tema (pista: hay que añadir en algún sitio el atributo `CascadeType.DELETE`) Tras ello, realizar los siguientes cambios en la aplicación:

- Añadir la opción de **añadir / eliminar temas**. En el caso de eliminar un tema, comprobar que se eliminan todas las noticias del mismo
- Modificar la aplicación para que, a la hora de añadir una noticia, se pida también el tema.
- Modificar la aplicación para que, en el listado de todas las noticias, aparezca el tema
- Añadir la opción de **Ver Noticias por Tema**. En esta opción se deberá mostrar un listado de todos los temas, junto con las noticias del mismo:

```
Tema : tema
Noticia URL Fecha
noticia URL dd/mm/aaaa
Hay n noticias de tema
```

Tema: tema

...

- Añadir la opción de **buscar noticias a partir de una fecha dada**. Se pedirá la fecha y se mostrarán las noticias (con sus temas) a partir de dicha fecha de modo que aparezcan primero las más recientes.
- Modificar el apartado de buscar una noticia por título para que, tras mostrar la noticia, muestre el resto de noticias del tema de la noticia encontrada.

### 7.10. Relaciones entre Entidades - ManyToMany

Cuando tenemos en JPA una relación de varios a varios podemos emplear la anotación `@ManyToMany` siempre y cuando no necesitemos información adicional en la relación. En nuestro ejemplo, queremos especificar los empleados que trabajan en los proyectos. En este sentido tenemos una relación de varios a varios, dado que un empleado puede trabajar en múltiples proyectos y en el mismo proyecto pueden trabajar varios empleados. Como ya sabemos, una relación varios a varios implica la creación de una tabla intermedia (`join table`) cuya clave primaria es la concatenación de las claves primarias de las tablas relacionadas. JPA puede crear esa tabla por nosotros pero, si queremos tener más control sobre dicha tabla, podemos emplear la anotación `@JoinTable`. Veamos en primer lugar a solución más simple:

En Empleado

```
@Entity
@Table(name = "pemp")
public class Empleado implements Serializable {

 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 private Integer id;

 @Column(columnDefinition = "VARCHAR2(50)")
 @Basic(optional = false)
 private String apellidos;

 private String nombre;

 @Column(name="fecha_alta")
 private Date fechaAlta;

 @OneToMany(mappedBy = "responsable")
 private List<Proyecto> proyectosGestionados;

 @ManyToMany
 private List<Proyecto> proyectosParticipa;
 ...

 public List<Proyecto> getProyectosParticipa() {
 return ProyectosParticipa;
 }

 public void setProyectosParticipa(List<Proyecto> ProyectosParticipa) {
 this.ProyectosParticipa = ProyectosParticipa;
 }
 ...
}
```

En Proyecto

```
@Entity
@Table(name="pproy")
public class Proyecto implements Serializable {

 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 private Long id;

 private String nombre;

 private Date fechaInicio;

 private Date fechaFin;

 @ManyToOne
 @JoinColumn(name="idResponsable")
 private Empleado responsable;

 @ManyToMany(mappedBy = "proyectosParticipa")
 private List<Empleado> participantes;

 public List<Empleado> getParticipantes() {
 return participantes;
 }

 public void setParticipantes(List<Empleado> participantes) {
 this.participantes = participantes;
 }
 ...
}
```

En este caso, consideramos como dueña de la relación a la entidad `Empleado` por lo que ponemos el `mappedBy` en `Proyecto`. Al ejecutar la aplicación se ejecutan las siguientes sentencias SQL:

Hibernate:

```
create table pemp_pproy (
 participantes_id number(10,0) not null,
 proyectosParticipa_id number(19,0) not null
)
Hibernate:
alter table pemp_pproy
 add constraint FKal4sk1pyb9vkxreospojacy44
 foreign key (proyectosParticipa_id)
```

```

 references pproy
Hibernate:
 alter table pemp_pproy
 add constraint FKmi02l8nn617ht3to8hwxj1r3t
 foreign key (participantes_id)
 references pemp

```

Como se puede ver, JPA crea la tabla intermedia y añade las dos claves extranjeras necesarias. Si queremos controlar los nombres de las tablas / columnas que se crean podemos emplear la anotación `@JoinTable` en `Empleado`:

```

@ManyToMany
@JoinTable(name = "empleadosenproyectos",
 joinColumns = @JoinColumn(name = "empId"),
 inverseJoinColumns = @JoinColumn(name = "proyId"))
private List<Proyecto> proyectosParticipa;

```

La tabla creada ahora:

```

create table empleadosenproyectos(
 empId number(10,0) not null,
 proyId number(19,0) not null
)

```

## Añadir información

Para añadir información debemos hacerlo desde el propietario de la relación (en este caso la entidad `Empleado`):

```

public class GestionProyectos {

 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 em.getTransaction().begin();
 Proyecto proyecto = em.find(Proyecto.class, 25L);
 Empleado e = em.find(Empleado.class, 2);
 e.getProyectosParticipa().add(proyecto);
 em.merge(e);
 e = em.find(Empleado.class, 3);
 e.getProyectosParticipa().add(proyecto);
 em.merge(e);
 em.getTransaction().commit();
 for(Empleado emp : proyecto.getParticipantes()){
 System.out.println(emp);
 }
 em.close();
 }
}

```

En la salida podemos ver:

```

...
Hibernate:
 insert
 into
 empleadosenproyectos
 (empId, proyId)
 values
 (?, ?)
Hibernate:
 insert
 into
 empleadosenproyectos
 (empId, proyId)
 values
 (?, ?)
Hibernate:
 select
 participan0_.proyId as proyid2_1_0_,
 participan0_.empId as empid1_1_0_,
 empleado1_.id as id1_2_1_,
 empleado1_.apellidos as apellidos2_2_1_,
 empleado1_.fecha_alta as fecha_alta3_2_1_,
 empleado1_.nombre as nombre4_2_1_
 from
 empleadosenproyectos participan0_,
 pemp empleado1_
 where
 participan0_.empId=empleado1_.id
 and participan0_.proyId=?

De Miguel López, Carlos

Simón Tercilla, Natalia

```

## Eliminar Información

Si queremos eliminar la relación entre un empleado y un proyecto, debemos hacerlo desde el empleado (dado que es el propietario de la relación). Para ello basta con quitar el proyecto de la lista de proyectos del empleado y modificar su información:

---

```

public class GestionProyectos {
 ...
 ...
}

```

```

public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 em.getTransaction().begin();
 Empleado e = em.find(Empleado.class, 2);
 e.getProyectosParticipa().remove(0);
 em.merge(e);
 em.getTransaction().commit();
 em.close();
}

}

```

La salida:

```

Hibernate:
 select
 empleado0_.id as id1_2_0_,
 empleado0_.apellidos as apellidos2_2_0_,
 empleado0_.fechaAlta as fechaAlta3_2_0_,
 empleado0_.nombre as nombre4_2_0_
 from
 pemp empleado0_
 where
 empleado0_.id=?
Hibernate:
 select
 proyectosp0_.empId as empid1_1_0_,
 proyectosp0_.proyId as proyid2_1_0_,
 proyecto1_.id as id1_3_1_,
 proyecto1_.fechaFin as fechafin2_3_1_,
 proyecto1_.fechaInicio as fechainicio3_3_1_,
 proyecto1_.nombre as nombre4_3_1_,
 proyecto1_.idResponsable as idresponsable5_3_1_,
 empleado2_.id as id1_2_2_,
 empleado2_.apellidos as apellidos2_2_2_,
 empleado2_.fechaAlta as fechaAlta3_2_2_,
 empleado2_.nombre as nombre4_2_2_
 from
 empleadosenproyectos proyectosp0_,
 pproj proyecto1_,
 pemp empleado2_
 where
 proyectosp0_.proyId=proyecto1_.id
 and proyecto1_.idResponsable=empleado2_.id(+)
 and proyectosp0_.empId=?
Hibernate:
 delete
 from
 empleadosenproyectos
 where
 empId=?

```

### 7.11. Relaciones entre Entidades - OneToOne

Las relaciones uno a uno pueden implementarse en una BBDD de dos maneras

- Con una clave extranjera
- Compartiendo la clave primaria

#### Con clave extranjera

En este caso, la tabla dependiente tendrá una restricción foreign key respecto a la tabla principal. Supongamos que queremos guardar la información de RRHH del empleado en una tabla adicional de modo que, al buscar usuarios no tengamos que recuperar un montón de información que sólo es relevante en contadas ocasiones. Es evidente que la relación entre el empleado y su información en esta tabla (curriculum, conocimientos, familiares...) es una relación de uno a uno. En nuestro caso vamos a crear una nueva entidad, llamada RRHH con un único campo que será el CV (además, lógicamente del id). En nuestro caso, asumiremos que la entidad principal será Empleado y RRHH será lo que se denomina el lado propietario (owning side) de la relación (el lado propietario es quien posee la restricción foreign key). Esto implica que, en la tabla RRHH es donde se almacenará el idEmpleado y la que llevará la anotación @JoinColumn. Por tanto, en la entidad Empleado:

```

@Entity
@Table(name = "pemp")
public class Empleado implements Serializable{
 ...
 @OneToOne(mappedBy = "empleado")
 private RRHH rrhh;

 public RRHH getRrhh() {
 return rrhh;
 }

 public void setRrhh(RRHH rrhh) {
 this.rrhh = rrhh;
 }
}

```

Y la entidad RRHH:

```
@Entity
```

```

public class RRHH {
 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 private Integer id;

 @OneToOne
 @JoinColumn(name="idEmpleado")
 private Empleado empleado;

 public Empleado getEmpleado() {
 return empleado;
 }

 public void setEmpleado(Empleado empleado) {
 this.empleado = empleado;
 }

 @Column(length = 100000)
 private String CV;

 public RRHH() {
 }

 public Integer getId() {
 return id;
 }

 public void setId(Integer id) {
 this.id = id;
 }

 public String getCV() {
 return CV;
 }

 public void setCV(String CV) {
 this.CV = CV;
 }
}

```

Veamos como podemos asociar un objeto RRHH a un empleado:

```

public class GestionProyectos {

 public static void main(String[] args) {
 EntityManager em = Persistence.createEntityManagerFactory("JPAPU")
 .createEntityManager();
 em.getTransaction().begin();
 RRHH rrhh = new RRHH();
 rrhh.setCV("Un currículum enorme!!!!");
 Empleado e = em.find(Empleado.class, 1);
 rrhh.setEmpleado(e);
 em.persist(rrhh);
 em.getTransaction().commit();
 em.close();
 }
}

```

La salida:

```

Hibernate:

create table RRHH (
 id number(10,0) not null,
 CV long,
 idEmpleado number(10,0),
 primary key (id)
)
Hibernate:

 alter table RRHH
 add constraint FKpha52uo2w98glgqmguo93kqf2
 foreign key (idEmpleado)
 references pemp
 ...

Hibernate:
 select
 empleado0_.id as id1_2_0_,
 empleado0_.apellidos as apellidos2_2_0_,
 empleado0_.fecha_alta as fecha_alta3_2_0_,
 empleado0_.nombre as nombre4_2_0_,
 rrhh1_.id as id1_4_1_,
 rrhh1_.CV as cv2_4_1_,
 rrhh1_.idEmpleado as idempleado3_4_1_
 from
 pemp empleado0_,
 RRHH rrhh1_
 where
 empleado0_.id=rrhh1_.idEmpleado(+)
 and empleado0_.id=?
```

```

Hibernate:
 select
 hibernate_sequence.nextval
 from
 dual
Hibernate:
 insert
 into
 RRHH
 (CV, idEmpleado, id)
 values
 (?, ?, ?)

```

Si comprobamos la tabla:

The screenshot shows the Oracle SQL Developer interface. At the top, there is a connection bar with the text "Connection: jdbc:oracle:thin:@localhost:1521:ORCL [scott on SCOTT]". Below it, a large text area contains the SQL command "SELECT \* FROM SCOTT.RRHH". To the left of this text area, there are two numbered lines: "1" and "2". Below the text area, there is a preview pane titled "SELECT \* FROM SCOTT.RRHH... X" with a table header row and one data row.

#	ID	CV	IDEMPLEADO
1		29 Un curriculum enorme!!!	1

### Compartiendo clave primaria (@MapsId)

La segunda alternativa pasa por emplear para la tabla `RRHH` la misma clave primaria que para la tabla `Empleado`, de modo que el `id` de `RRHH` coincida con el `id` del `Empleado`. En la entidad `RRHH` basta con etiquetar el campo con la anotación `@MapsId`.

```

@Entity
public class RRHH {
 @Id
 @GeneratedValue(strategy = GenerationType.SEQUENCE)
 private Integer id;

 @OneToOne
 @MapsId
 private Empleado empleado;

 public Empleado getEmpleado() {
 return empleado;
 }

 public void setEmpleado(Empleado empleado) {
 this.empleado = empleado;
 }

 @Column(length = 100000)
 private String CV;

 public RRHH() {
 }

 public Integer getId() {
 return id;
 }

 public void setId(Integer id) {
 this.id = id;
 }

 public String getCV() {
 return CV;
 }

 public void setCV(String CV) {
 this.CV = CV;
 }
}

```

Eliminamos la tabla `RRHH` del paso anterior y ejecutamos de nuevo la aplicación. La salida:

```

create table RRHH (
 CV long,
 empleado_id number(10,0) not null,
 primary key (empleado_id)
)
Hibernate:

alter table RRHH
add constraint FKo9503g2rmqt6oflj852yu556k
foreign key (empleado_id)
references pemp
...

```

Hibernate:

```

select
 empleado0_.id as id1_2_0_,
 empleado0_.apellidos as apellidos2_2_0_,
 empleado0_.fecha_alta as fecha_alta3_2_0_,
 empleado0_.nombre as nombre4_2_0_,
 rrhh1_.empleado_id as empleado_id2_4_1_,
 rrhh1_.CV as cv1_4_1_
from
 pemp empleado0_,
 RRHH rrhh1_
where
 empleado0_.id=rrhh1_.empleado_id(+)
 and empleado0_.id=?
Hibernate:
insert
into
 RRHH
 (CV, empleado_id)
values
 (?, ?)

```

Si comprobamos la tabla:

Connection: jdbc:oracle:thin:@localhost:1521:ORCL [scott on SCOTT]

1	SELECT * FROM SCOTT.RRHH
2	

SELECT \* FROM SCOTT.RRHH ... X

#	CV	EMPLEADO_ID
1	Un currículum enorme!!!	1

## 8. Proyecto TMDB

### 8.1. Librerías y Encriptación

#### Acceso al API de The Movie Database

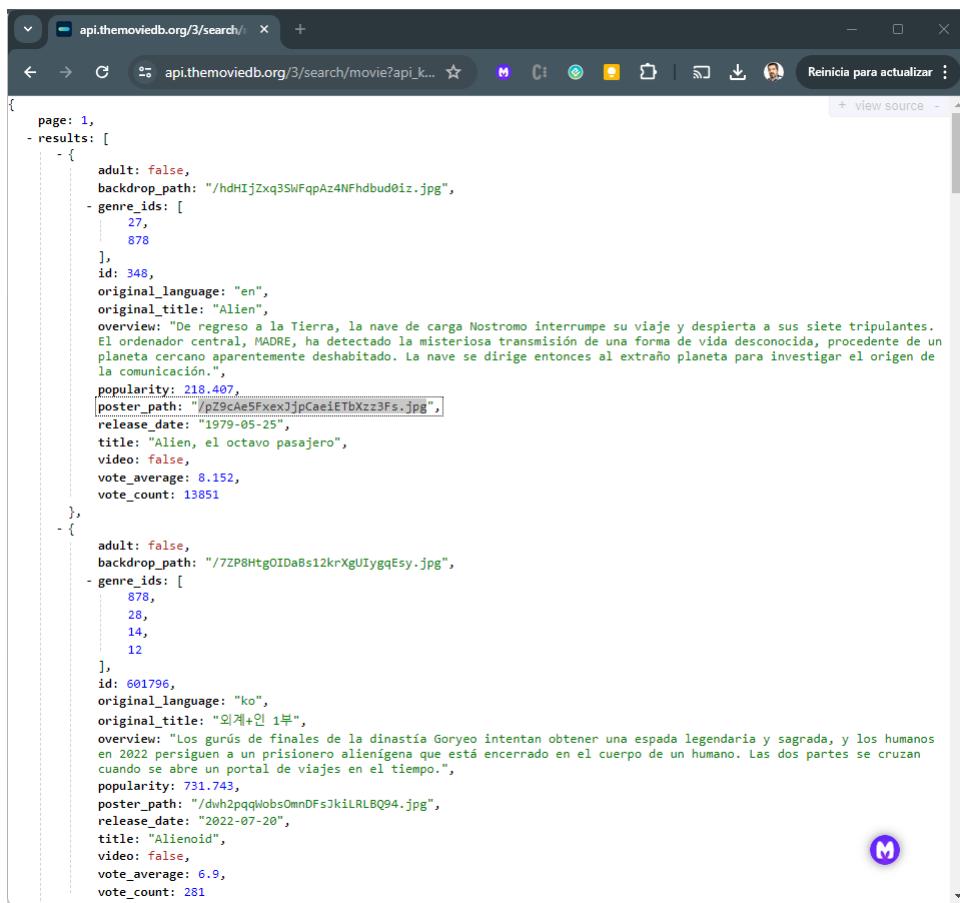
Vamos a diseñar una aplicación multiventana que nos va a permitir trabajar con información sobre películas proporcionadas por la página <https://www.themoviedb.org/> que contiene información sobre películas y programas de televisión.

En nuestro caso, nos vamos a centrar en las películas y vamos a hacer una aplicación que permita a diferentes usuarios crear listas de películas favoritas que almacenaremos en una BBDD.

Para acceder a la información de la BBDD necesitamos acceder al API (Application Program Interface) de la página web que está en <https://developer.themoviedb.org/docs/getting-started> y, para ello, deberemos registrarnos y obtener una clave de usuario (API KEY).

Un API no es más que una forma de acceder a funcionalidades proveidas por una aplicación. En el caso de servicios en Internet el acceso se debe hacer a través de HTTP y normalmente la información enviada y recibida se encapsula en objetos JSON (Javascript Simple Object Notation) que habrá que convertir a objetos de Java.

Veamos un ejemplo de acceso a la URL [https://api.themoviedb.org/3/search/movie?api\\_key=a425ed51317322081d09a80b9ba971a9&query=Alien&language=es](https://api.themoviedb.org/3/search/movie?api_key=a425ed51317322081d09a80b9ba971a9&query=Alien&language=es):



```
{
 page: 1,
 results: [
 {
 adult: false,
 backdrop_path: "/hdH1jZxq3SwFqpAz4NFhdbud0iz.jpg",
 genre_ids: [
 27,
 878
],
 id: 348,
 original_language: "en",
 original_title: "Alien",
 overview: "Durante su regreso a la Tierra, la nave de carga Nostromo interrumpe su viaje y despierta a sus siete tripulantes. El ordenador central, MADRE, ha detectado la misteriosa transmisión de una forma de vida desconocida, procedente de un planeta cercano aparentemente deshabitado. La nave se dirige entonces al extraño planeta para investigar el origen de la comunicación.",
 popularity: 218.407,
 poster_path: "/pZ9cae5FhexJjpCaeiETbXzz3Fs.jpg",
 release_date: "1979-05-25",
 title: "Alien, el octavo pasajero",
 video: false,
 vote_average: 8.152,
 vote_count: 13851
 },
 {
 adult: false,
 backdrop_path: "/7ZP8HtgOIDaBs12krXgUIygqEsy.jpg",
 genre_ids: [
 878,
 28,
 14,
 12
],
 id: 601796,
 original_language: "ko",
 original_title: "외계인 1부",
 overview: "Los gurús de finales de la dinastía Goryeo intentan obtener una espada legendaria y sagrada, y los humanos en 2022 persiguen a un prisionero alienígena que está encerrado en el cuerpo de un humano. Las dos partes se cruzan cuando se abre un portal de viajes en el tiempo.",
 popularity: 731.743,
 poster_path: "/dvh2pqWobsOmnDFsJkiLRBQ94.jpg",
 release_date: "2022-07-20",
 title: "Alienoid",
 video: false,
 vote_average: 6.9,
 vote_count: 281
 }
]
}
```

¿Cómo podemos acceder al API desde Java? Necesitamos la clase `HttpURLConnection` que nos permite realizar peticiones Http. Además, dado que la respuesta será un JSON, necesitamos una librería para poder acceder a la información devuelta:

```
<dependency>
 <groupId>org.json</groupId>
 <artifactId>json</artifactId>
 <version>20240303</version>
</dependency>
```

La aplicación:

```
import java.awt.Dimension;
import org.json.JSONArray;
import org.json.JSONObject;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.UnsupportedEncodingException;
import java.net.HttpURLConnection;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.URL;
import java.net.URLEncoder;
import java.util.logging.Level;
```

```

import java.util.logging.Logger;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;

public class MovieSearch {
 private static final String API_KEY = "a425ed51317322081d09a80b9ba971a9";
 private static final String NO_IMAGEN = "https://upload.wikimedia.org/wikipedia/commons/c/c2/No_image_poster.png";

 public static void main(String[] args) throws UnsupportedEncodingException {
 String pelicula = JOptionPane.showInputDialog("Película");
 pelicula = URLEncoder.encode(pelicula, "UTF-8");
 try {
 // Construye la URL de la solicitud
 URL url = new URI(
 "https://api.themoviedb.org/3/search/movie?api_key=" +
 API_KEY + "&query=" + pelicula + "&language=es"
).toURL();
 System.out.println(url.toString());
 // Abre la conexión
 HttpURLConnection connection = (HttpURLConnection) url.openConnection();
 connection.setRequestMethod("GET");

 // Lee la respuesta
 BufferedReader reader = new BufferedReader(
 new InputStreamReader(
 connection.getInputStream()
)
);
 StringBuilder response = new StringBuilder();
 String line;
 while ((line = reader.readLine()) != null) {
 response.append(line);
 }
 reader.close();

 // Procesa la respuesta JSON
 JSONObject jsonResponse = new JSONObject(response.toString());
 JSONArray results = jsonResponse.getJSONArray("results");
 if (results.length() > 0) {
 String listado = """
<html><h1>Resultados</h1>
<table border=1 bgcolor=white>
<tr><th></th><th>Título</th><th>Fecha Lanzamiento</th><th>Resumen</th></tr>
""";
 // Muestra los resultados de la búsqueda
 for (int i = 0; i < results.length(); i++) {
 JSONObject movie = results.getJSONObject(i);
 String imagen = movie.get("backdrop_path").toString();
 if (imagen.equalsIgnoreCase("null")){
 imagen = NO_IMAGEN;
 } else {
 imagen = "https://image.tmdb.org/t/p/w92/" + movie.get("poster_path").toString();
 }
 String titulo = movie.getString("title");
 String fecha = movie.getString("release_date");
 String resumen = movie.getString("overview");

 listado += "<tr><td></td>" +
 "<td>" + titulo + "</td>" +
 "<td>" + fecha + "</td>" +
 "<td width='500px>" + resumen + "</td></tr>";
 }
 listado +=
"""
</table>
</html>
""";
 JScrollPane jsp = new JScrollPane(new JLabel(listado));
 jsp.setPreferredSize(new Dimension(1250,700));
 JOptionPane.showMessageDialog(null, jsp,
 "Búsqueda : " + pelicula,
 JOptionPane.PLAIN_MESSAGE);
 } else {
 JOptionPane.showMessageDialog(
 null,
 "No se han encontrado resultados",
 "No Encontrado",
 JOptionPane.ERROR_MESSAGE);
 }
 // Cierra la conexión
 connection.disconnect();
 } catch (IOException e) {
 e.printStackTrace();
 } catch (URISyntaxException ex) {
 Logger.getLogger(MovieSearch.class.getName()).log(Level.SEVERE, null, ex);
 }
 }
}

```

Si, por ejemplo, introducimos Alien:

Búsqueda : Alien

**Resultados**

	Título	Fecha Lanzamiento	Resumen
	Alien, el octavo pasajero	1979-05-25	De regreso a la Tierra, la nave de carga Nostromo interrumpe su viaje y despierta a sus siete tripulantes. El ordenador central, MAIDEN, ha detectado la misteriosa transmisión de una forma de vida desconocida, procedente de un planeta cercano aparentemente deshabitado. La nave se dirige entonces al extraño planeta para investigar el origen de la comunicación.
	Alienoid	2022-07-20	Los gurús de finales de la dinastía Goryeo intentan obtener una espada legendaria y sagrada, y los humanos en 2022 persiguen a un prisionero alienígena que está encerrado en el cuerpo de un humano. Las dos partes se cruzan cuando se abre un portal de viajes en el tiempo.
	Aliens: El regreso	1986-07-18	Alien es un organismo perfecto, una máquina de matar cuya superioridad física sólo puede competir con su agresividad. La oficial Ripley y la tripulación de la nave "Nostromo" se habían enfrentado, en el pasado, a esa monstruosa criatura. Y sólo Ripley sobrevivió a la masacre. Después de vagar por el espacio durante varios años, Ripley fue rescatada. Durante ese tiempo, el planeta de Alien ha sido colonizado. Pero, de repente, se pierde la comunicación con la colonia y, para investigar los motivos, se envía una expedición de marines espaciales, capitaneados por Ripley. Allí les esperan miles de espeluznantes criaturas. Alien se ha reproducido y esta vez la lucha es por la supervivencia de la Humanidad.
	Alien	2017-05-16	

OK

NOTA: Existen librerías que simplifican el código para acceder a APIs de Internet como por ejemplo <https://kong.github.io/unirest-java/>

Dado que tenemos acceso al API, podríamos crear métodos para acceder a los diferentes endpoints y crear clases y objetos para manipular esa información. En cualquier caso, existen librerías que ya han hecho ese trabajo por nosotros.

En nuestro caso, vamos a emplear la librería <https://github.com/c-eg/themoviedbapi> que podemos incluir en nuestro pom.xml con:

```
<dependency>
 <groupId>uk.co.conoregan</groupId>
 <artifactId>themoviedbapi</artifactId>
 <version>2.0.4</version>
</dependency>
```

El mismo ejemplo anterior, pero con la librería:

```
import info.movito.themoviedbapi.TmdbApi;
import info.movito.themoviedbapi.model.core.Movie;
import info.movito.themoviedbapi.model.core.MovieResultsPage;
import info.movito.themoviedbapi.tools.TmdbException;
import java.awt.Dimension;
import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
```

/\*\*  
 \*  
 \* @author ichueca  
 \*/

```
public class TMDB {
 private static String API_KEY="eyJhbGciOiJIUzI1NiJ9eyJhdWQiOiJhNDI1ZWQ1MTMxNzMyMjA4MWQwOWE4MGJ5YmE5NzFhOSIsInIYiI6IjVjNWFhNTYzYzNhMzY4M2NkNTg3NTYwZSI:

 private static final String NO_IMAGEN = "https://upload.wikimedia.org/wikipedia/commons/c/c2/No_image_poster.png";
```

```
 public static void main(String[] args) throws TmdbException {
 try {
 TmdbApi tmdbApi = new TmdbApi(API_KEY);

 String pelicula = JOptionPane.showInputDialog("Pelicula");
 pelicula = URLEncoder.encode(pelicula, "UTF-8");

 MovieResultsPage results = tmdbApi.getSearch().searchMovie(pelicula, Boolean.TRUE, "es", null, 0, "es", null);
 if (results.getResults().size() > 0) {
 String listado = """
 <html><h1>Resultados</h1>
 <table border=1 bgcolor=white>
 <tr><th></th><th>Titulo</th><th>Fecha Lanzamiento</th><th>Resumen</th></tr>
 """;
 // Muestra los resultados de la búsqueda
 for (Movie movie : results.getResults()) {
 String imagen = movie.getPosterPath(); // movie.get("backdrop_path").toString();
 if (imagen == null) {
 imagen = NO_IMAGEN;
 } else {
 imagen = "https://image.tmdb.org/t/p/w92/" + movie.getPosterPath();
 }
 String titulo = movie.getTitle();
 String fecha = movie.getReleaseDate();
 String resumen = movie.getOverview();

```

```

 listado += "<tr><td></td>" +
 "<td>" + titulo + "</td>" +
 "<td>" + fecha + "</td>" +
 "<td width='500px'>" + resumen + "</td></tr>";
 }
 listado +=
 """
 </table>
 </html>
 """;
 JScrollPane jsp = new JScrollPane(new JLabel(listado));
 jsp.setPreferredSize(new Dimension(1250,700));
 JOptionPane.showMessageDialog(null, jsp,
 "Búsqueda : " + pelicula,
 JOptionPane.PLAIN_MESSAGE);
} else {
 JOptionPane.showMessageDialog(
 null,
 "No se han encontrado resultados",
 "No Encontrado",
 JOptionPane.ERROR_MESSAGE);
}
} catch (UnsupportedEncodingException ex) {
 Logger.getLogger(TMDB.class.getName()).log(Level.SEVERE, null, ex);
}
}
}

```

Evidentemente, la salida es la misma.

## Encriptación

Encriptar contraseñas es esencial para proteger la seguridad de los usuarios en aplicaciones y sistemas informáticos. Sin encriptación, las contraseñas almacenadas en una base de datos son vulnerables a ser interceptadas por atacantes en caso de una brecha de seguridad. Utilizar un algoritmo de encriptación robusto, como BCrypt, ayuda a mitigar este riesgo al convertir las contraseñas en una cadena irreconocible y única. Además, el uso de un "salto" (salt) añade una capa adicional de seguridad al generar un valor aleatorio único para cada contraseña encriptada, lo que dificulta aún más el proceso de descifrado mediante ataques de fuerza bruta o tablas de arco iris. BCrypt es una excelente opción para encriptar contraseñas en Java debido a su resistencia a los ataques de fuerza bruta y su capacidad para ajustar la complejidad del algoritmo. A continuación, se muestra un ejemplo de cómo encriptar y verificar una contraseña utilizando BCrypt en Java con la biblioteca jBCrypt:

Vamos a crear una clase de utilidad que emplearemos en nuestro proyecto:

```

import org.mindrot.jbcrypt.BCrypt;

/**
 *
 * @author IñigoChueca
 */
public class PasswordManager {
 // Método para encriptar una contraseña
 public static String encriptarContraseña(String contraseña) {
 // Generar un salto aleatorio
 String salto = BCrypt.gensalt();
 // Encriptar la contraseña con el salto generado
 String contraseñaEncriptada = BCrypt.hashpw(contraseña, salto);
 return contraseñaEncriptada;
 }

 // Método para verificar una contraseña encriptada
 public static boolean verificarContraseña(String contraseña, String contraseñaEncriptada) {
 // Utilizar BCrypt para verificar la contraseña
 return BCrypt.checkpw(contraseña, contraseñaEncriptada);
 }

 public static void main(String[] args) {
 String contraseña = "12345";
 String encriptada = PasswordManager.encriptarContraseña(contraseña);
 System.out.println(encriptada);

 String encriptada2 = PasswordManager.encriptarContraseña(contraseña);
 System.out.println(encriptada2);

 System.out.println(PasswordManager.verificarContraseña(contraseña, encriptada));
 System.out.println(PasswordManager.verificarContraseña(contraseña, encriptada2));
 }
}

```

La salida:

```

$2a$10$jKX9aPyyLG3hnM6Gxktkra.KyBSa9/sry3lbFYI8T19oPDkedlrl1YW
$2a$10$zA9FNPj1SB1JNDZJsM/YEu2Vcjc3uzT3kgf6fbnkQGa.xCio/fnm
true
true

```

Podemos ver que, aunque hemos empleado la misma contraseña en los dos casos, la contraseña encriptada es distinta. Esto se debe a que cada una emplea un salto diferente. ¿Y cómo sabe BCrypt qué salto debe emplear? Porque se almacena junto con la contraseña (son los primeros caracteres de la misma).

### 8.2. Login y Registro de Usuarios - DAO y Servicio

En primer lugar vamos a hacer la parte de la aplicación que se va a ejecutar en primer lugar y que se va a encargar de logear a los usuarios. La ventana de login también permitirá registrarse en la BBDD.

## Modelo

### Usuario

```
package org.zabalburu.tmdb.modelo;

/**
 *
 * @author IñigoChueca
 */
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class Usuario {
 @EqualsAndHashCode.Include
 private Integer id;
 private String nombre;
 private String apellido;
 private String usuario;
 private String password;
 private String email;
 private String imagen; // Ruta al fichero de imagen
}
```

### DAO

#### UsuarioDAO

```
import java.util.List;
import org.zabalburu.tmdb.modelo.Usuario;

public interface UsuarioDAO {
 Usuario getUsuario(Integer id);
 Usuario getUsuario(String usuario);
 Usuario nuevoUsuario(Usuario usuario);
 List<Usuario> getUsuarios();
}
```

#### UsuarioImpl

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import org.zabalburu.tmdb.exceptions.UsuarioExisteException;
import org.zabalburu.tmdb.modelo.Usuario;
import org.zabalburu.util.Conexion;
import org.zabalburu.util.PasswordManager;

/**
 *
 * @author IñigoChueca
 */
public class UsuarioImpl implements UsuarioDAO {

 @Override
 public Usuario getUsuario(Integer id) {
 Usuario usuarioEncontrado = null;
 try {
 Connection connection = Conexion.getConexion();
 String query = "SELECT * FROM usuarios WHERE id = ?";
 try (PreparedStatement pstmt = connection.prepareStatement(query)) {
 pstmt.setInt(1, id);
 try (ResultSet rs = pstmt.executeQuery()) {
 if (rs.next()) {
 usuarioEncontrado = crearUsuario(rs);
 }
 }
 }
 } catch (SQLException e) {
 e.printStackTrace();
 }
 return usuarioEncontrado;
 }

 @Override
 public Usuario getUsuario(String usuario) {
 Usuario usuarioEncontrado = null;
 try {
 Connection connection = Conexion.getConexion();
 String query = "SELECT * FROM usuarios WHERE lower(usuario) = lower(?)";
 try (PreparedStatement pstmt = connection.prepareStatement(query)) {
 pstmt.setString(1, usuario);
 try (ResultSet rs = pstmt.executeQuery()) {

```

```

 if (rs.next()) {
 usuarioEncontrado = crearUsuario(rs);
 }
 }
} catch (SQLException e) {
 e.printStackTrace();
}
return usuarioEncontrado;
}

@Override
public Usuario nuevoUsuario(Usuario usuario) throws UsuarioExisteException{
try {
 if (getUsuario(usuario.getUsuario())!=null){
 throw new UsuarioExisteException(usuario.getUsuario());
 }
 Connection cnn = Conexion.getConexion();
 ResultSet rst = cnn.createStatement().executeQuery("Select max(id) maximo From usuarios");
 rst.next();
 Integer id = rst.getInt("maximo") + 1;
 rst.close();
 usuario.setId(id);
 String query = "INSERT INTO usuarios (id, nombre, apellido, usuario, password, email, imagen) VALUES (?, ?, ?, ?, ?, ?, ?)";
 PreparedStatement pstmt = cnn.prepareStatement(query);
 pstmt.setInt(1, usuario.getId());
 pstmt.setString(2, usuario.getNombre());
 pstmt.setString(3, usuario.getApellido());
 pstmt.setString(4, usuario.getUsuario());
 pstmt.setString(5, PasswordManager.encriptarContraseña(usuario.getPassword()));
 pstmt.setString(6, usuario.getEmail());
 pstmt.setString(7, usuario.getImagen());
 pstmt.executeUpdate();

} catch (SQLException e) {
 e.printStackTrace();
}
return usuario;
}

@Override
public List<Usuario> getUsuarios() {
List<Usuario> usuarios = new ArrayList<>();
try {
 Connection connection = Conexion.getConexion();
 String query = "SELECT * FROM usuarios";
 try (PreparedStatement pstmt = connection.prepareStatement(query); ResultSet rs = pstmt.executeQuery()) {
 while (rs.next()) {
 Usuario usuario = crearUsuario(rs);
 usuarios.add(usuario);
 }
 }
} catch (SQLException e) {
 e.printStackTrace();
}
return usuarios;
}

private Usuario crearUsuario(ResultSet rs) throws SQLException {
 Usuario usuario = new Usuario();
 usuario.setId(rs.getInt("id"));
 usuario.setNombre(rs.getString("nombre"));
 usuario.setApellido(rs.getString("apellido"));
 usuario.setUsuario(rs.getString("usuario"));
 usuario.setPassword(rs.getString("password"));
 usuario.setEmail(rs.getString("email"));
 usuario.setImagen(rs.getString("imagen"));
 return usuario;
}
}

```

#### **UsuarioExisteException**

```

public class UsuarioExisteException extends RuntimeException{
 public UsuarioExisteException(String usuario){
 super("El usuario " + usuario + " ya está registrado. Utilice otro identificador de usuario");
 }
}

```

#### **Servicio**

##### **TMDBServicio (Singleton)**

```

import java.util.List;
import org.zabalburu.tmdb.dao.UsuarioDAO;
import org.zabalburu.tmdb.dao.UsuarioImpl;
import org.zabalburu.tmdb.modelo.Usuario;

/**
 *
 * @author IñigoChueca
 */

```

```

public class TMDBService {
 private static UsuarioDAO usuarioDAO = new UsuarioImpl();
 private static final TMDBService service = new TMDBService();

 private TMDBService() {
 }

 public static final TMDBService getService() {
 return service;
 }

 public Usuario getUsuario(Integer id){
 return usuarioDAO.getUsuario(id);
 }

 public Usuario getUsuario(String usuario){
 return usuarioDAO.getUsuario(usuario);
 }

 public Usuario nuevoUsuario(Usuario usuario){
 return usuarioDAO.nuevoUsuario(usuario);
 }

 public Usuario login(String usuario, String contraseña){
 Usuario u = usuarioDAO.getUsuario(usuario);
 if (u != null) {
 if (!PasswordManager.verificarContraeña(contraseña, u.getPassword())){
 u = null;
 }
 }
 return u;
 }

 public List<Usuario> getUsuarios(){
 return usuarioDAO.getUsuarios();
 }
}

```

### 8.3. Test DAO y Servicio

¿Cómo podríamos testear el dao de usuarios?. Tenemos un problema dado que queremos comprobar el funcionamiento de operaciones de escritura / eliminación y consulta y no podemos emplear la BBDD de producción dado que podríamos provocar efectos colaterales además de que la información es variable.

#### Base de Datos en Memoria (Derby)

Una solución pasa por emplear una BBDD distinta y habitualmente se emplea una BBDD en memoria que simplifica el uso de recursos. En nuestro caso vamos a emplear derby aunque hay unas cuantas más. Para ello, añadimos la dependencia:

```

<dependency>
 <groupId>org.apache.derby</groupId>
 <artifactId>derby</artifactId>
 <version>10.17.1.0</version>
 <scope>test</scope>
</dependency>

```

Para obtener una conexión en memoria: `cnn = DriverManager.getConnection("jdbc:derby:memory:tests;create=true");` Esta BBDD se eliminará automáticamente cuando finalice la aplicación.

#### Mockito

Pero tenemos otro problema. Nuestros métodos del DAO emplean la clase Conexión que establece una conexión con la BBDD de Oracle y nosotros necesitaremos que la conexión se haga sobre la BBDD de derby. Cambiar la clase Conexión o crear un método nuevo para las pruebas no es operativo dado que tendríamos que estar cambiando de una a otra constantemente. La solución pasa por emplear un mock. Los mocks nos permiten aislar los tests de las dependencias que pudieran tener en el exterior (en este caso, de la BBDD). La idea es crear un objeto que sustituya a otro cuando se invoque. En nuestro caso, la solución sería crear un objeto que intercepte las llamadas a Conexión y retorne otro objeto conectado a nuestra BBDD en memoria. De este modo, cuando testemos la aplicación, las llamadas del DAO al método getConexion de la clase Conexión se redirigirán a la clase mockeada. Para emplear mocks, la librería más utilizada es Mockito y tenemos que incluirla en nuestro pom:

```

<dependency>
 <groupId>org.mockito</groupId>
 <artifactId>mockito-core</artifactId>
 <version>5.11.0</version>
 <scope>test</scope>
</dependency>

```

A la hora de mockear una clase, hay que trabajar de diferente modo si queremos mockear métodos estáticos o no. En nuestro caso, vamos a mockear un método estático así que el proceso pasa por crear un objeto MockedStatic : `MockedStatic conn = Mockito.mockStatic(Conexion.class);` Tras ello, mockeamos los métodos estáticos deseados: `conn.when(() -> Conexion.getConexion()).thenReturn(cnn);` Como se ve, se pasa una expresión lambda con el método a mockear y lo que queremos retornar (en este caso la conexión a Derby). NOTA: En el caso de métodos no estáticos, lo único que cambia es que, en vez de MockedStatic emplearíamos Mocked. NOTA: Es importante cerrar el objeto mockeado antes de cada test.

#### Proceso

Dado que los tests deben ser independientes entre sí, vamos a tener la siguiente funcionalidad:

- Antes de empezar todos los test vamos a establecer la conexión con Derby, eliminar la tabla de usuarios (si existiera) y crearla de nuevo
- Antes de empezar cada test vamos a crear un par de usuarios y a preparar el método mockeado

- Cuando finalice cada test vamos a eliminar todos los usuarios de la tabla y liberar el mock
- Cuando finalicen todos los tests cerraremos la conexión a derby

La clase de Test:

```
package org.zabalburu.tmdb.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.List;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;
import org.zabalburu.tmdb.exceptions.UsuarioExisteException;

import org.zabalburu.tmdb.modelo.Usuario;
import org.zabalburu.util.Conexion;
import org.zabalburu.util.PasswordManager;

/**
 *
 * @author IñigoChueca
 */
public class UsuarioImplTest {

 private static Connection cnn;
 private static UsuarioDAO dao = new UsuarioImpl();
 MockedStatic<Conexion> conn;

 public UsuarioImplTest() {
 }

 @BeforeAll
 public static void setUpClass() throws Exception {
 cnn = DriverManager.getConnection("jdbc:derby:memory:tests;create=true");
 try {
 cnn.createStatement()
 .executeUpdate("Drop table usuarios");
 } catch (SQLException ex) {}
 cnn.createStatement()
 .executeUpdate(
 """
 Create Table Usuarios(
 id integer,
 nombre varchar(100),
 apellido varchar(100),
 usuario varchar(100),
 password varchar(100),
 email varchar(100),
 imagen varchar(100)
);
 """
);
 }

 @AfterAll
 public static void tearDownClass() throws Exception {
 cnn.close();
 }

 @BeforeEach
 public void setUp() throws Exception {
 PreparedStatement pst = cnn.prepareStatement("""
 insert into Usuarios
 values (?, ?, ?, ?, ?, ?, ?)
 """);
 pst.setInt(1, 1);
 pst.setString(2, "Juan");
 pst.setString(3, "López");
 pst.setString(4, "lopez");
 pst.setString(5, PasswordManager.encriptarContraseña("12345"));
 pst.setString(6, "jlopez@gmail.com");
 pst.setString(7, "noimage");
 pst.executeUpdate();
 pst.setInt(1, 2);
 pst.setString(2, "Ana");
 pst.setString(3, "Sanz");
 pst.setString(4, "asanz");
 pst.setString(5, PasswordManager.encriptarContraseña("12345"));
 pst.setString(6, "asanz@gmail.com");
 pst.setString(7, "noimage");
 pst.executeUpdate();
 pst.close();
 conn = Mockito.mockStatic(Conexion.class);
 conn.when(() -> Conexion.getConexion()).thenReturn(cnn);
 }
}
```

```

private void deleteAll() throws SQLException{
 cnn.createStatement().executeUpdate("delete from usuarios where 1=1");
}

@AfterEach
public void tearDown() throws Exception {
 deleteAll();
 conn.close();
}

/**
 * Test of getUsuario method, of class UsuarioImpl.
 */
@Test
public void testGetUsuario_Integer() throws SQLException {
 Usuario juan = dao.getUsuario(1);
 assertEquals(juan.getNombre(), "Juan");
 assertNull(dao.getUsuario(10));
}

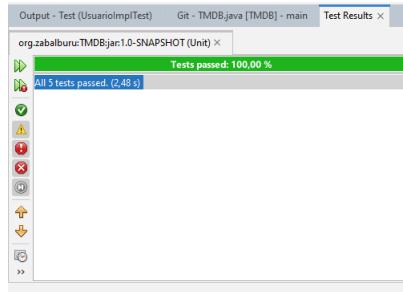
/**
 * Test of getUsuario method, of class UsuarioImpl.
 */
@Test
public void testGetUsuario_String() {
 Usuario juan = dao.getUsuario(1);
 assertEquals(juan.getNombre(), "Juan");
 assertNull(dao.getUsuario("jsimon"));
}

/**
 * Test of nuevoUsuario method, of class UsuarioImpl.
 */
@Test
public void testNuevoUsuario() {
 Usuario nuevo = new Usuario(0, "nuevo", "nuevo", "nuevo", PasswordManager.encriptarContraseña("12345"), "nuevo", "nuevo");
 assertDoesNotThrow(() -> dao.nuevoUsuario(nuevo));
 assertEquals(nuevo.getId(), 3);
 assertEquals(dao.getUsuarios().size(), 3);
 assertThrows(UsuarioExisteException.class, () -> {
 dao.nuevoUsuario(new Usuario(0, "", "", "nuevo", "", "", ""));
 });
}

/**
 * Test of getUsuarios method, of class UsuarioImpl.
 */
@Test
public void testGetUsuarios() throws SQLException{
 List<Usuario> usuarios = dao.getUsuarios();
 assertEquals(usuarios.size(), 2);
 assertEquals(usuarios.get(0).getUsuario(), "jlopez");
 deleteAll();
 usuarios = dao.getUsuarios();
 assertNotNull(usuarios);
 assertEquals(usuarios.size(),0);
}
}

```

Si ejecutamos el test:



NOTA: Los tests se pasan automáticamente cada vez que hacemos un Build

## Testeando el Servicio

En el caso del servicio, ahora nos interesa mockear el DAO dado que asumimos que lo testamos en otro sitio. Pero necesitamos que el servicio emplee el DAO mockeado y no el original. Para ello, vamos a añadir métodos set / get al DAO en el servicio:

```

public static UsuarioDAO getUsuarioDAO() {
 return usuarioDAO;
}

public static void setUsuarioDAO(UsuarioDAO usuarioDAO) {
 TMDBService.usuarioDAO = usuarioDAO;
}

```

El test es similar al anterior pero cambia un poco en el uso del objeto mockeado:

```
package org.zabalburu.tmdb.service;

import java.util.ArrayList;
import java.util.List;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.Assertions;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.zabalburu.tmdb.dao.UsuarioImpl;
import org.zabalburu.tmdb.modelo.Usuario;
import org.zabalburu.util.PasswordManager;

/**
 *
 * @author IñigoChueca
 */
public class TMDBServiceTest {
 private static UsuarioImpl mock;
 private static TMDBService servicio = TMDBService.getService();
 public TMDBServiceTest() {
 }

 @BeforeAll
 public static void setUpClass() {
 }

 @AfterAll
 public static void tearDownClass() {
 }

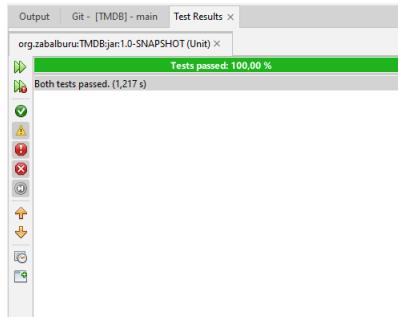
 @BeforeEach
 public void setUp() {
 mock = Mockito.mock(UsuarioImpl.class);
 TMDBService servicio = TMDBService.getService();
 TMDBService.setUsuarioDAO(mock);
 }

 @AfterEach
 public void tearDown() {
 }

 /**
 * Test of isDisponibleEmail method, of class TMDBService.
 */
 @Test
 public void testIsDisponibleEmail() {
 List<Usuario> usuarios = new ArrayList<>();
 usuarios.add(new Usuario(1,null,null,null,"jlopez@gmail.com",null));
 Mockito.when(mock.getUsuarios()).thenReturn(usuarios);
 assertTrue(servicio.isDisponibleEmail("csimon@gmail.com"));
 assertFalse(servicio.isDisponibleEmail("jlopez@gmail.com"));
 }

 /**
 * Test of login method, of class TMDBService.
 */
 @Test
 public void testLogin() {
 Usuario usuario = new Usuario(1,null,null,"jlopez",PasswordManager.encriptarContraseña("12345"),null,null);
 Mockito.when(mock.getUsuario("jlopez")).thenReturn(usuario);
 Mockito.when(mock.getUsuario("alopez")).thenReturn(null);
 Assertions.assertEquals(servicio.login("jlopez", "12345"), usuario);
 }
}
```

El test debería salir correctamente:



Como hemos comentado, si hay tests definidos, estos se ejecutarán cada vez que se haga un build de la aplicación:

```

T E S T S

Running org.zabalburu.tmdb.dao.UsuarioImplTest
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
WARNING: A Java agent has been loaded dynamically (C:\Users\IñigoChueca\.m2\repository\net\bytebuddy\byte-buddy-agent\1.14.12\byte-buddy-agent-1.14.12.jar)
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.123 s -- in org.zabalburu.tmdb.dao.UsuarioImplTest

Running org.zabalburu.tmdb.service.TMDBServiceTest
SLF4J(W): No SLF4J providers were found.
SLF4J(W): Defaulting to no-operation (NOP) logger implementation
SLF4J(W): See https://www.slf4j.org/codes.html#noProviders for further details.
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.427 s -- in org.zabalburu.tmdb.service.TMDBServiceTest

Results:

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
```

#### 8.4. Login y Registro de Usuarios - Vistas

Vamos a crear ahora la ventana de login.

##### FrmLogin.java

```
package org.zabalburu.tmdb.views;

import java.awt.Color;
import javax.swing.Icon;
import javax.swing.JOptionPane;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;
import jiconfont.swing.IconFontSwing;
import org.zabalburu.tmdb.modelo.Usuario;
import org.zabalburu.tmdb.service.TMDBService;

/**
 *
 * @author IñigoChueca
 */
public class FrmLogin extends javax.swing.JFrame {
 private final TMDBService service = TMDBService.getService();
 /**
 * Creates new form FrmLogin
 */
 public FrmLogin() {

 initComponents();
 Icon icon = IconFontSwing.buildIcon(GoogleMaterialDesignIcons.PERSON_ADD, 80, new Color(0,153,153));
 lblLogo.setIcon(icon);
 lblError.setText("");
 }

 /**
 * This method is called from within the constructor to initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is always
 * regenerated by the Form Editor.
 */
 @SuppressWarnings("unchecked")
 // <editor-fold defaultstate="collapsed" desc="Generated Code">
 private void initComponents() {
 java.awt.GridBagConstraints gridBagConstraints;

 pnlLogo = new javax.swing.JPanel();
 filler1 = new javax.swing.Box.Filler(new java.awt.Dimension(0, 0), new java.awt.Dimension(0, 0), new java.awt.Dimension(0, 32767));
 lblLogo = new javax.swing.JLabel();
 filler2 = new javax.swing.Box.Filler(new java.awt.Dimension(0, 0), new java.awt.Dimension(0, 0), new java.awt.Dimension(0, 32767));
 pnlUsuario = new javax.swing.JPanel();
 lblUsuario = new javax.swing.JLabel();
 txtUsuario = new javax.swing.JTextField();
 pwdPassword = new javax.swing.JPasswordField();
 btnEntrar = new javax.swing.JButton();
 btnSalir = new javax.swing.JButton();
 lblError = new javax.swing.JLabel();
 lblRegistro = new javax.swing.JLabel();

 setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
 setTitle("Login");

 pnlLogo.setBorder(javax.swing.BorderFactory.createEmptyBorder(0, 40, 0, 40));
 pnlLogo.setLayout(new javax.swing.BoxLayout(pnlLogo, javax.swingBoxLayout.Y_AXIS));
 pnlLogo.add(filler1);

 lblLogo.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
 pnlLogo.add(lblLogo);
 pnlLogo.add(filler2);

 getContentPane().add(pnlLogo, java.awt.BorderLayout.LINE_START);
 }
}
```

```

pnlUsuario.setBorder(javax.swing.BorderFactory.createEmptyBorder(0, 40, 0, 40));
pnlUsuario.setLayout(new java.awt.GridBagLayout());

lblUsuario.setFont(new java.awt.Font("Segoe UI", 0, 16)); // NOI18N
lblUsuario.setText("Identifíquese ");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(lblUsuario, gridBagConstraints);

txtUsuario.addKeyListener(new java.awt.event.KeyAdapter() {
 public void keyPressed(java.awt.event.KeyEvent evt) {
 pwdPasswordKeyPressed(evt);
 }
});
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 1;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
gridBagConstraints.weightx = 1.0;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(txtUsuario, gridBagConstraints);

pwdPassword.addKeyListener(new java.awt.event.KeyAdapter() {
 public void keyPressed(java.awt.event.KeyEvent evt) {
 pwdPasswordKeyPressed(evt);
 }
});
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 2;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(pwdPassword, gridBagConstraints);

btnEntrar.setText("Entrar");
btnEntrar.addActionListener(new java.awt.event.ActionListener() {
 public void actionPerformed(java.awt.event.ActionEvent evt) {
 btnEntrarActionPerformed(evt);
 }
});
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 3;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(btnEntrar, gridBagConstraints);

btnSalir.setText("Salir");
btnSalir.addActionListener(new java.awt.event.ActionListener() {
 public void actionPerformed(java.awt.event.ActionEvent evt) {
 btnSalirActionPerformed(evt);
 }
});
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 1;
gridBagConstraints.gridy = 3;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(btnSalir, gridBagConstraints);

lblError.setFont(new java.awt.Font("Segoe UI", 0, 18)); // NOI18N
lblError.setForeground(new java.awt.Color(255, 0, 0));
lblError.setText("Usuario / password erróneos");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 5;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.insets = new java.awt.Insets(10, 15, 0, 0);
pnlUsuario.add(lblError, gridBagConstraints);

lblRegistro.setFont(new java.awt.Font("Segoe UI", 0, 16)); // NOI18N
lblRegistro.setForeground(new java.awt.Color(0, 153, 153));
lblRegistro.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
lblRegistro.setText("¿Todavía no tiene usuario? Regístrate");
lblRegistro.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
lblRegistro.addMouseListener(new java.awt.event.MouseAdapter() {
 public void mousePressed(java.awt.event.MouseEvent evt) {
 lblRegistroMousePressed(evt);
 }
});
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 4;
gridBagConstraints.gridy = 4;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.insets = new java.awt.Insets(15, 15, 15, 15);

```

```

pnlUsuario.add(lblRegistro, gridBagConstraints);

getContentPane().add(pnlUsuario, java.awt.BorderLayout.CENTER);

setSize(new java.awt.Dimension(583, 436));
setLocationRelativeTo(null);
}// </editor-fold>

private void btnEntrarActionPerformed(java.awt.event.ActionEvent evt) {
 String user = txtUsuario.getText();
 String pwd = new String(pwdPassword.getPassword());
 if (user.isBlank() || pwd.isBlank()) {
 lblError.setText("Debe especificarse el usuario y la contraseña");
 return;
 }
 Usuario usuario = service.login(user, pwd);
 if (usuario == null){
 lblError.setText("Usuario / password erróneos");
 } else {
 JOptionPane.showMessageDialog(this, "Usuario Conectado Correctamente!");
 System.exit(0);
 }
}

private void pwdPasswordKeyPressed(java.awt.event.KeyEvent evt) {
 lblError.setText("");
}

private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
 System.exit(0);
}

private void lblRegistroMousePressed(java.awt.event.MouseEvent evt) {
 new FrmRegistro().setVisible(true);
 this.dispose();
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
 /* Set the Nimbus look and feel */
 /* Set the Nimbus look and feel */
 /*@editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) "*/
 /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
 * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
 */
 try {
 for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.UIManager.getInstalledLookAndFeels()) {
 if ("Nimbus".equals(info.getName())) {
 javax.swing.UIManager.setLookAndFeel(info.getClassName());
 break;
 }
 }
 } catch (ClassNotFoundException ex) {
 java.util.logging.Logger.getLogger(FrmLogin.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
 } catch (InstantiationException ex) {
 java.util.logging.Logger.getLogger(FrmLogin.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
 } catch (IllegalAccessException ex) {
 java.util.logging.Logger.getLogger(FrmLogin.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
 } catch (javax.swing.UnsupportedLookAndFeelException ex) {
 java.util.logging.Logger.getLogger(FrmLogin.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
 }
 // </editor-fold>

 /* Create and display the form */
 java.awt.EventQueue.invokeLater(new Runnable() {
 public void run() {
 new FrmLogin().setVisible(true);
 }
 });
}

// Variables declaration - do not modify
private javax.swing.JButton btnEntrar;
private javax.swing.JButton btnSalir;
private javax.swing.Box.Filler filler1;
private javax.swing.Box.Filler filler2;
private javax.swing.JLabel lblError;
private javax.swing.JLabel lblLogo;
private javax.swing.JLabel lblRegistro;
private javax.swing.JLabel lblUsuario;
private javax.swing.JPanel pnlLogo;
private javax.swing.JPanel pnlUsuario;
private javax.swing.JPasswordField pwdPassword;
private javax.swing.JTextField txtUsuario;
// End of variables declaration
}

```

Y lo llamamos desde la aplicación principal:

### TMDB.java

```
package org.zabalburu.tmdb;
```

```

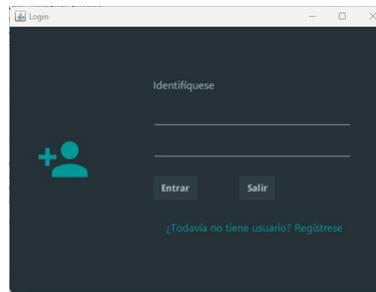
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeel;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;
import jiconfont.swing.IconFontSwing;
import mdlaf.MaterialLookAndFeel;
import mdlaf.themes.MaterialOceanicTheme;
import org.zabalburu.tmdb.views.FrmLogin;

/**
 *
 * @author IñigoChueca
 */
public class TMDB {

 public static void main(String[] args) {
 try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new MaterialOceanicTheme()));
 IconFontSwing.register(GoogleMaterialDesignIcons.getIconFont());
 } catch (UnsupportedLookAndFeelException ex) {
 Logger.getLogger(TMDB.class.getName()).log(Level.SEVERE, null, ex);
 }
 new FrmLogin().setVisible(true);
 }
}

```

La salida:



El formulario de registro:

```

package org.zabalburu.tmdb.views;

import java.awt.Color;
import java.awt.Container;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.imageio.ImageIO;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;
import javax.swing.filechooser.FileFilter;
import javax.swing.filechooser.FileNameExtensionFilter;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;
import jiconfont.swing.IconFontSwing;
import org.zabalburu.tmdb.modelo.Usuario;
import org.zabalburu.tmdb.service.TMDBService;

/**
 *
 * @author IñigoChueca
 */
public class FrmRegistro extends javax.swing.JFrame {

 private final TMDBService service = TMDBService.getService();
 private String imagen = "no_image.jpg";

 /**
 * Creates new form FrmLogin
 */
 public FrmRegistro() {
 initComponents();
 Icon icon = IconFontSwing.buildIcon(GoogleMaterialDesignIcons.PERSON_ADD, 80, new Color(0, 153, 153));
 lblLogo.setIcon(icon);
 lblFoto.setIcon(new ImageIcon("images/" + imagen));
 }
}

```

```

 lblError.setText(" ");
 btnGuardar.setEnabled(false);
 pwdPassword.getDocument().addDocumentListener(listener);
 pwdPassword.getDocument().putProperty("origen", pwdPassword);
 pwdPassword2.getDocument().addDocumentListener(listener);
 pwdPassword2.getDocument().putProperty("origen", pwdPassword);
 txtUsuario.getDocument().addDocumentListener(listener);
 txtUsuario.getDocument().putProperty("origen", txtUsuario);
 txtEmail.getDocument().addDocumentListener(listener);
 txtEmail.getDocument().putProperty("origen", txtEmail);
 }

 /**
 * This method is called from within the constructor to initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is always
 * regenerated by the Form Editor.
 */
 @SuppressWarnings("unchecked")
 // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-BEGIN:initComponents
 private void initComponents() {
 java.awt.GridBagConstraints gridBagConstraints;

 pnlLogo = new javax.swing.JPanel();
 filler1 = new javax.swing.Box.Filler(new java.awt.Dimension(0, 0), new java.awt.Dimension(0, 0), new java.awt.Dimension(0, 32767));
 lblLogo = new javax.swing.JLabel();
 filler2 = new javax.swing.Box.Filler(new java.awt.Dimension(0, 0), new java.awt.Dimension(0, 0), new java.awt.Dimension(0, 32767));
 pnlUsuario = new javax.swing.JPanel();
 lblRegistrese = new javax.swing.JLabel();
 txtEmail = new javax.swing.JTextField();
 btnGuardar = new javax.swing.JButton();
 btnSalir = new javax.swing.JButton();
 lblError = new javax.swing.JLabel();
 lblEmail = new javax.swing.JLabel();
 lblApellidos = new javax.swing.JLabel();
 lblNombre = new javax.swing.JLabel();
 txtNombre = new javax.swing.JTextField();
 lblPassword2 = new javax.swing.JLabel();
 pwdPassword2 = new javax.swing.JPasswordField();
 lblPassword = new javax.swing.JLabel();
 pwdPassword = new javax.swing.JPasswordField();
 txtApellidos = new javax.swing.JTextField();
 lblUsuario = new javax.swing.JLabel();
 txtUsuario = new javax.swing.JTextField();
 lblFoto = new javax.swing.JLabel();
 lblObligatorio = new javax.swing.JLabel();
 lblObligatorio2 = new javax.swing.JLabel();
 lblObligatorio3 = new javax.swing.JLabel();
 lblObligatorio4 = new javax.swing.JLabel();

 setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
 setTitle("Login");

 pnlLogo.setBorder(javax.swing.BorderFactory.createEmptyBorder(0, 40, 0, 40));
 pnlLogo.setLayout(new javax.swing.BoxLayout(pnlLogo, javax.swing.BoxLayout.Y_AXIS));
 pnlLogo.add(filler1);

 lblLogo.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
 pnlLogo.add(lblLogo);
 pnlLogo.add(filler2);

 getContentPane().add(pnlLogo, java.awt.BorderLayout.LINE_START);

 pnlUsuario.setBorder(javax.swing.BorderFactory.createEmptyBorder(0, 40, 0, 40));
 pnlUsuario.setLayout(new java.awt.GridBagLayout());

 lblRegistrese.setFont(new java.awt.Font("Segoe UI", 0, 16)); // NOI18N
 lblRegistrese.setText("Registrese");
 gridBagConstraints = new java.awt.GridBagConstraints();
 gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
 gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
 gridBagConstraints.insets = new java.awt.Insets(14, 0, 14, 14);
 pnlUsuario.add(lblRegistrese, gridBagConstraints);
 gridBagConstraints = new java.awt.GridBagConstraints();
 gridBagConstraints.gridx = 1;
 gridBagConstraints.gridy = 4;
 gridBagConstraints.gridwidth = 2;
 gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
 gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
 gridBagConstraints.weightx = 1.0;
 gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
 pnlUsuario.add(txtEmail, gridBagConstraints);

 btnGuardar.setText("Guardar");
 btnGuardar.addActionListener(new java.awt.event.ActionListener() {
 public void actionPerformed(java.awt.event.ActionEvent evt) {
 btnGuardarActionPerformed(evt);
 }
 });
 gridBagConstraints = new java.awt.GridBagConstraints();
 gridBagConstraints.gridx = 1;
 gridBagConstraints.gridy = 8;
 gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
 gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);

```

```

pnlUsuario.add(btnGuardar, gridBagConstraints);

btnSalir.setText("Salir");
btnSalir.addActionListener(new java.awt.event.ActionListener() {
 public void actionPerformed(java.awt.event.ActionEvent evt) {
 btnSalirActionPerformed(evt);
 }
});
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 2;
gridBagConstraints.gridy = 8;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(btnSalir, gridBagConstraints);

lblError.setFont(new java.awt.Font("Segoe UI", 0, 18)); // NOI18N
lblError.setForeground(new java.awt.Color(255, 0, 0));
lblError.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
lblError.setText("z");
lblError.setToolTipText("");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 10;
gridBagConstraints.gridwidth = 3;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.insets = new java.awt.Insets(10, 15, 0, 0);
pnlUsuario.add(lblError, gridBagConstraints);

lblEmail.setText("Email");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 4;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
pnlUsuario.add(lblEmail, gridBagConstraints);

lblApellidos.setText("Apellidos");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
pnlUsuario.add(lblApellidos, gridBagConstraints);

lblNombre.setText("Nombre");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 1;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
pnlUsuario.add(lblNombre, gridBagConstraints);
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 1;
gridBagConstraints.gridy = 1;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
gridBagConstraints.weightx = 1.0;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(txtNombre, gridBagConstraints);

lblPassword2.setText("Repetir Contraseña");
lblPassword2.setPreferredSize(new java.awt.Dimension(150, 16));
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 6;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.insets = new java.awt.Insets(0, 0, 0, 9);
pnlUsuario.add(lblPassword2, gridBagConstraints);
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 1;
gridBagConstraints.gridy = 6;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(pwdPassword2, gridBagConstraints);

lblPassword.setText("Contraseña");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 5;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
pnlUsuario.add(lblPassword, gridBagConstraints);
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 1;
gridBagConstraints.gridy = 5;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(pwdPassword, gridBagConstraints);
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 1;
gridBagConstraints.gridy = 2;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);

```

```

pnlUsuario.add(txtApellidos, gridBagConstraints);

lblUsuario.setText("Usuario");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 0;
gridBagConstraints.gridy = 3;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
pnlUsuario.add(lblUsuario, gridBagConstraints);
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 1;
gridBagConstraints.gridy = 3;
gridBagConstraints.gridwidth = 2;
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
gridBagConstraints.weightx = 1.0;
gridBagConstraints.insets = new java.awt.Insets(14, 14, 14, 14);
pnlUsuario.add(txtUsuario, gridBagConstraints);

lblFoto.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);
lblFoto.setText("<html>Seleccione
 Imagen</html>");
lblFoto.setBorder(javax.swing.BorderFactory.createLineBorder(new java.awt.Color(255, 255, 255)));
lblFoto.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
lblFoto.setPreferredSize(new java.awt.Dimension(100, 100));
lblFoto.addMouseListener(new java.awt.event.MouseAdapter() {
 public void mouseClicked(java.awt.event.MouseEvent evt) {
 lblFotoMouseClicked(evt);
 }
});
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 3;
gridBagConstraints.gridy = 1;
gridBagConstraints.gridheight = 2;
gridBagConstraints.weightx = 0.2;
gridBagConstraints.insets = new java.awt.Insets(6, 6, 6, 6);
pnlUsuario.add(lblFoto, gridBagConstraints);

lblObligatorio.setForeground(new java.awt.Color(255, 0, 0));
lblObligatorio.setText("*");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 3;
gridBagConstraints.gridy = 3;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
pnlUsuario.add(lblObligatorio, gridBagConstraints);

lblObligatorio2.setForeground(new java.awt.Color(255, 0, 0));
lblObligatorio2.setText("*");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 3;
gridBagConstraints.gridy = 5;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
pnlUsuario.add(lblObligatorio2, gridBagConstraints);

lblObligatorio3.setForeground(new java.awt.Color(255, 0, 0));
lblObligatorio3.setText("*");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 3;
gridBagConstraints.gridy = 6;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
pnlUsuario.add(lblObligatorio3, gridBagConstraints);

lblObligatorio4.setForeground(new java.awt.Color(255, 0, 0));
lblObligatorio4.setText("*");
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.gridx = 3;
gridBagConstraints.gridy = 4;
gridBagConstraints.anchor = java.awt.GridBagConstraints.LINE_START;
pnlUsuario.add(lblObligatorio4, gridBagConstraints);

getContentPane().add(pnlUsuario, java.awt.BorderLayout.CENTER);

setSize(new java.awt.Dimension(966, 512));
 setLocationRelativeTo(null);
} // </editor-fold> //GEN-END: initComponents

private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) { //GEN-FIRST:event_btnSalirActionPerformed
 System.exit(0);
} //GEN-LAST:event_btnSalirActionPerformed

private void btnGuardarActionPerformed(java.awt.event.ActionEvent evt) { //GEN-FIRST:event_btnGuardarActionPerformed
 Usuario usuario = new Usuario();
 usuario.setNombre(txtNombre.getText());
 usuario.setApellido(txtApellido.getText());
 usuario.setUsuario(txtUsuario.getText());
 usuario.setEmail(txtEmail.getText());
 usuario.setPassword(new String(pwdPassword.getPassword()));
 usuario.setImagen(imagen);
 usuario = service.nuevoUsuario(usuario);
 JOptionPane.showMessageDialog(this,
 "Usuario Registrado con Exito",
 "Usuario Registrado",
 JOptionPane.PLAIN_MESSAGE);
 System.exit(0);
} //GEN-LAST:event_btnGuardarActionPerformed

```

```

private void lblFotoMouseClicked(java.awt.event.MouseEvent evt) {//GEN-FIRST:event_lblFotoMouseClicked
 //UIManager.put("FileChooser.saveButtonText", "ACEPTAR");
 //UIManager.put("FileChooser.cancelButtonText", "CANCELAR");
 JFileChooser jfc = new JFileChooser();
 Container comp = (Container) ((Container) jfc.getComponents()[3]).getComponents()[3];
 (JButton) comp.getComponent(0).setText("ABRIR");
 (JButton) comp.getComponent(1).setText("CANCELAR");
 FileFilter imageFilter = new FileNameExtensionFilter("Imágenes", ImageIO.getReaderFileSuffixes());
 jfc.setFileFilter(imageFilter);
 jfc.setAcceptAllFileFilterUsed(false);
 if (jfc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
 try {
 File fichero = jfc.getSelectedFile();
 BufferedImage im = ImageIO.read(fichero);
 int max = Math.max(im.getWidth(), im.getHeight());
 double ratio = (double) 100 / max;
 int ancho = (int) (im.getWidth() * ratio);
 int alto = (int) (im.getHeight() * ratio);
 Image imag = im.getScaledInstance(ancho, alto, Image.SCALE_SMOOTH);
 BufferedImage bi = new BufferedImage(
 ancho,
 alto,
 BufferedImage.TYPE_INT_RGB);
 Graphics2D graphics2D = bi.createGraphics();
 graphics2D.drawImage(imag, 0, 0, null);
 graphics2D.dispose();
 lblFoto.setText("");
 lblFoto.setIcon(new ImageIcon(imag));
 ImageIO.write(bi, "jpg", new File("images", fichero.getName()));
 imagen = fichero.getName();
 } catch (IOException ex) {
 Logger.getLogger(FrmRegistro.class.getName()).log(Level.SEVERE, null, ex);
 }
 }
}//GEN-LAST:event_lblFotoMouseClicked

DocumentListener listener = new DocumentListener() {
 @Override
 public void insertUpdate(DocumentEvent e) {
 checkDocument(e);
 }

 @Override
 public void removeUpdate(DocumentEvent e) {
 checkDocument(e);
 }

 @Override
 public void changedUpdate(DocumentEvent e) {
 checkDocument(e);
 }

 void checkDocument(DocumentEvent evt) {
 String pwd = new String(pwdPassword.getPassword());
 String pwd2 = new String(pwdPassword2.getPassword());
 String usuario = txtUsuario.getText();
 String email = txtEmail.getText();
 Object origen = evt.getDocument().getProperty("origen");
 lblError.setText(" ");
 if (origen == pwdPassword) {
 if (pwd.isBlank() || pwd2.isBlank()) {
 btnGuardar.setEnabled(false);
 return;
 }
 if (!pwd.equals(pwd2)) {
 lblError.setText("Las contraseñas no coinciden");
 btnGuardar.setEnabled(false);
 return;
 }
 } else if (origen == txtUsuario) {
 if (usuario.isBlank()) {
 btnGuardar.setEnabled(false);
 return;
 }
 if (service.getUsuario(usuario) != null) {
 lblError.setText("El usuario no está disponible. Seleccione otra opción");
 btnGuardar.setEnabled(false);
 return;
 }
 }
 if (email.isBlank()) {
 btnGuardar.setEnabled(false);
 return;
 } else {
 String regex = "^(\\w+$|['+/-=?`{|}~^=]+(?:\\.(\\w+$|['+/-=?`{|}~^=]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,6}$)"; // Email
 Pattern pattern = Pattern.compile(regex);
 Matcher matcher = pattern.matcher(email);
 if (!matcher.matches())
 lblError.setText("El email no es válido");
 btnGuardar.setEnabled(false);
 return;
 }
 }
}

```

```

 if (!service.isDisponibleEmail(email)) {
 lblError.setText("Ya hay un usuario registrado con ese email");
 btnGuardar.setEnabled(false);
 return;
 }
 }
 btnGuardar.setEnabled(true);
}

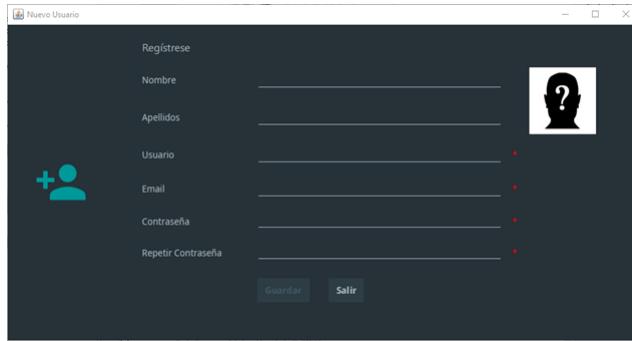
/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
 /* Set the Nimbus look and feel */
 //editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) "
 /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
 * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
 */
 try {
 for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.UIManager.getInstalledLookAndFeels()) {
 if ("Nimbus".equals(info.getName())) {
 javax.swing.UIManager.setLookAndFeel(info.getClassName());
 break;
 }
 }
 } catch (ClassNotFoundException ex) {
 java.util.logging.Logger.getLogger(FrmRegistro.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
 } catch (InstantiationException ex) {
 java.util.logging.Logger.getLogger(FrmRegistro.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
 } catch (IllegalAccessException ex) {
 java.util.logging.Logger.getLogger(FrmRegistro.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
 } catch (javax.swing.UnsupportedLookAndFeelException ex) {
 java.util.logging.Logger.getLogger(FrmRegistro.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);
 }
 //
 //

 /* Create and display the form */
 java.awt.EventQueue.invokeLater(new Runnable() {
 public void run() {
 new FrmRegistro().setVisible(true);
 }
 });
}

// Variables declaration - do not modify//GEN-BEGIN:variables
private javax.swing.JButton btnGuardar;
private javax.swing.JButton btnSalir;
private javax.swing.Box.Filler filler1;
private javax.swing.Box.Filler filler2;
private javax.swing.JLabel lblApellidos;
private javax.swing.JLabel lblEmail;
private javax.swing.JLabel lblError;
private javax.swing.JLabel lblFoto;
private javax.swing.JLabel lblLogo;
private javax.swing.JLabel lblNombre;
private javax.swing.JLabel lblObligatorio;
private javax.swing.JLabel lblObligatorio2;
private javax.swing.JLabel lblObligatorio3;
private javax.swing.JLabel lblObligatorio4;
private javax.swing.JLabel lblPassword;
private javax.swing.JLabel lblPassword2;
private javax.swing.JLabel lblRegistrese;
private javax.swing.JLabel lblUsuario;
private javax.swing.JPanel pnLogo;
private javax.swing.JPanel pnlUsuario;
private javax.swing.JPasswordField pwdPassword;
private javax.swing.JPasswordField pwdPassword2;
private javax.swing.JTextField txtApellidos;
private javax.swing.JTextField txtEmail;
private javax.swing.JTextField txtNombre;
private javax.swing.JTextField txtUsuario;
// End of variables declaration//GEN-END:variables
}

```

La salida:



La aplicación controlará la información asignando un `DocumentListener` a los controles obligatorios para validarlos.

### 8.5. Testando una Aplicación Gráfica

Para poder testear una aplicación Swing necesitamos una librería que nos permita ejecutar un formulario y acceder a sus controles. Vamos a emplear `AssertJ`.

Lo primero, como siempre, es la dependencia:

```
<dependency>
 <groupId>org.assertj</groupId>
 <artifactId>assertj-swing-junit</artifactId>
 <version>3.9.2</version>
 <scope>test</scope>
</dependency>
```

El proceso es el siguiente:

- Tenemos que obtener un `FrameFixture` del formulario que queremos controlar y testar. Para ello empleamos un código similar al siguiente:

```
Formularioframe = GuiActionRunner.execute(() -> new Formulario());
window = new FrameFixture(frame);
window.show();

• Con eso, obtenemos un objeto (window) que dispone de métodos para acceder a todos los controles en base a su propiedad name. Por ejemplo,
window.textBox("txt") nos daría acceso al JTextField con propiedad name txt. Sobre ese control podemos ejecutar tareas como añadir texto (enterText), eliminarlo (deleteText), presionar teclas (pressKey)... y podemos comprobar el texto que tiene con requireText
```

Veamos un ejemplo simple:

```
package org.zabalburu.tmdb.views;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;
import jiconfont.swing.IconFontSwing;
import mdlat.MaterialLookAndFeel;
import mdlat.themes.MaterialOceanicTheme;
import org.assertj.swing.edt.FailOnThreadViolationRepaintManager;
import org.assertj.swing.edt.GuiActionRunner;
import org.assertj.swing.fixture.FrameFixture;
import org.junit.After;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;
import org.zabalburu.tmdb.TMDB;
import org.zabalburu.util.Conexion;

/**
 *
 * @author IñigoChueca
 */
public class FrmLoginTest {

 public FrmLoginTest() {
 }

 private FrameFixture window;

 @BeforeAll
 public static void setUpOnce() {
 try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new MaterialOceanicTheme()));
 IconFontSwing.register(GoogleMaterialDesignIcons.getIconFont());
 } catch (UnsupportedLookAndFeelException ex) {
 Logger.getLogger(TMDB.class.getName()).log(Level.SEVERE, null, ex);
 }
 FailOnThreadViolationRepaintManager.install();
 }

 @BeforeEach
```

```

public void setUp() {
 FrmLogin frame = GuiActionRunner.execute(() -> new FrmLogin());
 window = new FrameFixture(frame);
 window.show(); // shows the frame to test*/
}

@Test
public void shouldShowError() {
 window.textBox("txtUsuario").enterText(""); // Emplea la propiedad name del control
 window.textBox("txtPassword").enterText("");
 window.button("btnEntrar").click();
 window.label("lblError").requireText("Debe especificarse el usuario y la contraseña");

 window.textBox("txtUsuario").enterText("");
 window.textBox("txtPassword").enterText("12345");
 window.button("btnEntrar").click();
 window.label("lblError").requireText("Debe especificarse el usuario y la contraseña");

 window.textBox("txtUsuario").enterText("ichueca"); // Emplea la propiedad name del control
 window.textBox("txtPassword").deleteText();
 window.button("btnEntrar").click();
 window.label("lblError").requireText("Debe especificarse el usuario y la contraseña");
}

@Test
public void shouldShowLoginError() {
 window.textBox("txtUsuario").enterText("ichueca"); // Emplea la propiedad name del control
 window.textBox("txtPassword").enterText("123456");
 window.button("btnEntrar").click();
 window.label("lblError").requireText("Usuario / password erróneos");
}

@AfterEach
public void tearDown() {
 window.cleanUp();
}
}

```

Estos tests comprueban los posibles errores a la hora de introducir información en la ventana de login.

Si añadimos el código para comprobar también un login correcto:

```

package org.zabalburu.tmdb.views;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;
import jiconfont.swing.IconFontSwing;
import mdlaf.MaterialLookAndFeel;
import mdlaf.themes.MaterialOceanicTheme;
import org.assertj.swing.edt.FailOnThreadViolationRepaintManager;
import org.assertj.swing.edt.GuiActionRunner;
import org.assertj.swing.fixture.FrameFixture;
import org.junit.After;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;
import org.zabalburu.tmdb.TMDB;
import org.zabalburu.util.Conexion;

/**
 *
 * @author IñigoChueca
 */
public class FrmLoginTest {

 public FrmLoginTest() {
 }

 private FrameFixture window;

 @BeforeAll
 public static void setUpOnce() {
 try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new MaterialOceanicTheme()));
 IconFontSwing.register(GoogleMaterialDesignIcons.getIconFont());
 } catch (UnsupportedLookAndFeelException ex) {
 Logger.getLogger(TMDB.class.getName()).log(Level.SEVERE, null, ex);
 }
 FailOnThreadViolationRepaintManager.install();
 }

 @BeforeEach
 public void setUp() {
 FrmLogin frame = GuiActionRunner.execute(() -> new FrmLogin());
 window = new FrameFixture(frame);
 }
}

```

```

 window.show(); // shows the frame to test*/
 }

 @Test
 public void shouldShowError() {
 window.textBox("txtUsuario").enterText(""); // Emplea la propiedad name del control
 window.textBox("txtPassword").enterText("");
 window.button("btnEntrar").click();
 window.label("lblError").requireText("Debe especificarse el usuario y la contraseña");

 window.textBox("txtUsuario").enterText("");
 window.textBox("txtPassword").enterText("12345");
 window.button("btnEntrar").click();
 window.label("lblError").requireText("Debe especificarse el usuario y la contraseña");

 window.textBox("txtUsuario").enterText("ichueca"); // Emplea la propiedad name del control
 window.textBox("txtPassword").deleteText();
 window.button("btnEntrar").click();
 window.label("lblError").requireText("Debe especificarse el usuario y la contraseña");
 }

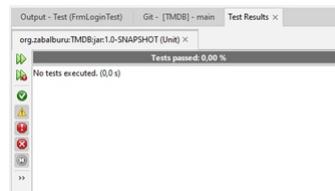
 @Test
 public void shouldShowLoginError() {
 window.textBox("txtUsuario").enterText("ichueca"); // Emplea la propiedad name del control
 window.textBox("txtPassword").enterText("123456");
 window.button("btnEntrar").click();
 window.label("lblError").requireText("Usuario / password erróneos");
 }

 @Test
 public void shouldShowLoginOk() {
 window.textBox("txtUsuario").enterText("ichueca"); // Emplea la propiedad name del control
 window.textBox("txtPassword").enterText("12345");
 window.button("btnEntrar").click();
 window.label("lblError").requireText("");
 window.optionPane().requireTitle("Usuario Conectado Correctamente!");
 }

 @AfterEach
 public void tearDown() {
 window.cleanUp();
 }
}

```

Si lo probamos:



Como vemos los tests no se ejecutan (es posible que se ejecute alguno pero no todos). El problema es que, cuando el login tiene éxito, finalizamos la aplicación con `System.exit(0)`, lo que finaliza también los tests a lo bestia.

Para solucionarlo podemos emplear la librería `system-lambda` que nos permite moquear llamadas a `System` (Mockito no lo permite por motivos de seguridad).

El código quedaría así:

```

 @Test
 public void shouldShowLoginOk() throws Exception{
 try{
 int statusCode = catchSystemExit(() -> {
 window.textBox("txtUsuario").enterText("ichueca"); // Emplea la propiedad name del control
 window.textBox("txtPassword").enterText("12345");
 window.button("btnEntrar").click();
 window.label("lblError").requireText("");
 window.optionPane().requireTitle("Usuario Conectado Correctamente!");
 });
 Assertions.assertEquals(0, statusCode);
 } catch (java.lang.UnsupportedOperationException ex){}
 }

```

Y todos los tests pasarán correctamente.

NOTA: El uso de esta última opción emplea una operación que está `deprecated` por lo que es posible que no funcione en futuras versiones de Java. Una solución podría ser lanzar una excepción personalizada para finalizar lo que nos permitiría capturarla en los test

## 8.6. Formulario Principal

Vamos a hacer un formulario que muestre las películas en cartelera, las más populares y las mejor valoradas.

En primer lugar, creamos el DAO:

```

package org.zabalburu.tmdb.dao;

import info.movisto.themoviedbapi.model.core.Movie;
import java.util.List;

```

```

/**
 *
 * @author Iñigo
 */
public interface TMDBDAO {
 List<Movie> getNowPlaying(Integer pagina);
 List<Movie> getPopular(Integer pagina);
 List<Movie> getTopRated(Integer pagina);
}

La implementación:

package org.zabalburu.tmdb.dao;

import info.movito.themoviedbapi.TmdbApi;
import info.movito.themoviedbapi.model.core.Movie;
import info.movito.themoviedbapi.model.core.MovieResultsPage;
import info.movito.themoviedbapi.model.movielists.MovieResultsPageWithDates;
import info.movito.themoviedbapi.tools.TmdbException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author Iñigo
 */
public final class TMDBImpl implements TMDBDAO{
 private static String API_KEY="eyJhbGciOiJIUzI1NiJ9eyJhdWQiOiJhNDI1ZWQ1MTMxNzMyMjA4MWQwOWE4MGIS5YmE5NzFhOSIsInNlYiI6IjVjNWFhNTYzYzNhMzY4M2NkNTg3NTYwZSI;

 private TmdbApi api = new TmdbApi(API_KEY);

 public static TMDBDAO get() {
 return dao;
 }

 private static TMDBDAO dao = new TMDBImpl();

 private TMDBImpl(){}

 @Override
 public List<Movie> getNowPlaying(Integer pagina) {
 try {
 MovieResultsPageWithDates result = api.getMovieLists().getNowPlaying("es_ES", pagina, "ES");
 return result.getResults();
 } catch (TmdbException ex) {
 Logger.getLogger(TMDBImpl.class.getName()).log(Level.SEVERE, null, ex);
 }
 return new ArrayList<>();
 }

 @Override
 public List<Movie> getPopular(Integer pagina) {
 try {
 MovieResultsPage result = api.getMovieLists().getPopular("es_ES", pagina, "ES");
 return result.getResults();
 } catch (TmdbException ex) {
 Logger.getLogger(TMDBImpl.class.getName()).log(Level.SEVERE, null, ex);
 }
 return new ArrayList<>();
 }

 @Override
 public List<Movie> getTopRated(Integer pagina) {
 try {
 MovieResultsPage result = api.getMovieLists().getTopRated("es_ES", pagina, "ES");
 return result.getResults();
 } catch (TmdbException ex) {
 Logger.getLogger(TMDBImpl.class.getName()).log(Level.SEVERE, null, ex);
 }
 return new ArrayList<>();
 }

 public static void main(String[] args) {
 System.out.println(TMDBImpl.get().getNowPlaying(0));
 }
}

```

#### El servicio:

```

package org.zabalburu.tmdb.service;

import info.movito.themoviedbapi.model.core.Movie;
import java.util.List;
import org.zabalburu.tmdb.dao.TMDBDAO;
import org.zabalburu.tmdb.dao.TMDBImpl;
import org.zabalburu.tmdb.dao.UsuarioDAO;
import org.zabalburu.tmdb.dao.UsuarioImpl;
import org.zabalburu.tmdb.modelo.Usuario;
import org.zabalburu.util.PasswordManager;

```

```

/**
 *
 * @author IñigoChueca
 */
public class TMDBService {
 private static UsuarioDAO usuarioDAO = new UsuarioImpl();

 public static UsuarioDAO getUsuarioDAO() {
 return usuarioDAO;
 }

 public static void setUsuarioDAO(UsuarioDAO usuarioDAO) {
 TMDBService.usuarioDAO = usuarioDAO;
 }
 private static final TMDBService service = new TMDBService();
 private static TMDBDAO tmdbDao = TMDBImpl.get();

 private TMDBService(){
 }

 public static final TMDBService getService(){
 return service;
 }

 public Usuario getUsuario(Integer id){
 return usuarioDAO.getUsuario(id);
 }

 public Usuario getUsuario(String usuario){
 return usuarioDAO.getUsuario(usuario);
 }

 public boolean isDisponibleEmail(String email){
 //Usuario u = usuarioDAO.getUsuarios().get(0);
 //System.out.println(u.getEmail().equalsIgnoreCase(email));
 List<Usuario> usuarios = usuarioDAO.getUsuarios();
 return usuarioDAO.getUsuarios()
 .stream()
 .filter(u -> u.getEmail().equalsIgnoreCase(email))
 .map(u -> false)
 .findAny()
 .orElse(true);
 }

 public Usuario nuevoUsuario(Usuario usuario){
 return usuarioDAO.nuevoUsuario(usuario);
 }

 public Usuario login(String usuario, String contraseña){
 Usuario u = usuarioDAO.getUsuario(usuario);
 if (u != null){
 if (!PasswordManager.verificarContraseña(contraseña, u.getPassword())){
 u = null;
 }
 }
 return u;
 }

 public List<Usuario> getUsuarios(){
 return usuarioDAO.getUsuarios();
 }

 public List<Movie> getNowPlaying(Integer pagina){
 return tmdbDao.getNowPlaying(pagina);
 }

 public List<Movie> getPopular(Integer pagina){
 return tmdbDao.getPopular(pagina);
 }

 public List<Movie> getTopRated(Integer pagina){
 return tmdbDao.getTopRated(pagina);
 }
}

```

## Panel de Usuario

Vamos a hacer un Panel de Usuario. Pero primero hacemos el componente para una imagen circular:

NOTA: Los componentes que hemos creado así, se pueden arrastrar directamente a otros formularios desde la ventana de diseño de NetBeans.

```

package org.zabalburu.tmdb.controls;

import java.awt.AlphaComposite;
import java.awt.Color;
import java.awt.Composite;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;

```

```

import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JComponent;
import javax.swing.JFrame;

/**
 *
 * @author IñigoChueca
 */
public class ImageAvatar extends JComponent{
 private Icon icon;
 private int borderSize = 2;

 public Icon getIcon() {
 return icon;
 }

 public void setIcon(Icon icon) {
 this.icon = icon;
 }

 public int getBorderSize() {
 return borderSize;
 }

 public void setBorderSize(int borderSize) {
 this.borderSize = borderSize;
 }
 @Override
 protected void paintComponent(Graphics g) {

 if (icon != null){
 int ancho = getWidth();
 int alto = getHeight();
 int min = Math.min(ancho, alto);
 int x = (ancho - min) / 2;
 int y = (alto - min) / 2;
 int border = borderSize * 2;
 min -= border;
 Dimension size = getAutoSize(icon, min);
 BufferedImage img = new BufferedImage(size.width, size.height, BufferedImage.TYPE_INT_ARGB);
 Graphics2D g2_img = img.createGraphics();
 g2_img.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
 g2_img.fillOval(0, 0, min, min);
 Composite composite = g2_img.getComposite();
 g2_img.setComposite(AlphaComposite.SrcIn);
 g2_img.setRenderingHint(RenderingHints.KEY_INTERPOLATION, RenderingHints.VALUE_INTERPOLATION_BILINEAR);
 g2_img.drawImage(toImage(icon), 0, 0, size.width, size.height, null);
 g2_img.setComposite(composite);
 g2_img.dispose();
 Graphics2D g2 = (Graphics2D) g;
 g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
 if (borderSize > 0){
 min += border;
 g2.setColor(getForeground());
 g2.fillOval(x, y, min, min);
 }
 if (isOpaque()){
 g2.setColor(getBackground());
 min -= border;
 g2.fillOval(x+borderSize, y+borderSize, WIDTH, HEIGHT);
 }
 g.drawImage(img, x+borderSize, y+borderSize, null);
 }
 super.paintComponent(g); // Generated from nbfs://nbhost/SystemFileSystem/Templates/Classes/Code/OverriddenMethodBody
 }

 private Image toImage(Icon icon){
 return ((ImageIcon) icon).getImage();
 }

 private Dimension getAutoSize(Icon image , int size){
 int w = size;
 int h = size;
 int iw = image.getIconWidth();
 int ih = image.getIconHeight();
 double xScale = (double) w / iw;
 double yScale = (double) h / ih;
 double scale = Math.max(xScale, yScale);
 int width = (int) (scale * iw);
 int height = (int) (scale * ih);
 if (width < 1){
 width = 1;
 }
 if (height < 1){
 height = 1;
 }
 return new Dimension(width, height);
 }

 public static void main(String[] args) {
 JFrame f = new JFrame();
 //f.setLayout(null);
 }
}

```

```

 ImageAvatar img = new ImageAvatar();
 img.setSize(new Dimension(100,100));
 img.setIcon(new ImageIcon("images/juanLopez.jpg"));
 img.setForeground(Color.BLUE);
 img.setBorderSize(2);
 img.setBackground(Color.red);
 f.add(img);
 f.setSize(new Dimension(150,150));
 f.setLocationRelativeTo(null);
 f.setVisible(true);
 f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 }
}

```



El panel:

```

package org.zabalburu.tmdb.panels;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;
import jiconfont.swing.IconFontSwing;
import mdmaf.MaterialLookAndFeel;
import mdmaf.themes.MaterialOceanicTheme;
import org.zabalburu.tmdb.TMDB;
import org.zabalburu.tmdb.modelo.Usuario;
import org.zabalburu.tmdb.service.TMDBService;

/**
 *
 * @author IñigoChueca
 */
public class PnlUsuario extends javax.swing.JPanel {
 private static Usuario usuario = null;
 public void setUsuario(Usuario usuario){
 PnlUsuario.usuario = usuario;
 avUsuario.setIcon(new ImageIcon("images/"+usuario.getImagen()));
 }

 public static Usuario getUsuario() {
 return usuario;
 }

 /**
 * Creates new form PnlUsuario
 */
 public PnlUsuario() {
 initComponents();
 avUsuario.setIcon(new ImageIcon("images/no_image.jpg"));
 txtBuscar.setBackground(Color.WHITE);
 }

 /**
 * This method is called from within the constructor to initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is always
 * regenerated by the Form Editor.
 */
 @SuppressWarnings("unchecked")
 // <editor-fold defaultstate="collapsed" desc="Generated Code">
 private void initComponents() {
 java.awt.GridBagConstraints gridBagConstraints;

 lblTitulo = new javax.swing.JLabel();
 txtBuscar = new javax.swing.JTextField();
 avUsuario = new org.zabalburu.tmdb.controls.ImageAvatar();

 setBorder(javax.swing.BorderFactory.createEmptyBorder(10, 10, 10, 10));
 setLayout(new java.awt.GridBagLayout());

 lblTitulo.setFont(new java.awt.Font("Segoe UI", 0, 24)); // NOI18N
 lblTitulo.setText("The Movie Database ");
 gridBagConstraints = new java.awt.GridBagConstraints();
 gridBagConstraints.insets = new java.awt.Insets(5, 5, 5, 5);
 add(lblTitulo, gridBagConstraints);

 txtBuscar.setFont(new java.awt.Font("Segoe UI", 0, 16)); // NOI18N
 txtBuscar.setForeground(new java.awt.Color(204, 204, 204));
 txtBuscar.setText("Buscar películas...");
 txtBuscar.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 5, 1, 1));
 txtBuscar.setMinimumSize(new java.awt.Dimension(71, 40));
 }
}

```

```

txtBuscar.setPreferredSize(new java.awt.Dimension(71, 40));
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.fill = java.awt.GridBagConstraints.HORIZONTAL;
gridBagConstraints.weightx = 1.0;
add(txtBuscar, gridBagConstraints);

avUsuario.setMaximumSize(new java.awt.Dimension(60, 60));
avUsuario.setMinimumSize(new java.awt.Dimension(60, 60));
avUsuario.setPreferredSize(new java.awt.Dimension(60, 60));
avUsuario.setRequestFocusEnabled(false);
gridBagConstraints = new java.awt.GridBagConstraints();
gridBagConstraints.insets = new java.awt.Insets(5, 5, 5, 5);
add(avUsuario, gridBagConstraints);
} // </editor-fold>

public static void main(String[] args) {
 TMDBService servicio = TMDBService.getService();
 try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new MaterialOceanicTheme()));
 IconFontSwing.register(GoogleMaterialDesignIcons.getIconFont());
 } catch (UnsupportedLookAndFeelException ex) {
 Logger.getLogger(TMDB.class.getName()).log(Level.SEVERE, null, ex);
 }
 JFrame frm = new JFrame();
 PnlUsuario pnl = new PnlUsuario();
 frm.add(pnl, BorderLayout.NORTH);
 pnl.setUsuario(servicio.getUsuario(2));
 System.out.println(servicio.getUsuario(1).getImagen());
 frm.setSize(new Dimension(1000,120));
 frm.setLocationRelativeTo(null);
 frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frm.setVisible(true);
}

// Variables declaration - do not modify
private org.zabalburu.tmdb.controls.ImageAvatar avUsuario;
private javax.swing.JLabel lblTitulo;
private javax.swing.JTextField txtBuscar;
// End of variables declaration
}

```



## Panel de Película

Antes de hacer el panel vamos a crear una **etiqueta personalizada para mostrar la media de votos de la película como un porcentaje**:

```

package org.zabalburu.tmdb.controls;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.font.TextAttribute;
import java.text.AttributedString;
import javax.swing.JFrame;
import javax.swing.JLabel;

```

```

/**
 *
 * @author IñigoChueca
 */
public class VoteJLabel extends JLabel {
 private enum Tamaño {
 PEQUEÑO, GRANDE
 };
 private int porcentaje = 0;
 private Tamaño tamaño;

 public VoteJLabel() {
 this.tamaño = Tamaño.PEQUEÑO;
 //this.setText(texto);
 }

 public int getPorcentaje() {
 return porcentaje;
 }

 public void setTamaño(Tamaño tamaño) {
 this.tamaño = tamaño;
 }

 public void setPorcentaje(int porcentaje) {
 this.porcentaje = porcentaje*10;
 repaint();
 }
}

```

```

@Override
public void paint(Graphics g) {
 super.paint(g); // Generated from nbfs://nbhost/SystemFileSystem/Templates/Classes/Code/OverriddenMethodBody
 Graphics2D g2d;
 if (g instanceof Graphics2D) {
 g2d = (Graphics2D) g;
 String texto = ""+porcentaje + "%";
 int anchoTexto = (int) g.getFontMetrics().getStringBounds(texto, g).getWidth();
 int ancho = this.getWidth() - anchoTexto ;
 int alto = this.getHeight() + (int) g.getFontMetrics().getStringBounds(texto, g).getHeight() - 5;

 g2d.setColor(Color.BLACK);
 int stroke = this.tamaño == Tamaño.PEQUEÑO ? 2 : 4;
 g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
 g2d.fillOval(stroke,stroke,this.getWidth()-stroke*2, this.getHeight()-stroke*2);
 g2d.setColor(Color.WHITE);
 AttributedString text = new AttributedString(texto);
 //g2d.drawString(texto , ancho / 2 - 3, alto / 2);
 if (this.tamaño == Tamaño.PEQUEÑO) {
 text.addAttribute(TextAttribute.FONT, new Font("Arial",Font.BOLD,12),0, texto.length() - 1);
 text.addAttribute(TextAttribute.SUPERSCRIPT, TextAttribute.SUPERSCRIPT_SUPER, texto.length() - 1, texto.length()); //this.setFont
 g2d.drawString(text.getIterator() , ancho / 2 + 5 , alto / 2 - 2);
 } else {
 text.addAttribute(TextAttribute.FONT, new Font("Arial",Font.BOLD,18),0, texto.length() - 1);
 text.addAttribute(TextAttribute.SUPERSCRIPT, TextAttribute.SUPERSCRIPT_SUPER, texto.length() - 1, texto.length()); //this.setFont
 g2d.drawString(text.getIterator() , ancho / 2 , alto / 2);
 }

 if (porcentaje < 40)
 g2d.setColor(Color.red);
 else if (porcentaje < 75)
 g2d.setColor(Color.yellow);
 else
 g2d.setColor(Color.green);

 g2d.setStroke(new BasicStroke(stroke));
 g.drawArc(stroke, stroke, this.getWidth()-stroke*2, this.getHeight()-stroke*2, 90, (int) (-porcentaje * 3.6)); // (int) (porcentaje * 3.6));
 }
}

public static void main(String[] args) {
 JFrame frm = new JFrame();
 frm.setLayout(null);
 frm.setSize(100, 100);
 //frm.setUndecorated(true);
 VoteJLabel lbl = new VoteJLabel();
 lbl.setOpaque(true);
 lbl.setTamaño(Tamaño.PEQUEÑO);
 lbl.setSize(new Dimension(30,30));
 lbl.setLocation(0,0);
 frm.add(lbl);
 VoteJLabel lbl2 = new VoteJLabel();
 lbl2.setOpaque(true);
 lbl2.setTamaño(Tamaño.GRANDE);
 lbl2.setSize(new Dimension(60,60));
 lbl2.setLocation(40, 0);
 frm.add(lbl2);

 frm.setLocationRelativeTo(null);
 frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frm.setVisible(true);
 lbl.setPorcentaje(7);
 lbl2.setPorcentaje(9);
}
}

```



El panel:

```

package org.zabalburu.tmdb.panels;

import info.movisto.themoviedbapi.model.core.Movie;
import java.net.MalformedURLException;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.URL;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;
import jiconfont.swing.IconFontSwing;
import mdlaf.MaterialLookAndFeel;

```

```

import mdlaf.themes.MaterialOceanicTheme;
import org.zabalburu.tmdb.TMDB;
import org.zabalburu.tmdb.service.TMDBService;

/**
 *
 * @author IñigoChueca
 */
public class PnlMovie extends javax.swing.JPanel {

 private Movie movie;

 public PnlMovie() {
 }

 /**
 * Creates new form pnlMovie
 */
 public PnlMovie(Movie movie) {
 this.movie = movie;
 initComponents();
 lblTitulo.setText("<html><center>" + this.movie.getTitle() + "</center></html>");
 try {
 URL url = new URI("https://image.tmdb.org/t/p/w92/" +
 + this.movie.getPosterPath()).toURL();
 lblPoster.setIcon(new ImageIcon(url));
 lblVotos.setPorcentaje(this.movie.getVoteAverage().intValue());
 } catch (URISyntaxException ex) {
 Logger.getLogger(PnlMovie.class.getName()).log(Level.SEVERE, null, ex);
 } catch (MalformedURLException ex) {
 Logger.getLogger(PnlMovie.class.getName()).log(Level.SEVERE, null, ex);
 }
 }

 /**
 * This method is called from within the constructor to initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is always
 * regenerated by the Form Editor.
 */
 @SuppressWarnings("unchecked")
 // <editor-fold defaultstate="collapsed" desc="Generated Code">
 private void initComponents() {

 lblTitulo = new javax.swing.JLabel();
 lblVotos = new org.zabalburu.tmdb.controls.VoteJLabel();
 lblPoster = new javax.swing.JLabel();

 setMaximumSize(new java.awt.Dimension(120, 230));
 setMinimumSize(new java.awt.Dimension(120, 230));
 setPreferredSize(new java.awt.Dimension(120, 230));
 setLayout(null);

 lblTitulo.setFont(new java.awt.Font("Segoe UI", 0, 10)); // NOI18N
 lblTitulo.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
 lblTitulo.setText("jLabel1");
 lblTitulo.setVerticalAlignment(javax.swing.SwingConstants.TOP);
 lblTitulo.setVerticalTextPosition(javax.swing.SwingConstants.TOP);
 add(lblTitulo);
 lblTitulo.setBounds(0, 140, 92, 70);

 lblVotos.setPorcentaje(90);
 add(lblVotos);
 lblVotos.setBounds(60, 115, 30, 30);

 lblPoster.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
 lblPoster.setVerticalAlignment(javax.swing.SwingConstants.TOP);
 add(lblPoster);
 lblPoster.setBounds(0, 0, 92, 138);
 } // </editor-fold>
 public static void main(String[] args) {
 try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new MaterialOceanicTheme()));
 IconFontSwing.register(GoogleMaterialDesignIcons.getIconFont());
 } catch (UnsupportedLookAndFeelException ex) {
 Logger.getLogger(TMDB.class.getName()).log(Level.SEVERE, null, ex);
 }
 JFrame frm = new JFrame();
 TMDBService servicio = TMDBService.getService();
 PnlMovie pnl = new PnlMovie(servicio.getNowPlaying(0).get(0));
 frm.add(pnl);
 frm.setSize(pnl.getPreferredSize());
 frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frm.setLocationRelativeTo(null);
 frm.setVisible(true);
 }

 // Variables declaration - do not modify
 private javax.swing.JLabel lblPoster;
 private javax.swing.JLabel lblTitulo;
 private org.zabalburu.tmdb.controls.VoteJLabel lblVotos;
 // End of variables declaration
}

```



## Panel de Películas

Muestra la información de varias películas:

```
package org.zabalburu.tmdb.panels;

import info.movito.themoviedbapi.model.core.Movie;
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;
import jiconfont.swing.IconFontSwing;
import mtlaf.MaterialLookAndFeel;
import mtlaf.themes.MaterialOceanicTheme;
import org.zabalburu.tmdb.TMDB;
import org.zabalburu.tmdb.service.TMDBService;

/**
 *
 * @author IñigoChueca
 */
public class PnlMoviesList extends javax.swing.JPanel {

 private List<Movie> movies;

 public PnlMoviesList() {
 this(new ArrayList<>());
 }

 public void setMovies(List<Movie> movies) {
 this.movies = movies;
 JPanel pnlPelis = new JPanel();
 pnlPelis.setLayout(new BoxLayout(pnlPelis, BoxLayout.X_AXIS));
 pnlPelis.add(Box.createHorizontalGlue());
 for (int i = 0; i < movies.size(); i++) {
 pnlPelis.add(new PnlMovie(movies.get(i)));
 this.add(Box.createHorizontalStrut(10));
 }
 pnlPelis.add(Box.createHorizontalGlue());
 this.setLayout(new BorderLayout());
 JScrollPane jsp = new JScrollPane(pnlPelis);
 this.setPreferredSize(new Dimension(1000, 300));
 this.add(jsp);
 }

 /**
 * Creates new form pnlMoviesList
 */
 public PnlMoviesList(List<Movie> movies) {
 initComponents();
 //int min = Math.min(shown, movies.size());
 }

 public static void main(String[] args) {
 try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new MaterialOceanicTheme()));
 IconFontSwing.register(GoogleMaterialDesignIcons.getIconFont());
 } catch (UnsupportedLookAndFeelException ex) {
 Logger.getLogger(TMDB.class.getName()).log(Level.SEVERE, null, ex);
 }
 JFrame frm = new JFrame();
 TMDBService servicio = TMDBService.getService();
 PnlMoviesList pnl = new PnlMoviesList();
 pnl.setMovies(servicio.getNowPlaying(0));
 frm.add(pnl);
 frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frm.setSize(pnl.getPreferredSize());
 frm.setLocationRelativeTo(null);
 frm.setVisible(true);
 }
}
```

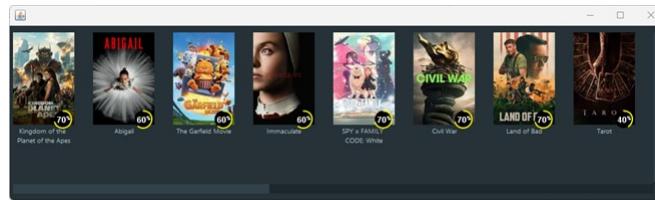
```

 * This method is called from within the constructor to initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is always
 * regenerated by the Form Editor.
 */
@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {

 setBorder(javax.swing.BorderFactory.createEmptyBorder(10, 5, 10, 5));
 setMaximumSize(new java.awt.Dimension(160, 230));
 setMinimumSize(new java.awt.Dimension(160, 230));
 setPreferredSize(new java.awt.Dimension(160, 230));
 setLayout(new javax.swing.BoxLayout(this, javax.swingBoxLayout.LINE_AXIS));
} // </editor-fold>

// Variables declaration - do not modify
// End of variables declaration
}

```



## Formulario Principal

El formulario principal muestra un panel de películas para cada búsqueda y el panel de usuario:

```

package org.zabalburu.tmdb.views;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import jiconfont.icons.google_material_design_icons.GoogleMaterialDesignIcons;
import jiconfont.swing.IconFontSwing;
import mdlat.MaterialLookAndFeel;
import mdlat.themes.MaterialOceanicTheme;
import org.zabalburu.tmdb.TMDB;
import org.zabalburu.tmdb.modelo.Usuario;
import org.zabalburu.tmdb.service.TMDBService;

/**
 *
 * @author IñigoChueca
 */
public class FrmPrincipal extends javax.swing.JFrame {
 private TMDBService servicio = TMDBService.getService();

 private Usuario usuario;

 public Usuario getUsuario() {
 return usuario;
 }

 public void setUsuario(Usuario usuario) {
 this.usuario = usuario;
 this.pnlUsuariol.setUsuario(usuario);
 }

 /**
 * Creates new form FrmPrincipal
 */
 public FrmPrincipal() {
 initComponents();
 pnlMasPopulares.setMovies(servicio.getPopular(0));
 pnlMejorValorados.setMovies(servicio.getTopRated(0));
 pnlActualidad.setMovies(servicio.getNowPlaying(0));
 }

 /**
 * This method is called from within the constructor to initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is always
 * regenerated by the Form Editor.
 */
 @SuppressWarnings("unchecked")
 // <editor-fold defaultstate="collapsed" desc="Generated Code">
 private void initComponents() {
 java.awt.GridBagConstraints gridBagConstraints;

 pnlUsuariol = new org.zabalburu.tmdb.panels.PnlUsuario();
 jPanel2 = new javax.swing.JPanel();
 lblActualidad = new javax.swing.JLabel();
 pnlActualidad = new org.zabalburu.tmdb.panels.PnlMoviesList();
 lblMasPopulares = new javax.swing.JLabel();
 pnlMasPopulares = new org.zabalburu.tmdb.panels.PnlMoviesList();

```



```

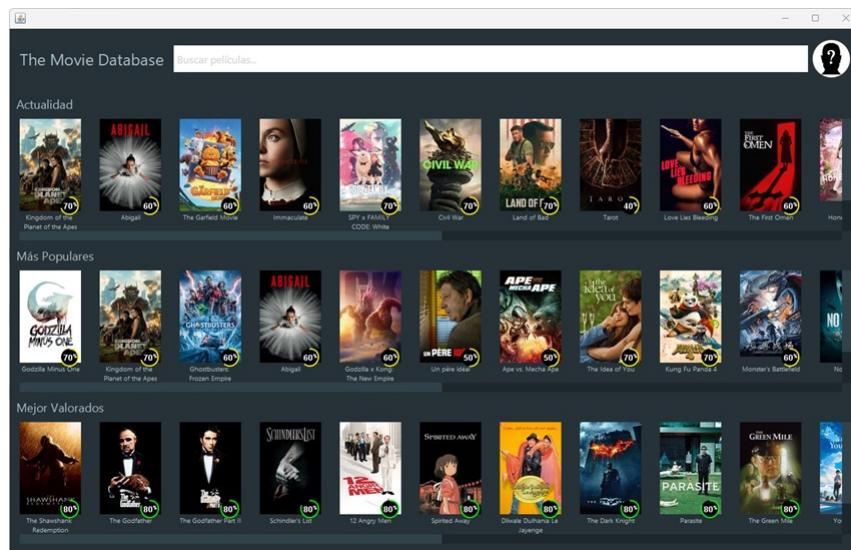
 setSize(new java.awt.Dimension(1293, 829));
 setLocationRelativeTo(null);
 }// </editor-fold>

 /**
 * @param args the command line arguments
 */
 public static void main(String args[]) {
 try {
 UIManager.setLookAndFeel(new MaterialLookAndFeel(new MaterialOceanicTheme()));
 IconFontSwing.register(GoogleMaterialDesignIcons.getIconFont());
 } catch (UnsupportedLookAndFeelException ex) {
 Logger.getLogger(TMDB.class.getName()).log(Level.SEVERE, null, ex);
 }

 /* Create and display the form */
 java.awt.EventQueue.invokeLater(new Runnable() {
 public void run() {
 new FrmPrincipal().setVisible(true);
 }
 });
 }

 // Variables declaration - do not modify
 private javax.swing.JPanel jPanel2;
 private javax.swing.JLabel lblActualidad;
 private javax.swing.JLabel lblMasPopulares;
 private javax.swing.JLabel lblMejorValorados;
 private org.zabalburu.tmdb.panels.PnlMoviesList pnlActualidad;
 private org.zabalburu.tmdb.panels.PnlMoviesList pnlMasPopulares;
 private org.zabalburu.tmdb.panels.PnlMoviesList pnlMejorValorados;
 private org.zabalburu.tmdb.panels.PnlUsuario pnlUsuario;
 // End of variables declaration
}

```



Y ya lo único que queda es llamar a este formulario en el formulario de login:

```

private void btnEntrarActionPerformed(java.awt.event.ActionEvent evt) {
 String user = txtUsuario.getText();
 String pwd = new String(pwdPassword.getPassword());
 if (user.isBlank() || pwd.isBlank()){
 lblError.setText("Debe especificarse el usuario y la contraseña");
 return;
 }
 Usuario usuario = service.login(user, pwd);
 if (usuario == null){
 lblError.setText("Usuario / password erróneos");
 } else {
 JOptionPane.showMessageDialog(this, "Usuario Conectado Correctamente!");
 FrmPrincipal form = new FrmPrincipal();
 form.setUsuario(usuario);
 form.setVisible(true);
 this.dispose();
 }
}

```

Y en el de registro:

```

private void btnGuardarActionPerformed(java.awt.event.ActionEvent evt) {
 Usuario usuario = new Usuario();
 usuario.setNombre(txtNombre.getText());
 usuario.setApellido(txtApellidos.getText());
 usuario.setUsuario(txtUsuario.getText());
 usuario.setEmail(txtEmail.getText());

```

```

 usuario.setPassword(new String(pwdPassword.getPassword()));
 usuario.setImagen(imagen);
 usuario = service.nuevoUsuario(usuario);
 JOptionPane.showMessageDialog(this,
 "Usuario Registrado con Éxito",
 "Usuario Registrado",
 JOptionPane.PLAIN_MESSAGE);
 FrmPrincipal form = new FrmPrincipal();
 form.setUsuario(usuario);
 form.setVisible(true);
 this.dispose();
 }
}

```

## 8.7. ActividadTMDB

### Instrucciones

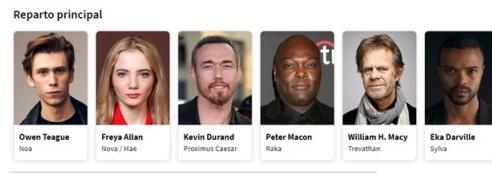
#### Classroom

Se pide diseñar un formulario inspirado en la vista de detalles de una película de TMDB.



### Reto

Añadir el apartado de actores de manera similar a lo hecho con el panel de películas:



NOTA: Para obtener más información podéis emplear el método `api.getMovies().getDetails(id, "es-ES")` que retorna un objeto de tipo `MovieDb` con más información sobre la película.

## 9. Crear una Aplicación Ejecutable

### 9.1. Creación de una Aplicación Ejecutable

En Java, habitualmente las clases se empaquetan en archivos con extensión .jar (que son un tipo de archivos zip) mediante la herramienta jar de java. Si queremos que el archivo jar sea ejecutable, debemos modificar el archivo /META-INF/MANIFEST.MF para indicar cuál es la clase principal. Una vez hecho esto podemos ejecutar la aplicación simplemente con el comando: java -jar archivo.jar O haciendo doble clic. Este proceso es complejo (sobre todo si necesitamos incluir librerías de terceros) y puede simplificarse con el uso de Maven que genera dicho jar por nosotros. Pero tenemos que tener en cuenta varias cosas:

1. Si nuestra aplicación tiene dependencias, hay que incluirlas en el fichero pom.xml
2. Además, si tenemos recursos que queremos incluir en el jar (imágenes, documentos...) también hay que indicarlo en dicho fichero
3. A la hora de generar el archivo .jar es necesario indicarle a maven que queremos añadir dichas dependencias al mismo e indicarle cuál es la clase principal de la aplicación (la que se va a ejecutar)

El fichero pom.xml creado inicialmente tiene un aspecto similar al siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<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
<modelVersion>4.0.0</modelVersion>
<groupId>paquete_base</groupId>
<artifactId>Aplicacion</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>1.8</maven.compiler.source>
 <maven.compiler.target>1.8</maven.compiler.target>
</properties>
</project>
```

#### Añadir Dependencias

Como ya sabemos, las dependencias son librerías de terceros que podemos emplear en nuestra aplicación y que Maven se encargará de descargar de Internet junto con las dependencias de dichas librerías. Las librerías disponibles en Maven se gestionan mediante repositorios, el más importante de los cuales es Maven Central en el que podemos encontrar librerías para casi cualquier cosa. **EJEMPLO:** Supongamos que queremos trabajar con documentos de Microsoft (Word, Excel,...). Podemos emplear la librería Apache POI que buscamos en el repositorio:

The screenshot shows the Maven Repository search results for 'apache poi'. The search bar at the top contains 'apache poi'. The results page has a sidebar with 'Repository' (Central, Spring Plugins, Spring Lib M, Apache Releases, Sonatype, Ibiblio, Spring Lib Release, JBossEA), 'Group' (org.apache, com.github, org.wso2, org.eclipse, maven2.org, maven-, org.mule, org.clojars), and 'Category' (Web App). The main area displays 'Found 26280 results' and lists four entries for Apache POI:

Rank	Dependency	Usages	Group
1.	Apache POI org.apache.poi » poi	1,438 usages	Apache
2.	Apache POI org.apache.poi » poi-ooxml	1,369 usages	Apache
3.	Apache POI org.apache.poi » poi-ooxml-schemas	309 usages	Apache
4.	Apache POI org.apache.poi » poi-scratchpad	290 usages	Apache

Each entry includes a small thumbnail icon of the Apache logo and a link to the dependency's page, along with the last release date (Feb 14, 2020).

Una vez localizada, hacemos clic en la misma:

[mvnrepository.com/artifact/org.apache.poi/poi](https://mvnrepository.com/artifact/org.apache.poi/poi)

**MVN REPOSITORY**

Indexed Artifacts (16.8M)

Apache POI

Apache POI - Java API To Access Microsoft Format Files

License	Apache 2.0
Categories	Excel Libraries
Tags	apache excel spreadsheet
Used By	1,438 artifacts

Central (54) Spring Plugins (2) Spring Lib M (2) BeDataDriven (6) ImageJ Public (1) ICM (5) Alfresco 3rdParty (3) Pellucid (7)  
Geomajas (1) Alfresco (7) Talend (2)

Version	Repository	Usages	Date
4.1.2	Central	58	Feb, 2020
4.1.1	Central	90	Oct, 2019
4.1.0	Central	109	Apr, 2019
4.0.1	Central	116	Dec, 2018
4.0.0	Central	68	Aug, 2018
3.17	Central	304	Sep, 2017
3.17-beta1	Central	12	Jun, 2017
3.16	Central	132	Apr, 2017
3.16-beta2	Central	10	Jan, 2017
3.16-beta1	Central	8	Nov, 2016
3.15	Central	137	Sep, 2016
3.15-beta2	Central	9	Jun, 2016

Y elegimos la versión:

[mvnrepository.com/artifact/org.apache.poi/poi/4.1.2](https://mvnrepository.com/artifact/org.apache.poi/poi/4.1.2)

**MVN REPOSITORY**

Indexed Artifacts (16.8M)

Apache POI » 4.1.2

Apache POI - Java API To Access Microsoft Format Files

License	Apache 2.0
Categories	Excel Libraries
Organization	Apache Software Foundation
HomePage	<a href="http://poi.apache.org/">http://poi.apache.org/</a>
Date	(Feb 14, 2020)
Files	Jar (2.8 MB) <a href="#">View All</a>
Repositories	Central
Used By	1,438 artifacts

Maven Gradle SBT Ivy Grape Leiningen Buildr

```
<!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->
<dependency>
 <groupId>org.apache.poi</groupId>
 <artifactId>poi</artifactId>
 <version>4.1.2</version>
</dependency>
```

Include comment with link to declaration

Haciendo clic en el cuadro, seleccionamos el contenido que es lo que tenemos que añadir al pom.xml en una etiqueta dependency dentro de otra llamada dependencies:

```
<?xml version="1.0" encoding="UTF-8"?>
<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 /modelVersion>
<modelVersion>4.0.0</modelVersion>
<groupId>net.zabalburu</groupId>
<artifactId>Aplicacion</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>1.8</maven.compiler.source>
 <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies>
 <!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->
 <dependency>
 <groupId>org.apache.poi</groupId>
 <artifactId>poi</artifactId>
 <version>4.1.2</version>
 </dependency>
</dependencies>
```

```
</project>
```

Evidentemente, si hay más dependencias las añadimos dentro de dependencies

### Añadir Recursos

Nuestra aplicación puede necesitar el uso de recursos estáticos de diferente tipo. Si queremos incorporar dichos recursos al archivo jar, debemos indicarlo en el pom. En el ejemplo se añaden todas las imágenes png y jpg.

```
<?xml version="1.0" encoding="UTF-8"?>
<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
<modelVersion>4.0.0</modelVersion>
<groupId>net.zabalburu</groupId>
<artifactId>Aplicacion</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>1.8</maven.compiler.source>
 <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies>
 <!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->
 <dependency>
 <groupId>org.apache.poi</groupId>
 <artifactId>poi</artifactId>
 <version>4.1.2</version>
 </dependency>
</dependencies>
<build>
 <resources>
 <resource>
 <directory>src/main/java</directory>
 <includes>
 <include>**/*.png</include>
 <include>**/*.jpg</include>
 </includes>
 </resource>
 </resources>
</build>
</project>
```

NOTA IMPORTANTE: A la hora de generar una aplicación ejecutable todas las rutas a los recursos deben ser absolutas:

```
getClass().getResource("/net/zabalburu/aplicacion/...");
```

### Generando un jar Ejecutable

Para generar el archivo ejecutable vamos a emplear un **plugin de maven** ([Maven Assembly](#)) que se encarga de gestionar el proceso por nosotros. Como queremos que la aplicación sea ejecutable en cualquier sitio necesitamos incluir las dependencias (las librerías) en el jar. Por último, especificamos la clase principal mediante la etiqueta mainClass del plugin:

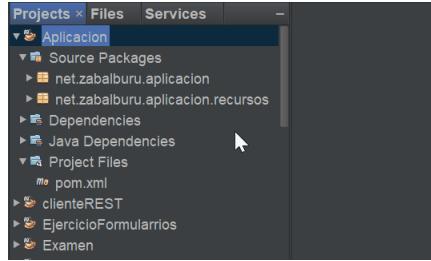
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
<modelVersion>4.0.0</modelVersion>
<groupId>net.zabalburu</groupId>
<artifactId>Aplicacion</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>1.8</maven.compiler.source>
 <maven.compiler.target>1.8</maven.compiler.target>
</properties>
<dependencies>
 <!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->
 <dependency>
 <groupId>org.apache.poi</groupId>
 <artifactId>poi</artifactId>
 <version>4.1.2</version>
 </dependency>
</dependencies>
<build>
 <resources>
 <resource>
 <directory>src/main/java</directory>
 <includes>
 <include>**/*.png</include>
 <include>**/*.jpg</include>
 </includes>
 </resource>
 </resources>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-assembly-plugin</artifactId>
 <executions>
 <execution>
 <phase>package</phase>
 <goals>
 <goal>single</goal>
 </goals>
 </execution>
 </executions>
 </plugin>
 </plugins>
</build>
</project>
```

```

<archive>
 <manifest>
 <mainClass>
 net.zabalburu.aplicacion.Programa
 </mainClass>
 </manifest>
</archive>
<descriptorRefs>
 <descriptorRef>jar-with-dependencies</descriptorRef>
</descriptorRefs>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>

```

Una vez modificado el pom.xml tenemos que **construir el proyecto**. Para ello hacemos un Build o un Clean --> Build with Dependencies:



Tras un rato, se habrán generado dos archivos con extensión jar en la carpeta target dentro del proyecto:

Nombre	Fecha de modificación	Tipo	Tamaño
archive-tmp	03/04/2020 12:56	Carpeta de archivos	
clases	03/04/2020 12:37	Carpeta de archivos	
generated-sources	03/04/2020 12:37	Carpeta de archivos	
maven-archiver	03/04/2020 12:37	Carpeta de archivos	
maven-status	03/04/2020 12:37	Carpeta de archivos	
test-classes	03/04/2020 12:37	Carpeta de archivos	
Aplicacion-1.0-SNAPSHOT.jar	03/04/2020 12:56	Executable Jar File	7 KB
Aplicacion-1.0-SNAPSHOT-jar-with-dependencies.jar	03/04/2020 12:56	Executable Jar File	6,069 KB

Haciendo **doble clic sobre el archivo con dependencias** se ejecutará la aplicación (siempre y cuando esté Java correctamente instalado). Dicho archivo podrá ejecutarse de la misma forma en cualquier ordenador con Java independientemente del Sistema Operativo. Evidentemente el nombre del archivo puede modificarse a voluntad.

Si queremos ejecutarlos desde la línea de comandos, emplearíamos

```
java -jar archivo.jar
```

