1. Formas de gestionar valores constantes en Java "puro" y ventajas de .properties
Cuatro formas distintas de gestionar elementos constantes sin frameworks ni gestores de dependencias:

- **Clases o interfaces de constantes (public static final)**: Definición de valores inmutables en código fuente Java como miembros estáticos.
- **Tipos enumerados (enum)**: Creación de tipos definidos por el usuario con atributos constantes asociados.
- **Variables de entorno del sistema (System.getenv())**: Lectura de valores configurados directamente en el sistema operativo del entorno de ejecución.
- **Propiedades del sistema JVM (System.getProperty())**: Paso de parámetros desde la línea de comandos al iniciar la máquina virtual mediante argumentos -Dclave=valor.

Ventajas de utilizar un archivo .properties frente a constantes en código fuente:

- **Modificación sin recompilación**: Permite alterar rutas o separadores sin necesidad de volver a compilar las clases .java ni empaquetar el archivo .jar.
- **Desacoplamiento**: Desvincula los datos de entorno y parametrización respecto a la lógica de negocio de la aplicación.
- **Flexibilidad multientorno**: Facilita el cambio de configuraciones entre desarrollo, pruebas y producción cambiando únicamente el archivo físico.
- **Gestión de credenciales y seguridad**: Evita subir datos sensibles directamente al repositorio de código fuente.

2. Valores adecuados y no adecuados para guardar en un archivo .properties

- **Valores adecuados**:
    - Rutas de acceso a ficheros locales y directorios del sistema.
    - Caracteres delimitadores y separadores de formatos estructurados (como la coma , o punto y coma ; en ficheros CSV).
    - Parámetros de conexión a recursos externos (URL de base de datos, puertos de red, tiempos de tiempo de espera o *timeouts*).
    - Cadenas de texto parametrizables y etiquetas para soporte de internacionalización.
- **Valores NO adecuados**:
    - **Contraseñas y claves API sin cifrar**: Un fichero .properties es texto plano y expone credenciales críticas si no existe una capa de encriptación adicional.
    - **Estructuras de datos complejas o anidadas**: La estructura del fichero es plana (clave=valor), lo que dificulta modelar colecciones u objetos complejos.
    - **Lógica de negocio o reglas del sistema**: Los valores que determinan el flujo de ejecución del programa deben permanecer compilados dentro del código fuente Java.

3. Limitaciones de los archivos .properties y alternativas modernas

Limitaciones principales:

- Estructura lineal orientada exclusivamente a pares clave-valor planos.
- Ausencia de comprobación de tipos de datos de forma nativa (todos los valores son interpretados como cadenas de texto).
- Problemas de codificación de caracteres si no se gestiona explícitamente la codificación UTF-8 en versiones clásicas.

Alternativas modernas:

- **Ficheros YAML / JSON**: Ofrecen soporte nativo para jerarquías, anidamiento de objetos y tipos de datos estructurados.
- **Bases de datos de configuración**: Permiten persistencia dinámica y consulta de valores en tiempo de ejecución.
- **Anotaciones de frameworks (ej. Spring @Value / @ConfigurationProperties)**: Permiten la inyección de dependencias parametrizada directamente sobre atributos Java con validación de tipos.

4. Carga desde Java y comportamiento ante claves inexistentes

```
Properties prop = new Properties();
try (InputStream input = new FileInputStream("config/application.properties")) {
    prop.load(input);
} catch (IOException ex) {
    ex.printStackTrace();
}
```

Acceso a los valores: Se realiza mediante el método prop.getProperty("clave").

Comportamiento si la clave no existe:

- Si se utiliza getProperty("clave_inexistente"), el método devuelve null.
- Si se invoca una operación sobre la devolución sin verificar (por ejemplo .trim() o .split()), el programa lanzará una excepción NullPointerException.
- Para evitarlo, se puede usar getProperty("clave_inexistente", "valor_por_defecto"), devolviendo el valor por defecto en caso de no encontrar la clave.