1. Archivos de Texto (Plano)

Lectura (de Fichero a ArrayList)

Usamos BufferedReader para leer líneas y split() para partirlas.

```
// Lectura (Texto)
ArrayList<Producto> lista = new ArrayList<>();
try (BufferedReader br = new BufferedReader(new
FileReader(ruta))) {
    String linea;
    while ((linea = br.readLine()) != null) {
        String[] campos = linea.split(";"); // Separador
        if (campos.length == 3) {
            int id = Integer.parseInt(campos[0]);
            String nombre = campos[1];
            double precio = Double.parseDouble(campos[2]);
            lista.add(new Producto(id, nombre, precio));
        }
    }
} catch (IOException | NumberFormatException e) {
            e.printStackTrace();
}
```

Escritura (de ArrayList a Fichero)

Usamos PrintWriter (o BufferedWriter) para escribir líneas formateadas.

```
// Escritura (Texto)
try (PrintWriter pw = new PrintWriter(new FileWriter(ruta))) {
    for (Producto p : listaProductos) {
        String linea = p.getId() + ";" + p.getNombre() + ";" +
p.getPrecio();
        pw.println(linea); // Escribe la línea y un salto
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

2. Archivos Binarios (Serialización)

Idea: Guardar/leer el objeto Java directamente. Es el más fácil si se permite.

¡Requisito Clave! Tu clase Producto debe implementar Serializable.

```
public class Producto implements java.io.Serializable { ... }
```

Lectura (de Fichero a ArrayList)

Usamos ObjectInputStream y leemos el ArrayList **entero de una sola vez**.

```
// Lectura (Binario)
ArrayList<Producto> lista = new ArrayList<>();
try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(ruta))) {
    // Leemos el ArrayList completo que guardamos
    lista = (ArrayList<Producto>) ois.readObject();

} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
```

Escritura (de ArrayList a Fichero)

Usamos ObjectOutputStream y escribimos el ArrayList **entero de una sola vez**.

```
// Escritura (Binario)
try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(ruta))) {
    // Escribimos el ArrayList completo
    oos.writeObject(listaProductos);

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

3. Archivos XML (DOM)

Idea: Cargar el XML en memoria como un árbol de Nodos. Usamos org.w3c.dom, javax.xml.parsers y javax.xml.transform.

Lectura (de Fichero a ArrayList)

Parseamos el archivo y recorremos los nodos Element para extraer los datos.

```
// Lectura (XML DOM)
ArrayList<Producto> lista = new ArrayList<>();
DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(new File(ruta));
doc.getDocumentElement().normalize(); // Buena práctica

// Obtenemos todos los nodos con la etiqueta <producto>
NodeList nList = doc.getElementsByTagName("producto");

for (int i = 0; i < nList.getLength(); i++) {
    Element e = (Element) nList.item(i);

    // Leer atributo 'id'
    int id = Integer.parseInt(e.getAttribute("id"));
    // Leer texto de etiqueta hija <nombre>
    String nombre =
e.getElementsByTagName("nombre").item(0).getTextContent()
;
    double precio =
Double.parseDouble(e.getElementsByTagName("precio").item
(0).getTextContent());

    lista.add(new Producto(id, nombre, precio));
}
```

Escritura (de ArrayList a Fichero)

Creamos el Document en memoria, añadimos elementos iterando la lista y al final lo "transformamos" a un archivo.

```
// Escritura (XML DOM)
DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.newDocument();

// 1. Crear raíz <catalogo>
Element raiz = doc.createElement("catalogo");
doc.appendChild(raiz);

// 2. Recorrer lista y crear nodos
for (Producto p : listaProductos) {
    Element prodElem = doc.createElement("producto");
    raiz.appendChild(prodElem);

    // Añadir atributo: <producto id="...">
    prodElem.setAttribute("id", String.valueOf(p.getId()));

    // Añadir elemento hijo <nombre>...</nombre>
    Element nomElem = doc.createElement("nombre");
    nomElem.setTextContent(p.getNombre());
    prodElem.appendChild(nomElem);

    // Añadir elemento hijo <precio>...</precio>
    Element precioElem = doc.createElement("precio");
    precioElem.setTextContent(String.valueOf(p.getPrecio()));
    prodElem.appendChild(precioElem);
}

// 3. Guardar el 'doc' en un fichero
Transformer t =
TransformerFactory.newInstance().newTransformer();
t.setOutputProperty(javax.xml.transform.OutputKeys.INDENT,
"yes"); // Para que se vea bonito
t.transform(new DOMSource(doc), new StreamResult(new
File(ruta)));
```

4. Archivos de Acceso Aleatorio (RandomAccessFile)

Idea: Cada registro (objeto) debe ocupar exactamente el mismo tamaño en bytes.

Problema: Los String tienen tamaño variable.

Solución: Se define un tamaño fijo para el String (ej. 20 caracteres) y se rellena (padding) o trunca.

Cálculo de Tamaño (Ejemplo):

- int id: 4 bytes
- double precio: 8 bytes
- String nombre [20 chars]: 20 * 2 = 40 bytes (porque 1 char en Java = 2 bytes)
- **TAMAÑO_REGISTRO = 4 + 8 + 40 = 52 bytes**

Lectura (de Fichero a ArrayList)

Usamos RandomAccessFile(ruta, "r"). Leemos en bloques fijos de 52 bytes (según el cálculo) hasta el fin de fichero (EOF).

```
// Lectura (Aleatorio)
final int TAMANO_NOMBRE = 20; // ¡Debe ser el mismo que al escribir!
ArrayList<Producto> lista = new ArrayList<>();

try (RandomAccessFile raf = new RandomAccessFile(ruta, "r")) {

    // Mientras el puntero no llegue al final
    while (raf.getFilePointer() < raf.length()) {
        int id = raf.readInt();
        double precio = raf.readDouble();

        // Leer el array de chars de tamaño fijo
        char[] nombreChars = new char[TAMANO_NOMBRE];
        for (int i = 0; i < TAMANO_NOMBRE; i++) {
            nombreChars[i] = raf.readChar();
        }
        String nombre = new String(nombreChars).trim(); // .trim() quita espacios sobrantes

        lista.add(new Producto(id, nombre, precio));
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Escritura (de ArrayList a Fichero)

Usamos RandomAccessFile(ruta, "rw"). Escribimos campo a campo, asegurando el tamaño fijo del String.

```
// Escritura (Aleatorio)
final int TAMANO_NOMBRE = 20; // 20 caracteres fijos

try (RandomAccessFile raf = new RandomAccessFile(ruta, "rw")) {
    for (Producto p : listaProductos) {
        raf.seek(raf.length()); // Posicionarse al final para añadir

        raf.writeInt(p.getId());        // Escribe 4 bytes
        raf.writeDouble(p.getPrecio()); // Escribe 8 bytes

        // Preparar el String de tamaño fijo (40 bytes)
        StringBuilder nombreFijo = new StringBuilder(p.getNombre());
        nombreFijo.setLength(TAMANO_NOMBRE); // Trunca o rellena con '\0'

        raf.writeChars(nombreFijo.toString()); // Escribe 40 bytes
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

1. Archivos de Texto (Plano)

Lectura: El código para leer un archivo de texto línea por línea con BufferedReader es correcto. Utiliza split(":") para dividir los campos, lo cual es un enfoque estándar. La conversión de tipos con Integer.parseInt y Double.parseDouble también es correcta.

Escritura: El uso de PrintWriter para escribir en el fichero es adecuado. El código itera sobre una lista y formatea cada objeto Producto en una línea de texto.

Observación: El código de escritura tiene un pequeño error de sintaxis en la concatenación de strings. Debería ser: String linea = p.getId() + ":" + p.getNombre() + ":" + p.getPrecio(); En el documento aparece con + " + y +*+ que parecen erratas.

2. Archivos Binarios (Serialización)

Concepto: La explicación es clara y precisa. La clave es que la clase Producto debe implementar la interfaz Serializable.

Lectura: El código para leer el ArrayList completo de una vez usando ObjectInputStream es correcto y una de las grandes ventajas de la serialización.

Escritura: Igualmente, el código para escribir el ArrayList completo con ObjectOutputStream es correcto y eficiente.

3. Archivos XML (DOM)

Concepto: La idea de cargar el documento XML en memoria como un árbol de nodos (DOM) está bien explicada.

Lectura: El proceso es correcto:
Se crea un DocumentBuilder para parsear el fichero.
Se obtiene la lista de nodos con la etiqueta "producto".
Se itera sobre los nodos, extrayendo el atributo "id" y el contenido de las etiquetas hijas "nombre" y "precio".

Escritura: La lógica para crear el documento XML también es correcta:
Se crea un Document vacío en memoria.
Se crea el elemento raíz y se añade al documento.
Se recorre la lista de productos, y para cada uno, se crea un elemento <producto> con su atributo id y sus elementos hijos <nombre> y <precio>.
Finalmente, se utiliza Transformer para volcar la estructura de memoria (doc) a un fichero físico, con indentación para que sea legible.

4. Archivos de Acceso Aleatorio (RandomAccessFile)

Concepto: La idea fundamental está muy bien explicada: cada registro debe tener un tamaño fijo. El problema de los String de tamaño variable y su solución (usar un tamaño fijo y rellenar/truncar) también está correctamente planteado.
Cálculo de Tamaño: El cálculo de ejemplo es correcto. Un int ocupa 4 bytes, un double 8 bytes, y un char en Java ocupa 2 bytes, por lo que un String de 20 caracteres requiere 40 bytes. El tamaño total del registro sería 52 bytes.

Lectura: El código lee campos en el orden en que fueron escritos (int, double, char[]). El uso de .trim() es una buena práctica para eliminar los espacios de relleno al reconstruir el String.

Escritura: El código se posiciona al final del fichero para añadir nuevos registros (raf.seek(raf.length())), lo cual es correcto para operaciones de inserción. El uso de StringBuilder para fijar la longitud del nombre es una forma eficiente de asegurar el tamaño fijo del String.