David Torres Aranha

Actividad 1.7: Tratamiento ficheros Formación. Uso de fichero "properties"

Investigación sobre el uso de los ficheros "properties" de configuración

Si se tienen muchos elementos en nuestro código que tiene valores constantes (rutas, separadores CSV, credenciales de base de datos claves de API, etc.) ¿investiga 4 formas distintas de gestionar estos elementos con valores constantes para un proyecto java "puro" (no es parte de un framework, ni carga dependencias como Maven/Gradle)?

1. **Clases de constantes (`public static final`)**: Definir atributos estáticos e inmutables dentro del propio código.
2. **Archivos .properties (`java.util.Properties`)**: Guardar pares clave-valor en un fichero externo leído mediante la API estándar de Java.
3. **Variables de entorno (`System.getenv`)**: Leer variables configuradas directamente en el sistema operativo o contenedor.
4. **Propiedades del sistema JVM (`System.getProperty`)**: Pasar parámetros en la línea de comandos al ejecutar la aplicación con la opción -D.

¿Qué ventajas tiene usar un archivo .properties frente a escribir valores directamente como constantes en el código fuente?

- **Sin recompilación**: Permite modificar la configuración sin tener que volver a compilar los archivos .java a .class.
- **Seguridad**: Evita exponer credenciales sensibles directamente en el repositorio de código fuente.
- **Mantenibilidad**: Centraliza todos los parámetros del sistema en un solo fichero fácil de editar.
- **Multi-entorno**: Facilita adaptar la aplicación a distintos entornos (desarrollo, pruebas, producción) cambiando únicamente el archivo.

¿Qué tipo de valores son adecuados para guardarse en un archivo .properties, y cuáles no? ¿Por qué?

- **Adecuados**: Parámetros de entorno (URLs, puertos), credenciales de acceso, configuraciones de formato (separadores CSV) y textos traducidos (internacionalización), porque requieren flexibilidad y cambios frecuentes sin alterar la lógica de programación.
- **No adecuados**: Lógica de negocio, algoritmos del programa y datos jerárquicos o complejos, ya que los algoritmos deben formar parte del código fuente compilado y .properties solo soporta cadenas de texto plano sencillas sin estructura interna

¿Qué limitaciones tiene el uso de archivos .properties y qué otras alternativas modernas existen (por ejemplo, YAML, JSON, bases de datos o Spring @Value)?

- **Limitaciones**: Carece de estructura jerárquica (es totalmente plano), no admite tipado explícito (todo se lee como String) y no ofrece validación nativa.
- **Alternativas modernas**:
  - **YAML / JSON**: Permiten estructuras anidadas, listas y tipado más claro.
  - **Bases de datos / Gestores de secretos**: Permiten configuraciones dinámicas y almacenamiento cifrado de credenciales (p. ej. Vault).
  - **Spring @Value**: Permite inyectar parámetros directamente en atributos de clases con conversión de tipos automática.

¿Cómo cargar desde Java un archivo .properties y acceder a sus valores? ¿Qué ocurre si una clave no existe?

Se utiliza un flujo de entrada (`InputStream`) para leer el archivo desde el classpath y cargarlo con la clase `java.util.Properties`:

```
Properties prop = new Properties();
try (InputStream input =
MiClase.class.getClassLoader().getResourceAsStream("config.properties")) {
    prop.load(input);
    String valor = prop.getProperty("mi.clave");
}
```

Si una clave no existe:

- `getProperty("clave")` devuelve `null` (no lanza excepción).
- Si se utiliza `getProperty("clave", "valorPorDefecto")`, devuelve la cadena indicada como valor por defecto.