

UNIDAD 1 parte 4: Gestión de Ficheros XML StAX

Índice

1.	<i>Acceso a ficheros XML con StAX</i>	2
1.1	StAX vs. DOM y SAX	2
1.2	Requisitos y Componentes para Trabajar con StAX	2
2.	<i>APIs para trabajar con StAX</i>	3
2.1	Crear la fábrica StAX lectura	3
2.2	Crear la Fábrica StAX - WRITER	6
3.	<i>APIs para trabajar con StAX modelo cursor</i>	8
3.1	Creación del XMLStreamReader	8
3.2	Lectura con XMLStreamReader	9
3.3	Creación de un XMLStreamWriter.	12
3.4	Escritura con XMLStreamWriter.	13
3.5	Importancia de Cerrar los Streams y Parsers	14
4.	<i>APIs para trabajar con StAX-Modelo evento</i>	14
4.1	Creacion de un XMLEventReader	15
4.2	Lectura con XMLEventReader	16
4.3	¿Qué es QName y por qué se usa en StAX?	17
4.4	Creación de un XMLEventWriter	18
4.5	Escritura con XMLEventWriter	19

1. Acceso a ficheros XML con StAX

StAX (Streaming API for XML) es una API **pull** para procesar documentos XML **de forma secuencial y controlada por el programador**.

StAX es un enfoque **moderno y eficiente** para el procesamiento de XML en Java. Su principal innovación reside en su modelo de procesamiento "**pull**", que marca una diferencia clave respecto a las otras APIs (DOM y SAX).

A diferencia de DOM **no construye un árbol en memoria**; a diferencia de SAX, **el control del avance lo tiene el código que lee (pull), no el parser (push)**.

Combina las ventajas de SAX (bajo consumo de memoria) y DOM (control del flujo y capacidad de escritura) mediante dos modelos:

- ✓ **cursor** (pull, XMLStreamReader/XMLStreamWriter)
- ✓ **iterator/event** (pull por eventos, XMLEventReader/XMLEventWriter).

Características principales:

- **Procesamiento rápido y de bajo consumo de memoria**, adecuado para documentos grandes.
- **Permite leer y también escribir XML en streaming** (XMLStreamReader / XMLStreamWriter y XMLEventReader / XMLEventWriter).
- **El programador controla el flujo** llamando a `next() / nextEvent()` y consultando el evento actual.
- Los **eventos/tokens típicos** son: START_ELEMENT, END_ELEMENT, CHARACTERS, START_DOCUMENT, END_DOCUMENT, PROCESSING_INSTRUCTION, etc.
- **Ofrece dos modelos:**
 - Cursor (XMLStreamReader/XMLStreamWriter) — más liviano y de bajo nivel
 - Event (XMLEventReader/XMLEventWriter) — orientado a objetos (StartElement, Characters,EndElement).

1.1 StAX vs. DOM y SAX

Para entender mejor las ventajas de StAX, es útil compararlo directamente con DOM y SAX:

Característica	DOM (Document Object Model)	SAX (Simple API for XML)	StAX (Streaming API for XML)
Modelo de Proceso	Aleatorio (Random Access)	"Push" (El parser empuja/notifica)	"Pull" (El código tira / solicita)
Consumo de Memoria	Alto. Carga el documento completo en una estructura de árbol.	Bajo. Procesa elemento por elemento sin cargar todo.	Bajo. Procesa secuencialmente sin cargar todo.
Control del Flujo	Completo. Acceso directo a cualquier nodo.	Inverso. El parser controla el flujo y llama a <i>handlers</i> del programador.	Completo. El programador controla el flujo con llamadas explícitas (<code>next() / nextEvent()</code>).
Capacidad de Escritura	Fácil. Se modifica el árbol en memoria y luego se serializa.	Difícil/No natural. SAX es fundamentalmente de solo lectura .	Fácil. Proporciona interfaces directas para escribir (XMLStreamWriter/XMLEventWriter).
Documentos Grandes	Mal rendimiento/problemas de memoria.	Muy adecuado.	Muy adecuado.

1.2 Requisitos y Componentes para Trabajar con StAX

Mientras que en SAX el programador debe **implementar interfaces handler** (como ContentHandler) para recibir los eventos "empujados" por el *parser*, en StAX el programador **"solicita"** los eventos directamente.

Componentes Necesarios

Los componentes básicos son similares, pero la **clase central** para la interacción es distinta:

1. **Un Parser XML:** Similar a SAX y DOM, necesitas un motor capaz de leer y analizar XML. StAX es una API estándar de Java (JAXP/JDK), por lo que las implementaciones de *parser* modernas lo soportan.
2. **Las Clases StAX:** Principalmente las que se encuentran en el paquete **javax.xml.stream**. Ejemplo: `XMLInputFactory`, `XMLOutputFactory`, `XMLStreamReader`, `XMLStreamWriter`, `XMLEventReader`, `XMLEventWriter`, `StartElement`, `EndElement`, `Characters`, `XMLEvent`, `XMLEventFactory`.
3. **Un Documento XML.**

StAX y modelos utilizados

La principal decisión al usar StAX recae en **elegir el modelo**:

- Utiliza **XMLStreamReader y XMLStreamWriter** (*Modelo Cursor*) si la velocidad y la eficiencia son la máxima prioridad, ya que es más ligero y rápido. Preferible para rendimiento y ficheros grandes.
- Utiliza **XMLEventReader y XMLEventWriter** (*Modelo Eventos*) si se prefiere una programación más orientada a objetos, donde cada evento es un objeto tipificado y el código puede ser más legible. Ligeramente más coste por creación de objetos.

2. APIs para trabajar con StAX

La API StAX es estándar en Java (a partir de Java 6) y se trabaja principalmente a través del paquete **javax.xml.stream**.

Para comenzar a trabajar con StAX, lo primero es entender las **Fábricas** que te permiten obtener los *parsers* (lectores/escritores), tal como ocurre con DOM (DocumentBuilderFactory) y SAX (SAXParserFactory).

En StAX, las clases centrales para la configuración y creación de *parsers* son: **XMLInputFactory** y **XMLOutputFactory**. Ambas se encuentran en el paquete javax.xml.stream.

El objetivo de llamar a estas clases centrales es obtener una instancia de la fábrica y configurarla para el procesamiento (ej., activar namespaces o validación, etc.)

Esta configuración es **común y válida para ambos modelos**:

- **Modelo Cursor:** lectura/escritura basada en llamadas imperativas (`next()`, `writeStartElement()`, etc.)
- **Modelo por Eventos:** lectura/escritura basada en objetos tipificados (`XMLEvent`, `StartElement`, `Characters`, etc.)

2.1 Crear la fábrica StAX lectura

XMLInputFactory (Para Leer XML)

La **XMLInputFactory** es la clase que se utiliza para configurar y crear instancias de los parses lectores StAX, ya sea el `XMLStreamReader` (Modelo Cursor) o el `XMLEventReader` (Modelo Eventos).

Función Principal

- **Creación de Lectores:** Su método principal es `createXMLStreamReader()` o `createXMLEventReader()`, que toma como argumento la fuente del documento (un `InputStream`, `Reader`, o `Source`).
- **Configuración:** Permite establecer propiedades en el parser subyacente (como el manejo de espacios de nombres o la validación), de forma similar a como lo hacía la `SAXParserFactory`.

Métodos Clave

Método	Propósito
<code>XMLInputFactory.newInstance()</code>	Método estático para crear y obtener una instancia de la fábrica.
<code>createXMLStreamReader(Source source)</code>	Crea y devuelve un XMLStreamReader (Modelo Cursor).
<code>createXMLEventReader(Source source)</code>	Crea y devuelve un XMLEventReader (Modelo Eventos).
<code>setProperty(String name, Object value)</code>	Establece una propiedad de configuración para el <i>parser</i> (ej., para la validación).
<code>getProperty(String name)</code>	Recupera el valor de una propiedad de configuración.

```
// 1. OBTENER la instancia de la fábrica
XMLInputFactory factory = XMLInputFactory.newInstance();
```

Configuración en XMLInputFactory

La configuración del *parser* (como el manejo de *namespaces*, la validación, o la gestión de DTDs) se realiza mediante llamadas a `setProperty(String name, Object value)` justo después de obtener la instancia de la fábrica, pero **antes** de llamar a `createXMLStreamReader()` o `createXMLEventReader()`.

```
public abstract void setProperty(String name, Object value)
throws IllegalArgumentException
```

Elemento	Tipo	Descripción
<code>name</code>	<code>String</code>	El nombre de la propiedad a configurar. Se recomienda usar las constantes definidas en <code>XMLInputFactory</code> .
<code>value</code>	<code>Object</code>	El valor que se asignará a la propiedad. Comúnmente es un <code>Boolean.TRUE</code> o <code>Boolean.FALSE</code> .
Excepción	<code>IllegalArgumentException</code>	Se lanza si la implementación del <i>parser</i> no soporta la propiedad solicitada.

Propiedades Comunes de XMLInputFactory

Nombre de la Propiedad (name)	Valor (value)	Propósito
<code>IS_NAMESPACE_AWARE</code>	<code>Boolean.TRUE</code> / <code>Boolean.FALSE</code>	Indica si el <i>parser</i> debe reconocer y manejar espacios de nombres (namespaces) . Recomendado: TRUE
<code>IS_VALIDATING</code>	No soportada	No activa validación en StAX. Ignorada o lanza excepción.
<code>IS_COALESCING</code>	<code>Boolean.TRUE</code> / <code>Boolean.FALSE</code>	Si es TRUE, el <i>parser</i> combina bloques de texto adyacentes (CHARACTERS) en un solo evento, simplificando la lectura. Recomendado: TRUE
<code>SUPPORT_DTD</code>	<code>Boolean.TRUE</code> / <code>Boolean.FALSE</code>	Permite procesar DOCTYPE y entidades DTD, pero no activa la validación.

```
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, Boolean.TRUE);
```

Validación con DTD o XSD

La especificación de **StAX (JSR-173)** fue diseñada para ser un *parser* de bajo nivel, rápido y eficiente. Por diseño, la **validación completa** de esquemas (XSD o DTD) **no es una funcionalidad obligatoria ni nativa del XMLInputFactory** estándar, y por lo tanto, no es portable.

- **StAX no permite validar contra DTD directamente, incluso con IS_VALIDATING=true.** Aunque puede procesar documentos con DOCTYPE si SUPPORT_DTD está activado, **no comprueba si el contenido cumple el DTD.**

Recomendación:

- Validar previamente con **SAX o DOM** si se requiere validación estructural obligatoria.
- Luego procesar con StAX si el documento es válido.

- **Aunque StAX no valida por sí mismo con XSD, puede integrarse con la API javax.xml.validation para validar documentos XML contra un esquema XSD antes de procesarlos.** Esta validación es portable y recomendada.

Requisitos:

- El documento XML debe incluir referencia al esquema XSD mediante atributos **xmlns:xsi** y **xsi:schemaLocation** o **xsi:noNamespaceSchemaLocation**.

```
<libros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="libros.xsd">
    ...
</libros>
```

- **¿Cuándo se puede usar javax.xml.validation.Validator con StAXSource?**

Tipo de esquema	¿Se puede usar Validator con StAXSource ?	¿Funciona con StAX puro?	Comentario
XSD	<input checked="" type="checkbox"/> Sí	<input checked="" type="checkbox"/> Sí	Recomendado y soportado oficialmente
DTD	<input type="checkbox"/> No	<input type="checkbox"/> No	No está soportado por javax.xml.validation

Validación con XSD (XML Schema Definition)

Aunque StAX no valida por sí mismo, la **integración con la API de validación JAXP** (javax.xml.validation) es la forma **portable y recomendada** para validar contra esquemas.

Documento XML: El documento XML debe utilizar **atributos de namespace** para referenciar el esquema XSD, principalmente xmlns para el *namespace* y xsi:schemaLocation (o xsi:noNamespaceSchemaLocation) para la ubicación del archivo .xsd.

```
<libros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="libros.xsd">
    ...
</libros>
```

Configuración Esquema (XSD):

Para garantizar que la validación se realice contra el XSD se suele trabajar con propiedades específicas, algunas de las cuales provienen de las constantes de javax.xml.XMLConstants.

- ✗ No se debe usar IS_VALIDATING, ya que no está soportado por StAX y puede lanzar excepción

```
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, Boolean.TRUE);
//Desactiva el soporte DTD. Si el documento usa XSD, evitamos la sobrecarga o
//posibles problemas de seguridad del DTD.
factory.setProperty(XMLInputFactory.SUPPORT_DTD, Boolean.FALSE);
```

Ejemplo completo de validación con XSD MODO 1 (CURSOR)

```
// 1. Crear el esquema XSD
SchemaFactory schemaFactory = SchemaFactory.newInstance
        (XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = schemaFactory.newSchema(new File(rutaXSD));
Validator validator = schema.newValidator();
```

```

// 2. Crear lector StAX con configuración adecuada
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, Boolean.TRUE);
factory.setProperty(XMLInputFactory.SUPPORT_DTD, Boolean.FALSE);

XMLStreamReader reader = factory.createXMLStreamReader(
    new FileInputStream(rutaXML), "UTF-8");

// 3. Validar usando StAXSource
validator.validate(new StAXSource(reader));
System.out.println("Documento válido según XSD.");

```

Configuración de Seguridad (Recomendada para ambos)

Independientemente de si se usa DTD o XSD, es una **práctica profesional estándar** mitigar las vulnerabilidades de seguridad comunes (como los ataques de Entidad Externa, XXE). Esto se hace limitando el acceso a recursos externos.

Propiedad (Técnica)	Valor	Propósito
IS_SUPPORTING_EXTERNAL_ENTITIES	Boolean.FALSE	Máxima Seguridad: Desactiva que el parser intente buscar y cargar DTDs o entidades externas (es vital para la seguridad).
XMLConstants.ACCESS_EXTERNAL_DTD	"" (String vacío)	Seguridad Reforzada: Restringe el acceso a cualquier DTD externo (propiedad estándar JAXP).
XMLConstants.FEATURE_SECURE_PROCESSING	Boolean.TRUE	limitar los recursos que el parser puede consumir y restringir el acceso a ciertos componentes externos

```

XMLInputFactory factory = XMLInputFactory.newInstance();
// 1. CONFIGURACIÓN DE SEGURIDAD (ANTI-XXE)
// a) Desactivar el soporte para entidades externas
factory.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES,
    Boolean.FALSE );
// b) Restringir el acceso a DTDs externos y otros recursos (propiedad JAXP
estándar)
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "" );
// El String vacío significa que no se permite el acceso a URLs externas
// c) Activar el procesamiento seguro: impone límites de recursos y restricciones
//generales
factory.setProperty(XMLConstants.FEATURE_SECURE_PROCESSING,
    Boolean.TRUE );

```

2.2 Crear la Fábrica StAX - WRITER

- La clase XMLOutputFactory se utiliza para configurar y crear instancias de los parsers escritores StAX, tanto en el **modelo cursor (XMLStreamWriter)** como en el modelo por eventos (XMLEventWriter).
- Se utiliza para **generar documentos XML válidos y de gran tamaño** a partir de datos estructurados en una aplicación Java, sin consumir mucha memoria.
- **Ambos modelos son una forma eficiente y segura de generar XML en streaming.**
 - En vez de construir todo el documento en memoria, **vas escribiendo elemento a elemento**.
 - Se utilizan porque estos archivos son **demasiado grandes** para ser construidos con el Modelo DOM (que consumiría mucha memoria) y su naturaleza **secuencial**.

- Es ideal cuando los **datos son muy grandes**, necesitas garantizar que el XML sea siempre bien formado y/oquieres integrarlo en cadenas de procesos (transformaciones, exportaciones, feeds).

Nota: feed es un documento (generalmente XML o, más modernamente, JSON) que contiene una lista o secuencia de elementos de datos que cambian o se actualizan con frecuencia.

Su principal propósito es permitir que otros programas o servicios web consuman y procesen esa información de forma regular y automatizada.

Ejemplos: Feeds de Noticias en formato RSS o Atom, Feeds de Catálogo de Productos (E-commerce) que tienen que generar su catálogo completo (a menudo con miles o millones de productos) a plataformas de terceros para publicidad, comparaciones de precios o listados de mercado en formato XML requerido por Google Merchant Center o Amazon Marketplace).

Función Principal de XMLOutputFactory

- **Creación de escritores:**
 - `createXMLStreamWriter()` → para el modelo cursor
 - `createXMLEventWriter()` → para el modelo por eventos
- **Configuración previa:** permite establecer propiedades que afectan al comportamiento del escritor, como la gestión automática de namespaces o el cierre del stream subyacente.

Métodos

Método	Propósito
<code>XMLOutputFactory.newInstance()</code>	Método estático para crear y obtener una instancia de la fábrica.
<code>createXMLStreamWriter(OutputStream stream, String encoding)</code>	Crea y devuelve un XMLStreamWriter asociado a una salida y especifica la codificación (Ej: "UTF-8").
<code>createXMLStreamWriter(Writer stream)</code>	Crea un escritor que genera XML directamente en un Writer de caracteres.
<code>setProperty(String name, Object value)</code>	Establece una propiedad de configuración para el escritor.
<code>createXMLEventWriter(OutputStream stream, String encoding)</code>	Crea y devuelve un XMLEventWriter asociado a una salida (OutputStream) y especifica la codificación (por ejemplo, "UTF-8").
<code>createXMLEventWriter(Writer stream)</code>	Crea un escritor por eventos que escribe en un Writer usando objetos XMLEvent.

```
// 1. OBTENER la instancia de la fábrica para escribir
XMLOutputFactory factory = XMLOutputFactory.newInstance();
```

Configuración en XMLOutputFactory

La configuración es generalmente más simple que en la lectura. Se realiza mediante `setProperty(String name, Object value)` antes de crear el **XMLStreamWriter**.

Nombre de la Propiedad	Valor	Propósito
<code>IS_REPAIRING_NAMESPACES</code>	<code>Boolean.TRUE</code> / <code>Boolean.FALSE</code>	Si es TRUE , el escritor creará automáticamente prefijos de namespaces (Ej: ns1:) si se utilizan URIs sin declarar. Recomendado: TRUE.
<code>IS_AUTO_CLOSE_OUTPUT</code>	<code>Boolean.TRUE</code> / <code>Boolean.FALSE</code>	Si es TRUE , el XMLStreamWriter cerrará el OutputStream subyacente al llamar a <code>writer.close()</code> .

```
// Configurar Propiedades (Ej: Reparar Namespaces)
factory.setProperty(XMLOutputFactory.IS_REPAIRING_NAMESPACES, Boolean.TRUE);
```

El problema que esta propiedad soluciona ocurre el programador **define un elemento que pertenece a un namespace** (usando su **URI**), pero **no ha declarado explícitamente** el prefijo (`xmlns:prefijo="..."`) que lo acompaña.

¿Qué hace la "Reparación" (TRUE)?

1. **Le dicee al Writer:** "Quiero un elemento del namespace con la URI `http://misdatos.com/v1`."
2. **El Writer revisa:** "¿Ya existe un prefijo declarado (como `xmlns:datos="http://misdatos.com/v1"`)?"
3. **Si NO existe:** El XMLStreamWriter **automáticamente inventa un prefijo** (generalmente ns1, ns2, etc.) y lo declara en el elemento raíz o en el elemento más cercano.

Esto simplifica tu código, ya que no tienes que preocuparte por gestionar y escribir las declaraciones `xmlns:prefijo` a mano.

Ejemplo de Escritura

Modo	Código Java (Simplificado)	XML de Salida
Reparación (TRUE) (Recomendado)	<code>writer.writeStartElement("http://misdatos.com/v1", "libro");</code>	<code><ns1:libro xmlns:ns1="http://misdatos.com/v1">...</ns1:libro></code>
Sin Reparación (FALSE) (Requiere gestión manual)	<code>writer.writeStartElement("http://misdatos.com/v1", "libro");</code>	ERROR (A menos que declares <code>xmlns:prefijo</code> manualmente antes).

3. APIs para trabajar con StAX modelo cursor

Una vez que hemos obtenido y configurado correctamente la instancia de `XMLInputFactory`, el siguiente paso es crear el lector XML que implementa el **modelo cursor**: el `XMLStreamReader`.

3.1 Creación del XMLStreamReader

El Modelo Cursor es de bajo nivel y utiliza un método simple para avanzar y devolver códigos de token.

Método	Descripción con ejemplo
<code>createXMLStreamReader(InputStream stream)</code>	Crea el lector desde un flujo de bytes, usando la codificación detectada automáticamente o por defecto. Ejemplo: <code>XMLStreamReader lector = factory.createXMLStreamReader(new FileInputStream("archivo.xml"));</code>
<code>createXMLStreamReader(InputStream stream, String encoding)</code>	Crea el lector desde un flujo de bytes, especificando la codificación (por ejemplo "UTF-8"). Ejemplo: <code>XMLStreamReader lector = factory.createXMLStreamReader(new FileInputStream("archivo.xml"), "UTF-8");</code>
<code>createXMLStreamReader(Reader reader)</code>	Crea el lector desde un flujo de caracteres, útil si ya se ha gestionado la codificación. Ejemplo: <code>XMLStreamReader lector = factory.createXMLStreamReader(new FileReader("archivo.xml"));</code>
<code>createXMLStreamReader(Source source)</code>	Crea el lector desde una fuente abstracta (StreamSource, DOMSource, etc.), útil en transformaciones. Ejemplo: <code>XMLStreamReader lector = factory.createXMLStreamReader(new StreamSource(new File("archivo.xml")));</code>
<code>createXMLStreamReader(String systemId, InputStream stream)</code>	Crea el lector desde un flujo de bytes, asociando una URI como identificador del sistema. Ejemplo: <code>XMLStreamReader lector = factory.createXMLStreamReader("http://ejemplo.com/catalogo", new FileInputStream("archivo.xml"));</code>
<code>createXMLStreamReader(String systemId, Reader reader)</code>	Crea el lector desde un flujo de caracteres, asociando una URI como identificador del sistema. Ejemplo: <code>XMLStreamReader lector = factory.createXMLStreamReader("http://ejemplo.com/catalogo", new FileReader("archivo.xml"));</code>

Ejemplo:

```

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.XMLConstants;
import javax.xml.stream.XMLStreamException;
import java.io.FileReader;
import java.io.IOException; // Necesario para FileReader
import java.io.FileNotFoundException;
public class ConfiguracionStAX {
    public XMLStreamReader configurarYCrearReader(String rutaArchivo) {
        XMLInputFactory factory = XMLInputFactory.newInstance();
        try {
            // 2. CONFIGURAR PROPIEDADES CLAVE (Validación y Seguridad)
            // a) Activar la validación del esquema (DTD o XSD)
            factory.setProperty(XMLInputFactory.IS_VALIDATING, Boolean.TRUE);
            // b) Activar el procesamiento seguro (anti-XXE)
            factory.setProperty(XMLConstants.FEATURE_SECURE_PROCESSING,
                Boolean.TRUE);

            // 3. CREAR el lector (Reader)
            XMLStreamReader reader = factory.createXMLStreamReader(new
                FileReader(rutaArchivo));
            return reader;

        } catch (FileNotFoundException e) {
            System.err.println("ERROR: Archivo XML no encontrado en la ruta: " +
                rutaArchivo);
        } catch (IllegalArgumentException e) {
            System.err.println("ERROR: La implementación del Parser no soporta
                alguna propiedad configurada.");
        } catch (XMLStreamException e) {
            // Captura errores de sintaxis, parseo y validación (si IS_VALIDATING es TRUE)
            System.err.println("ERROR DE PARSEO/VALIDACIÓN en XML: " +
                e.getMessage());
        }
        return null; // Devuelve null si falla la creación
    }
}

```

3.2 Lectura con XMLStreamReader

El **XMLStreamReader** es la interfaz que implementa el modelo "Pull". El programador controla el flujo llamando al método `next()` para avanzar al siguiente *token* o evento.

Métodos para la Lectura

Propósito	Método	Tipo de Retorno	Constante (XMLStreamConstants)
Control de Flujo	<code>next()</code>	int	Avanza el cursor al siguiente <i>token</i> y devuelve su tipo.
	<code>hasNext()</code>	boolean	Devuelve true si hay más <i>tokens</i> para leer.
Identificación	<code>getEventType()</code>	int	Devuelve el tipo del <i>token</i> actual (útil dentro del bucle).
Información del Elemento	<code>getLocalName()</code>	String	Nombre local del elemento actual (si el <i>token</i> es START_ELEMENT o END_ELEMENT).
	<code>getText()</code>	String	Contenido de texto (si el <i>token</i> es CHARACTERS).
Atributos	<code>getAttributeCount()</code>	int	Número de atributos del elemento actual.
	<code>getAttributeLocalName(i)</code>	String	Nombre local del atributo en la posición i.
	<code>getAttributeValue(String namespaceURI, String localName)</code>	String	Valor del atributo (la forma más común de acceder).

Constantes Clave de XMLStreamConstants

El XMLStreamReader utiliza constantes estáticas de la interfaz **XMLStreamConstants** para identificar el tipo de *token* que se está procesando.

Estas son las constantes más importantes que representan los diferentes tipos de **tokens** o **eventos** que el *parser* encuentra al leer el documento XML secuencialmente.

Constante (int value)	Nombre de la Constante	Propósito
1	START_ELEMENT	Indica el inicio de una etiqueta XML (Ej: <libro>). La información sobre atributos está disponible en este momento.
2	END_ELEMENT	Indica el cierre de una etiqueta XML (Ej: </libro>).
3	CHARACTERS	Representa el contenido de texto dentro de un elemento (Ej: "Título" dentro de <título>Título</título>).
4	CDATA	Representa un bloque de datos de caracteres no analizados (CDATA).
5	COMMENT	Representa un comentario XML (Ej: ` `).
7	PROCESSING_INSTRUCTION	Representa una instrucción de procesamiento (Ej: `<?php echo "hola"; ?>`).
8	`DTD`	Representa la declaración DTD (Document Type Definition) que define la estructura del documento.
9	`ENTITY_REFERENCE`	Representa una referencia a una entidad (Ej: `©`).
10	`START_DOCUMENT`	Indica el inicio del documento (después de la declaración XML).
11	`END_DOCUMENT`	Indica el final del documento (el último token que se lee).
12	`NAMESPACE`	Indica un evento de declaración de namespace (solo relevante en ciertas configuraciones).

Uso en el Código (Modelo Cursor)

En el código, estas constantes se utilizan dentro del bloque switch para determinar qué acción tomar con el *token* actual.

La clave del Modelo Cursor es que el reader avanza token por token, y el programador decide qué tokens (START_ELEMENT, CHARACTERS, etc.) son relevantes para la lógica.

```
// Ejemplo de uso de las constantes en el bucle principal de StAX
while (reader.hasNext()) {
    // Aquí es donde se obtiene el valor entero de la constante
    int eventType = reader.next();
    // Uso de la expresión switch con flecha (->)
    switch (eventType) {
        case XMLStreamConstants.START_ELEMENT -> {
            // Lógica para procesar la apertura de etiqueta y sus atributos
            String nombre = reader.getLocalName();
            System.out.println("-> Apertura de: " + nombre);
        }
        case XMLStreamConstants.CHARACTERS -> {
            // Lógica para extraer el contenido de texto
            String texto = reader.getText().trim();
            if (!texto.isEmpty()) {
                System.out.println("-> Contenido: " + texto);
            }
        }
        case XMLStreamConstants.END_ELEMENT -> {
            // Lógica para procesar el cierre de etiqueta (ej: final de un objeto)
            System.out.println("Cierre de: " + reader.getLocalName());
        }
        case XMLStreamConstants.START_DOCUMENT ->
            // Para casos de una sola línea, se puede omitir el bloque {}
            System.out.println("Documento listo para procesar.");
            // Podemos añadir un default si es necesario
    }
}
```

Ejemplo: Dado el siguiente XML

```
<?xml version="1.0" encoding="UTF-8"?>
<catalogo>
    <libro id="L001">
        <titulo>El Arte de Programar con StAX</titulo>
        <autor>Ana García</autor>
        <precio moneda="EUR">29.95</precio>
    </libro>
    <libro id="L002">
        <titulo>Principios de XML Moderno</titulo>
        <autor>Benito López</autor>
        <precio moneda="USD">35.00</precio>
    </libro>
    <libro id="L003">
        <titulo>Seguridad en Parsers XML</titulo>
        <autor>Carla Diaz</autor>
        <precio moneda="EUR">45.50</precio>
    </libro>
</catalogo>
```

Lectura del anterior xml utilizando StAX -Modelo Cursor

```
public static void leerCatalogo(XMLStreamReader reader) {
    if (reader == null) return;
    // Variable para guardar el nombre de la etiqueta que se acaba de abrir
    String etiquetaActual = "";
    System.out.println("---- INICIO DE PROCESAMIENTO ----");

    try {
        // Bucle principal: El corazón del modelo PULL
        while (reader.hasNext()) {
            int eventType = reader.next();
            switch (eventType) {
                case XMLStreamConstants.START_ELEMENT -> {
                    // 1. Guardamos el nombre de la nueva etiqueta
                    etiquetaActual = reader.getLocalName();
                    // 2. Lógica específica al encontrar el inicio de un LIBRO
                    if ("libro".equals(etiquetaActual)) {
                        // Extraemos el ID como atributo
                        String id = reader.getAttributeValue(null, "id");
                        System.out.println("\nLIBRO ENCONTRADO [ID: " + id + "]");
                    }
                }
                case XMLStreamConstants.CHARACTERS -> {
                    // 3. Procesamos el texto del nodo
                    String texto = reader.getText().trim();
                    if (!texto.isEmpty()) {
                        // Usamos etiquetaActual para saber qué tipo de dato estamos leyendo
                        System.out.println(" -> " + etiquetaActual.toUpperCase() + ":" + texto);
                    }
                }
                case XMLStreamConstants.END_ELEMENT -> {
                    // 4. Lógica de cierre
                    if ("libro".equals(reader.getLocalName())) {
                        System.out.println("-----");
                    }
                    // Limpiar etiquetaActual
                    // etiquetaActual = "";
                }
                case XMLStreamConstants.START_DOCUMENT ->
                    // Casos de una sola linea pueden ir sin llaves
                    System.out.println("Documento listo para ser leido.");
            }
        }
    } catch (Exception e) {
        System.err.println("Error durante la lectura StAX: " +
                           e.getMessage());
    }
}
```

Salida:

```
--- INICIO DE PROCESAMIENTO ---

LIBRO ENCONTRADO [ID: L001]
-> TITULO: El Arte de Programar con StAX
-> AUTOR: Ana García
-> PRECIO: 29.95
-----

LIBRO ENCONTRADO [ID: L002]
-> TITULO: Principios de XML Moderno
-> AUTOR: Benito López
-> PRECIO: 35.00
-----

LIBRO ENCONTRADO [ID: L003]
-> TITULO: Seguridad en Parsers XML
-> AUTOR: Carla Diaz
-> PRECIO: 45.50
-----

--- FIN DE PROCESAMIENTO ---
```

3.3 Creación de un XMLStreamWriter.

El **Modelo Cursor** para la escritura es una **operación push**, donde el programador llama a los métodos en el orden exacto en que deben aparecer los *tokens* en el XML de salida.

Método	Descripción con ejemplo
createXMLStreamWriter(OutputStream stream)	Crea el escritor que genera XML en un flujo de bytes, usando la codificación por defecto del sistema. Ejemplo: <code>XMLStreamWriter escritor = factory.createXMLStreamWriter(new FileOutputStream("salida.xml"));</code>
createXMLStreamWriter(OutputStream stream, String encoding)	Crea el escritor que genera XML en un flujo de bytes, especificando la codificación (por ejemplo "UTF-8"). Ejemplo: <code>XMLStreamWriter escritor = factory.createXMLStreamWriter(new FileOutputStream("salida.xml"), "UTF-8");</code>
createXMLStreamWriter(Writer writer)	Crea el escritor que genera XML directamente en un flujo de caracteres (Writer), útil si ya se ha gestionado la codificación. Ejemplo: <code>XMLStreamWriter escritor = factory.createXMLStreamWriter(new FileWriter("salida.xml"));</code>
createXMLStreamWriter(Result result)	Crea el escritor que genera XML en una fuente abstracta (javax.xml.transform.Result), útil en transformaciones con JAXP. Ejemplo: <code>XMLStreamWriter escritor = factory.createXMLStreamWriter(new StreamResult(new File("salida.xml")));</code>

```
// 1. OBTENER la instancia de la fábrica
XMLOutputFactory factory = XMLOutputFactory.newInstance();

XMLStreamWriter writer = null;
// Asegura que el escritor cree automáticamente los prefijos de namespaces
// si se necesitan.
factory.setProperty(XMLOutputFactory.IS_REPAIRING_NAMESPACES, Boolean.TRUE);

// 2. CREAR el XMLStreamWriter
// Se asocia a un FileOutputStream para escribir en un archivo físico
// y se especifica la codificación "UTF-8".
writer = factory.createXMLStreamWriter(new FileOutputStream(rutaArchivo),
                                         "UTF-8" );
```

3.4 Escritura con XMLStreamWriter.

Propósito	Método	Descripción
Documento	<code>writeStartDocument(String encoding, String version)</code>	Escribe la declaración inicial (Ej: <?xml version="1.0" encoding="UTF-8"?>).
Elementos	<code>writeStartElement(String localName)</code>	Escribe la etiqueta de apertura (Ej: <nombre>).
	<code>writeEndElement()</code>	Escribe la etiqueta de cierre (Ej: </nombre>).
Contenido	<code>writeCharacters(String text)</code>	Escribe el contenido de texto. Maneja automáticamente el escape de caracteres especiales de XML Ej: < se convierte en <, & por &), previniendo errores de formato.
Atributos	<code>writeAttribute(String localName, String value)</code>	Escribe un atributo. Debe ser llamado <i>inmediatamente</i> después de writeStartElement().
Finalización	<code>flush() / close()</code>	<code>flush()</code> fuerza a escribir los datos del <i>buffer</i> . <code>close()</code> libera recursos.

Control de indentación en XMLStreamWriter (modelo cursor)

El modelo cursor de StAX (XMLStreamWriter) **no aplica indentación automática**. Por defecto, escribe el XML como una secuencia continua de etiquetas sin saltos de línea ni espacios.

Por tanto, el **control de la indentación también es manual**: el programador debe insertar explícitamente saltos de línea (`writeCharacters("\n")`) y espacios (`writeCharacters(" ")`) para simular la estructura jerárquica del documento.

En el modelo cursor, **la indentación se escribe directamente**, sin poder almacenarla en una variable:

```
writer.writeCharacters("\n");           // salto de línea
writer.writeCharacters("    ");          // 1 nivel de indentación
writer.writeStartElement("libro");       // etiqueta
```

Plantilla de Uso del XMLStreamWriter

Esta es la estructura fundamental que se sigue en Java para generar cualquier archivo XML usando el Modelo Cursor de StAX.

```
public void generarXML(String rutaArchivo) {
    XMLOutputFactory factory = XMLOutputFactory.newInstance();
    XMLStreamWriter writer = null;
    try {
        // 1. Obtención del Writer (ver sección 1.2)
        writer = factory.createXMLStreamWriter(new FileOutputStream(rutaArchivo),
                                                "UTF-8");
        // --- INICIO DE LA SECUENCIA DE ESCRITURA ---
        // 2. Declaración XML
            writer.writeStartDocument("UTF-8", "1.0");
            writer.writeCharacters("\n"); // Opcional: para añadir un salto de
                                         // linea por formato
        // 3. Elemento Raíz
            writer.writeStartElement("elemento_raiz"); // <elemento_raiz>
            writer.writeAttribute("version", "1.0"); // Añadir atributo a la raíz

        // 4. Escribir Contenido
        // Se puede llamar a métodos auxiliares aquí, o escribir directamente:
            writer.writeCharacters("\n ");
            writer.writeStartElement("hijo"); // <hijo>
            writer.writeCharacters("Contenido de texto"); // Contenido de texto
            writer.writeEndElement(); // </hijo>

        // 5. Cierre de la Raíz
            writer.writeCharacters("\n");
            writer.writeEndElement(); // </elemento_raiz>

        // 6. Cierre del Documento (Garantiza que todas las etiquetas sean cerradas)
            writer.writeEndDocument();
        // --- FIN DE LA SECUENCIA DE ESCRITURA ---
    } catch (Exception e) {
        System.err.println("Error durante la escritura: " + e.getMessage());
    } finally {
```

```
// 7. Limpieza y Cierre (Siempre en un bloque finally)
try {
    if (writer != null) {
        writer.flush();
        writer.close();
    }
} catch (Exception e) { }
```

3.5 Importancia de Cerrar los Streams y Parsers

Tanto el **XMLStreamReader** como el **XMLStreamWriter** manipulan recursos del sistema operativo (archivos de disco, conexiones de red) y utilizan memoria temporal. Por lo tanto, **deben cerrarse siempre de forma explícita**.

La especificación histórica de StAX no exigía que XMLStreamReader/XMLStreamWriter implementaran AutoCloseable, por lo que usar try-with-resources directamente sobre ellos no es 100% portable. Varias implementaciones modernas lo han hecho, pero en código que debe correr en entornos diversos conviene aplicar patrones defensivos.

```
XMLOutputFactory of = XMLOutputFactory.newInstance();
try (OutputStream os = Files.newOutputStream(path);
     BufferedOutputStream bos = new BufferedOutputStream(os)) {
    XMLStreamWriter writer = null;
    try {
        writer = of.createXMLStreamWriter(bos, "UTF-8");
        // escribir...
        writer.flush();
    } finally {
        if (writer != null) {
            try { writer.close(); } catch (XMLStreamException ignored) {}
        }
    }
}
```

4. APIs para trabajar con StAX-Modelo evento

El **Modelo por Eventos** (Event API) es la segunda forma de procesar documentos XML mediante StAX. A diferencia del **Modelo Cursor**, que devuelve un valor entero (int) para identificar el tipo de *token* (como START_ELEMENT = 1), este modelo **trabaja con objetos que representan eventos XML** (inicio de elemento, texto, fin de elemento, etc.).

- **En el Modelo Cursor:** Se devuelve un **código entero**, y luego se llama a reader.getLocalName() o reader.getText().
- **En el Modelo de Eventos:** El parser **devuelve un objeto** que implementa la interfaz XMLEvent. Este objeto ya está tipificado, por ejemplo, como StartElement o Characters.

Proporciona una **programación más orientada a objetos** y un **código más legible**, aunque con un pequeño coste adicional en rendimiento.

Qué cambia respecto al modelo cursor

Modelo Cursor (XMLStreamReader)	Modelo por Eventos (XMLEventReader)
Trabaja con códigos enteros (<code>XMLStreamConstants.START_ELEMENT</code> , etc.)	Cada token es un objeto XMLEvent con sus propios métodos.
Bajo nivel, más rápido.	Orientado a objetos, más legible.
Se avanza con <code>reader.next()</code> .	Se avanza con <code>eventReader.nextEvent()</code> .
Se accede con métodos como <code>getLocalName()</code> , <code>getText()</code> .	Se usa <code>event.asStartElement()</code> , <code>event.asCharacters()</code> , etc.

Componentes Centrales del Modelo Eventos

Las clases principales en este modelo son:

1. **XMLEventReader**: Interfaz utilizada para la lectura. En lugar de avanzar a un *token* (como next()), avanza a un objeto **XMLEvent** (usando nextEvent()).
2. **XMLEventWriter**: Interfaz utilizada para la escritura. Recibe objetos XMLEvent y los escribe en la salida.
3. **XMLEventFactory**: Es una nueva fábrica que se introduce en este modelo. Su función es crear fácilmente los objetos de evento necesarios para la escritura (Ej: crear un objeto **StartElement** o **Characters**).

4.1 Creación de un XMLEventReader

Método	Descripción con ejemplo de uso
createXMLEventReader(InputStream stream)	Crea el lector desde un flujo de bytes, usando la codificación detectada automáticamente o por defecto. Ejemplo: XMLEventReader lector = factory.createXMLEventReader(new FileInputStream("archivo.xml"));
createXMLEventReader(InputStream stream, String encoding)	Crea el lector desde un flujo de bytes, especificando la codificación (por ejemplo "UTF-8"). Ejemplo: XMLEventReader lector = factory.createXMLEventReader(new FileInputStream("archivo.xml"), "UTF-8");
createXMLEventReader(Reader reader)	Crea el lector desde un flujo de caracteres, útil si ya se ha gestionado la codificación. Ejemplo: XMLEventReader lector = factory.createXMLEventReader(new FileReader("archivo.xml"));
createXMLEventReader(Source source)	Crea el lector desde una fuente abstracta (StreamSource, DOMSource, etc.), útil en transformaciones. Ejemplo: XMLEventReader lector = factory.createXMLEventReader(new StreamSource(new File("archivo.xml")));
createXMLEventReader(String systemId, InputStream stream)	Crea el lector desde un flujo de bytes, asociando una URI como identificador del sistema. Ejemplo: XMLEventReader lector = factory.createXMLEventReader("http://ejemplo.com/catalogo", new FileInputStream("archivo.xml"));
createXMLEventReader(String systemId, Reader reader)	Crea el lector desde un flujo de caracteres, asociando una URI como identificador del sistema. Ejemplo: XMLEventReader lector = factory.createXMLEventReader("http://ejemplo.com/catalogo", new FileReader("archivo.xml"));

Ejemplo

```
public class ConfiguracionStAXEventos {
    public XMLEventReader configurarYCrearEventReader(String rutaArchivo) {
        XMLInputFactory factory = XMLInputFactory.newInstance();
        try {
            // 1. CONFIGURAR PROPIEDADES CLAVE
            factory.setProperty(XMLInputFactory.IS_VALIDATING, Boolean.TRUE);
            factory.setProperty(XMLConstants.FEATURE_SECURE_PROCESSING, Boolean.TRUE);
            // 2. CREAR el lector por eventos
            XMLEventReader lector = factory.createXMLEventReader(new FileReader(rutaArchivo));
            return lector;
        } catch (FileNotFoundException e) {
            System.err.println("ERROR: Archivo XML no encontrado en la ruta: " + rutaArchivo);
        } catch (IllegalArgumentException e) {
            System.err.println("ERROR: La implementación del Parser no soporta alguna propiedad configurada.");
        } catch (XMLStreamException e) {
            System.err.println("ERROR DE PARSEO/VALIDACIÓN en XML: " + e.getMessage());
        }
    }
}
```

4.2 Lectura con XMLEventReader

El **XMLEventReader** implementa el modelo "**Pull por eventos**", donde el programador controla el flujo llamando a `nextEvent()` para obtener el siguiente objeto **XMLEvent**.

Cada evento encapsula información sobre una estructura del documento XML (inicio de etiqueta, texto, cierre, etc.).

Este modelo es más legible y modular que el cursor, ya que cada token se representa como un objeto con métodos específicos según su tipo (`StartElement`, `Characters`, `EndElement`, etc.).

Métodos para la lectura con XMLEventReader

Propósito	Método	Tipo de retorno	Clase asociada
Control de flujo	<code>hasNext()</code>	boolean	Verifica si hay más eventos por leer
Obtener evento	<code>nextEvent()</code>	<code>XMLEvent</code>	Devuelve el siguiente evento del documento
Identificación	<code>event.isStartElement()</code>	boolean	¿Es un evento de apertura de etiqueta?
	<code>event.isEndElement()</code>	boolean	¿Es un evento de cierre de etiqueta?
	<code>event.isCharacters()</code>	boolean	¿Es un evento de texto?
Información del elemento	<code>startElement.getName().getLocalPart()</code>	String	Nombre local de la etiqueta
	<code>characters.getData()</code>	String	Contenido de texto
Atributos	<code>startElement.getAttributeByName(Qname)</code>	Attribute	Obtiene el atributo por nombre

Ejemplo de uso en código (modelo evento)

```

while (lector.hasNext()) {
    // Se obtiene el siguiente evento del flujo XML
    XMLEvent event = lector.nextEvent();

    // Si el evento representa el inicio de una etiqueta (START_ELEMENT)
    if (event.isStartElement()) {
        StartElement start = event.asStartElement(); // Se convierte el evento a StartElement
        String nombre = start.getName().getLocalPart(); // Se obtiene el nombre local de la /
                                                       //etiqueta
        System.out.println("-> Apertura de: " + nombre); // Se muestra por consola
    }

    // Si el evento representa contenido de texto (CHARACTERS)
    if (event.isCharacters()) {
        Characters texto = event.asCharacters(); // Se convierte el evento a Characters
        String contenido = texto.getData().trim(); // Se obtiene el texto y se eliminan
                                                    //espacios en blanco
        if (!contenido.isEmpty()) { // Se ignoran textos vacíos o solo con espacios
            System.out.println("-> Contenido: " + contenido); // Se muestra el contenido
                                                    //textual
        }
    }

    // Si el evento representa el cierre de una etiqueta (END_ELEMENT)
    if (event.isEndElement()) {
        EndElement end = event.asEndElement(); // Se convierte el evento a EndElement
        System.out.println("Cierre de: " + end.getName().getLocalPart()); // Se muestra el
                                                               //nombre de la etiqueta cerrada
    }
}

```

4.3 ¿Qué es QName y por qué se usa en StAX?

QName (Qualified Name) es una clase que representa el **nombre completo de una etiqueta o atributo XML**, incluyendo:

- **Nombre local** → "id", "libro", "precio"
- **Espacio de nombres (namespace URI)** → "http://ejemplo.com/catalogo"
- **Prefijo** → "ns" (si existe en el XML)

Se utiliza en StAX para acceder correctamente a elementos y atributos cuando el XML utiliza **espacios de nombres**.

¿Dónde se usa QName?

■ En el modelo evento (XMLEventReader):

- ✓ Para obtener el nombre completo de una etiqueta:

```
QName nombre = startElement.getName();
nombre.getLocalPart();           // "libro"
nombre.getNamespaceURI();        // "http://ejemplo.com/catalogo"
nombre.getPrefix();              // "ns"
```

- ✓ Para acceder a un atributo por nombre:

```
Attribute attr = startElement.getAttributeByName(new QName("id"));
```

- ✓ Si el atributo tiene espacio de nombres:

```
Attribute attr = startElement.getAttributeByName
    (new QName("http://ejemplo.com/catalogo", "id"));
```

■ En el modelo cursor (XMLStreamReader):

- ✓ Para obtener el nombre completo del elemento actual:

```
QName nombre = reader.getName();
```

- ✓ Para obtener el nombre de un atributo en una posición:

```
QName nombreAttr = reader.getAttributeName(i);
```

■ Ejemplo XML con namespace

```
<ns:libro xmlns:ns="http://ejemplo.com/catalogo" ns:id="L001">
    <ns:title>Ejemplo</ns:title>
</ns:libro>
```

```
QName clave = new QName("http://ejemplo.com/catalogo", "id");
Attribute attr = startElement.getAttributeByName(clave);
String valor = attr.getValue(); // "L001"
```

Ejemplo completo: lectura XML con XMLEventReader

```
public static void leerCatalogoEventos(XMLEventReader lector) {
    // Si el lector es nulo, se aborta el procesamiento
    if (lector == null) return;
    // Variable para guardar el nombre de la última etiqueta abierta
    String etiquetaActual = "";
    System.out.println("--- INICIO DE PROCESAMIENTO ---");
    try {
        // Bucle principal: mientras haya eventos por leer
        while (lector.hasNext()) {
            // Se obtiene el siguiente evento del flujo XML
            XMLEvent event = lector.nextEvent();
            // Si el evento es una apertura de etiqueta (START_ELEMENT)
            if (event.isStartElement()) {
                StartElement start = event.asStartElement();

                // Se guarda el nombre local de la etiqueta abierta
                etiquetaActual = start.getName().getLocalPart();
```

```

// Si la etiqueta es <libro>, se intenta extraer el atributo "id"
if ("libro".equals(etiquetaActual)) {
    // Se accede al atributo "id" usando QName (sin namespace)
    Attribute idAttr = start.getAttributeByName(new QName("id"));

    // Si el atributo existe, se imprime su valor
    if (idAttr != null) {
        String id = idAttr.getValue();
        System.out.println("\nLIBRO ENCONTRADO [ID: " + id + "]");

    }
}

// Si el evento contiene texto (CHARACTERS)
if (event.isCharacters()) {
    // Se obtiene el contenido textual y se elimina el espacio sobrante
    String texto = event.asCharacters().getData().trim();

    // Si el texto no está vacío, se imprime junto al nombre de la etiqueta actual
    if (!texto.isEmpty()) {
        System.out.println(" -> " + etiquetaActual.toUpperCase() + ":" + texto);
    }
}
// Si el evento es cierre de etiqueta (END_ELEMENT)
if (event.isEndElement()) {
    EndElement end = event.asEndElement();

    // Si se cierra un <libro>, se imprime línea de separación
    if ("libro".equals(end.getName().getLocalPart())) {
        System.out.println("-----");
    }
}

// Se cierra el lector al finalizar
lector.close();

} catch (Exception e) {
    // Captura de errores durante el procesamiento del XML
    System.err.println("Error durante la lectura StAX (evento): " + e.getMessage());
}
}
}

```

Salida:

```

--- INICIO DE PROCESAMIENTO ---

LIBRO ENCONTRADO [ID: L001]
-> TITULO: El Arte de Programar con St
-> AUTOR: Ana García
-> PRECIO: 29.95
-----

LIBRO ENCONTRADO [ID: L002]
-> TITULO: Principios de XML Moderno
-> AUTOR: Benito López
-> PRECIO: 35.00
-----

LIBRO ENCONTRADO [ID: L003]
-> TITULO: Seguridad en Parsers XML
-> AUTOR: Carla Díaz
-> PRECIO: 45.50
-----
```

4.4 Creación de un XMLEventWriter

El XMLEventWriter implementa el modelo "**Pull por eventos**", donde el **programador genera objetos XMLEvent** (como StartElement, Characters, EndElement) y los envía secuencialmente al escritor. Este modelo es más modular y expresivo que el cursor, ideal para **construir documentos XML bien formados**.

Método	Descripción con ejemplo
createXMLEventWriter(OutputStream stream)	Crea el escritor que genera XML en un flujo de bytes, usando la codificación por defecto del sistema. Ejemplo: XMLEventWriter escritor = factory.createXMLEventWriter(new FileOutputStream("salida.xml"));
createXMLEventWriter(OutputStream stream, String encoding)	Crea el escritor en un flujo de bytes, especificando la codificación (por ejemplo "UTF-8"). Ejemplo: XMLEventWriter escritor = factory.createXMLEventWriter(new FileOutputStream("salida.xml"), "UTF-8");
createXMLEventWriter(Writer writer)	Crea el escritor en un flujo de caracteres (Writer), útil si ya se ha gestionado la codificación. Ejemplo: XMLEventWriter escritor = factory.createXMLEventWriter(new FileWriter("salida.xml"));
createXMLEventWriter(Result result)	Crea el escritor en una fuente abstracta (StreamResult, DOMResult, etc.), útil en transformaciones con JAXP. Ejemplo: XMLEventWriter escritor = factory.createXMLEventWriter(new StreamResult(new File("salida.xml")));

4.5 Escritura con XMLEventWriter

Los **eventos XML** (etiquetas, atributos, texto, CDATA, DTD, etc.) se crean mediante la clase **XMLEventFactory**.

El XMLEventWriter **no genera eventos, sino que los recibe mediante el método add()** y los escribe en el documento.

Esta separación permite modularidad: primero se construyen los eventos, luego se escriben.

Qué clase proporciona cada método?

Tipo de operación	Clase responsable	Ejemplos de métodos
Creación de eventos XML	XMLEventFactory	createStartElement(...), createEndElement(...), createCharacters(...), createAttribute(...), createNamespace(...), createCData(...), createDTD(...), createStartDocument(...), createEndDocument()
Escritura de eventos en el documento	XMLEventWriter	add(XMLEvent evento), flush(), close()

Métodos para crear documentos

Categoría	Método	Propósito con ejemplo
Inicio de documento	eventFactory.createStartDocument(String encoding, String version)	Crea la declaración XML inicial. Ejemplo: <?xml version="1.0" encoding="UTF-8"?>. Se usa como primera línea del documento.
Declaración DTD	eventFactory.createDTD(String dtdData)	Crea una declaración de tipo de documento. Ejemplo: <!DOCTYPE catalogo SYSTEM "catalogo.dtd">. Se usa para vincular un DTD externo.
Namespace	eventFactory.createNamespace(String prefix, String uri)	Crea una declaración de espacio de nombres. Ejemplo: xmlns:ns="http://ejemplo.com". Útil para documentos con prefijos.
Apertura de etiqueta	eventFactory.createStartElement(String prefix, String uri, String localName)	Crea una etiqueta de apertura. Ejemplo: <libro>. También puede incluir atributos y namespaces.
Atributo	eventFactory.createAttribute(String nombre, String valor)	Crea un atributo para una etiqueta. Ejemplo: id="L001". Se añade al crear el StartElement.
Contenido textual	eventFactory.createCharacters(String texto)	Crea contenido de texto dentro de una etiqueta. Ejemplo: "Ana García".

Contenido CDATA	eventFactory.createCDATA(String texto)	Crea una sección CDATA para texto con caracteres especiales. Ejemplo: <![CDATA[Texto con < y &]]>.
Cierre de etiqueta	eventFactory.createEndElement(String prefix, String uri, String localName)	Crea una etiqueta de cierre. Ejemplo: </libro>.
Fin de documento	eventFactory.createEndDocument()	Crea el evento de cierre del documento XML. Se recomienda añadirlo antes de writer.close().
Escritura	writer.add(XMLEvent evento)	Añade un evento al documento XML. Se usa para escribir cada parte del documento.
Finalización	writer.flush() / writer.close()	Escribe y cierra el documento. flush() asegura que todo se ha escrito; close() libera recursos.

Control de indentación en XMLEventWriter

El modelo StAX no aplica indentación automática al generar XML. La clase XMLEventWriter escribe los eventos tal como se le envían, sin formato adicional.

Por tanto, **el control de la indentación es manual**: el programador debe insertar eventos de texto (Characters) que representen saltos de línea ("\\n") y espacios (" "), simulando la estructura visual del documento.

Puedes usar la clase XMLEventFactory para crear eventos de texto que simulen la indentación:

```
XMLEvent saltoLinea = eventFactory.createCharacters("\\n");
XMLEvent tab1 = eventFactory.createCharacters("    "); // 4 espacios
XMLEvent tab2 = eventFactory.createCharacters("        "); // 8 espacios
```

Y añadirlos estratégicamente antes de cada etiqueta o contenido:

```
writer.add(saltoLinea);
writer.add(tab1);
writer.add(eventFactory.createStartElement("", "", "libro"));
```

Ejemplo de uso en código (modelo evento)

```
// 1. Instanciación de escritor y fábrica de eventos
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
XMLEventWriter writer = outputFactory.createXMLEventWriter(
(new FileOutputStream("archivo.xml"), "UTF-8"));
XMLEventFactory eventFactory = XMLEventFactory.newInstance();

// 2. Eventos de indentación
XMLEvent saltoLinea = eventFactory.createCharacters("\\n");
XMLEvent tab1 = eventFactory.createCharacters("    ");
XMLEvent tab2 = eventFactory.createCharacters("        ");

// 3. Inicio del documento
writer.add(eventFactory.createStartDocument("UTF-8", "1.0"));
writer.add(saltoLinea);

// 4. Elemento raíz
writer.add(eventFactory.createStartElement("", "", "raiz"));
writer.add(saltoLinea);

// 5. Elemento hijo indentado
writer.add(tab1);
writer.add(eventFactory.createStartElement("", "", "elemento"));
writer.add(eventFactory.createCharacters("contenido"));
writer.add(eventFactory.createEndElement("", "", "elemento"));
writer.add(saltoLinea);

// 6. Cierre del elemento raíz
writer.add(eventFactory.createEndElement("", "", "raiz"));
writer.add(saltoLinea);

// 7. Fin del documento
writer.add(eventFactory.createEndDocument());
writer.close();
```

Ejemplo completo: generar XML desde un fichero de texto (libros.txt)**Contenido del archivo libros.txt**

```
L001;El Arte de Programar;Ana García;29.95;EUR
L002;Java para FP;Luis Pérez;24.50;EUR
L003;XML y StAX;Marta López;19.99;USD
```

Código Java para generar catalogo.xml

java

```
public static void escribirCatalogoDesdeArchivo(String rutaEntrada, String rutaSalida) {
    try (BufferedReader lector = new BufferedReader(new FileReader(rutaEntrada))) {
        XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
        XMLEventWriter writer = outputFactory.createXMLEventWriter(new
FileOutputStream(rutaSalida), "UTF-8");
        XMLEventFactory eventFactory = XMLEventFactory.newInstance();

        XMLEvent saltoLinea = eventFactory.createCharacters("\n");
        XMLEvent tab1 = eventFactory.createCharacters("    ");
        XMLEvent tab2 = eventFactory.createCharacters("        ");

        writer.add(eventFactory.createStartDocument("UTF-8", "1.0"));
        writer.add(saltoLinea);

        writer.add(eventFactory.createStartElement("", "", "catalogo"));
        writer.add(saltoLinea);

        String linea;
        while ((linea = lector.readLine()) != null) {
            String[] partes = linea.split(";");
            if (partes.length != 5) continue;

            String id = partes[0], titulo = partes[1], autor = partes[2],
            precio = partes[3], moneda = partes[4];

            writer.add(tab1);
            Attribute idAttr = eventFactory.createAttribute("id", id);
            writer.add(eventFactory.createStartElement("", "", "libro"
            , java.util.Collections.singletonList(idAttr).iterator(), null));
            writer.add(saltoLinea);

            writer.add(tab2);
            writer.add(eventFactory.createStartElement("", "", "titulo"));
            writer.add(eventFactory.createCharacters(titulo));
            writer.add(eventFactory.createEndElement("", "", "titulo"));
            writer.add(saltoLinea);

            writer.add(tab2);
            writer.add(eventFactory.createStartElement("", "", "autor"));
            writer.add(eventFactory.createCharacters(autor));
            writer.add(eventFactory.createEndElement("", "", "autor"));
            writer.add(saltoLinea);

            writer.add(tab2);
            Attribute monedaAttr = eventFactory.createAttribute("moneda", moneda);
            writer.add(eventFactory.createStartElement("", "", "precio",
            java.util.Collections.singletonList(monedaAttr).iterator(), null));
            writer.add(eventFactory.createCharacters(precio));
            writer.add(eventFactory.createEndElement("", "", "precio"));
            writer.add(saltoLinea);

            writer.add(tab1);
            writer.add(eventFactory.createEndElement("", "", "libro"));
            writer.add(saltoLinea);
        }

        writer.add(eventFactory.createEndElement("", "", "catalogo"));
        writer.add(saltoLinea);

        writer.add(eventFactory.createEndDocument());
        writer.close();

        System.out.println("Catálogo generado correctamente.");
    } catch (Exception e) {
        System.err.println("Error al generar catálogo: " + e.getMessage());
    }
}
```

catalogo.xml generado

```
<?xml version="1.0" encoding="UTF-8"?>
<catalogo>
    <libro id="L001">
        <titulo>El Arte de Programar</titulo>
        <autor>Ana García</autor>
        <precio moneda="EUR">29.95</precio>
    </libro>
    <libro id="L002">
        <titulo>Java para FP</titulo>
        <autor>Luis Pérez</autor>
        <precio moneda="EUR">24.50</precio>
    </libro>
    <libro id="L003">
        <titulo>XML y StAX</titulo>
        <autor>Marta López</autor>
        <precio moneda="USD">19.99</precio>
    </libro>
```

5. Diferencias entre XMLStreamReader y XMLEventReader

Ambos lectores (XMLStreamReader y XMLEventReader) usan las mismas constantes de tipo de evento (**XMLStreamConstants.START_ELEMENT**, **CHARACTERS**, etc.), la diferencia clave está en el modelo de recorrido y en cómo se accede a la información. Vamos a compararlos:

Característica	XMLStreamReader (Cursor)	XMLEventReader (Evento)
Modelo de recorrido	Basado en cursor: se mueve sobre el flujo	Basado en eventos: devuelve objetos XMLEvent
Tipo de acceso	Métodos directos (getLocalName(), getText())	Métodos sobre eventos (event.asStartElement())
Estado interno	El lector mantiene el estado	Cada evento es independiente
Lectura de texto	reader.getText()	event.asCharacters().getData()
Lectura de atributos	reader.getAttributeValue(...)	startElement.getAttributeByName(...)
Flujo de control	Avanza con reader.next()	Avanza con reader.nextEvent()
Utilidades	Más bajo nivel, más eficiente	Más expresivo, más orientado a objetos

¿Por qué ambos usan las constantes de XMLStreamConstants?

Porque esa constante representa el **tipo de evento XML**, y es común a ambos modelos. Internamente, el parser XML define los tipos de evento como enteros:

```
public interface XMLStreamConstants {
    int START_ELEMENT = 1;
    int END_ELEMENT = 2;
    int CHARACTERS = 4;
    ...
}
```

Tanto el cursor como el evento devuelven ese tipo con:

- **reader.getEventType()** → en cursor
- **event.getEventType()** → en evento

Pero cómo accedes a los datos depende del modelo.

Aunque XMLStreamReader y XMLEventReader usan las mismas constantes para identificar tipos de evento, su modelo de recorrido es distinto. El cursor trabaja directamente sobre el flujo, mientras que el modelo evento encapsula cada paso como un objeto XMLEvent. El primero es más eficiente, el segundo más expresivo. Ambos son válidos en FP, según el tipo de parser que se quiera construir.