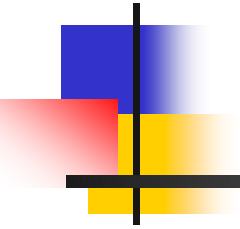




Tema 5: Lenguajes de Intercambio de Datos: XML y JSON

- 5.1: Introducción
- 5.2: El Lenguaje XML
- 5.3: Parsing de Documentos XML
- 5.4: El Lenguaje JSON
- 5.5: Parsing de Documentos JSON

5.1: Introducción





Introducción

- Queremos exponer la capa modelo creada en el Tema 3 como un servicio que pueda ser utilizado por aplicaciones remotas
- Las aplicaciones remotas
 - Pueden residir en otras máquinas o incluso acceder a la capa modelo a través de Internet
 - Pueden estar escritas en cualquier lenguaje de programación
 - Proporcionarán su propia interfaz gráfica (si la tienen) y pueden utilizar otras bases de datos o aplicaciones adicionales además de la nuestra
- Son necesarios lenguajes o formatos de datos robustos, que permitan expresar y transmitir información compleja entre sistemas heterogéneos
 - Ejemplos: XML, JSON, YAML



Ejemplo (1)

Ejemplo: Un portal externo desea ofrecer el servicio de búsqueda y visión de películas

- Con su propia capa de interfaz gráfica
- Integrándola con sus propios servicios
 - Añadiendo críticas de las películas escritas por sus usuarios
 - Cuando los usuarios del portal buscan películas, el portal invocará findMovies() en nuestro servicio web (que invocará a nuestra capa modelo)
 - Al mostrar los resultados, añadirá las críticas a cada película obtenidas de su base de datos
 - Ofreciendo descuentos a algunos usuarios
 - Para esos usuarios, el portal calculará el precio a mostrar al usuario restándole el descuento al precio devuelto
 - Al hacer el usuario la compra real, el portal usará su propia tarjeta de crédito y nos pagará el precio real
 - Desde el punto de vista de nuestra capa modelo, todas las compras desde el portal son de un único usuario
 - Etc.

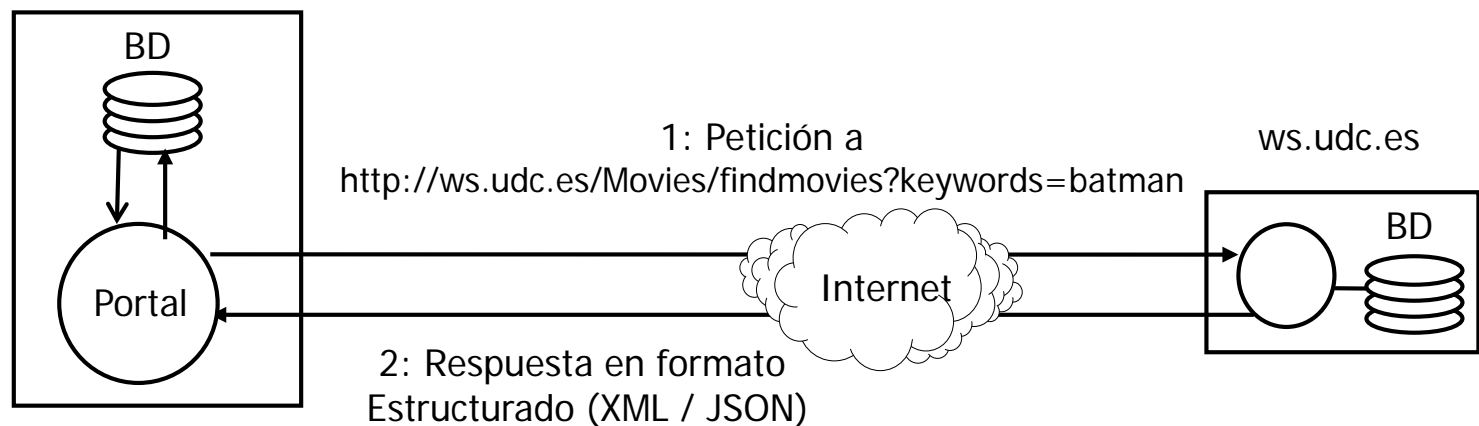


Ejemplo (2)

- En este tema nos centraremos en la búsqueda de películas
- Para ilustrar mejor algunos aspectos de XML y JSON supondremos que nuestro modelo da más información sobre cada película
 - Director/es de la película
 - Principales actores
 - Fecha de estreno
 - Género: uno o varios de entre una lista de valores posibles (comedia, romance, thriller,...)

Ejemplo (3)

- Una posible opción es que nuestro servicio exponga una URL a la que es posible pasarle como parámetros (método GET de HTTP) las keywords de búsqueda
- El portal remoto invocará dicha URL, nuestro servicio invocará a la capa modelo para obtener los datos y los devolverá al portal en un formato estructurado fácil de parsear desde cualquier lenguaje (e.g. XML o JSON)





Ejemplo (y 4)

- Recordatorio HTTP

- HTTP es un protocolo de nivel de aplicación que funciona sobre TCP, mediante el cual un cliente y un servidor pueden intercambiar fácilmente información textual
- HTTP no tiene nada que ver con HTML
 - Desde el punto de vista de HTTP, HTML, XML o JSON son simplemente texto que intercambian cliente y servidor
- HTTP soporta diversos métodos de invocación de URLs
 - El más popular es GET, utilizado para acceder a los datos del recurso referenciado.
 - Las peticiones GET soportan paso de parámetros en la URL con la sintaxis `?param1=val1¶m2=val2`



Campos de aplicación

- **Intercambio de datos entre aplicaciones heterogéneas**
 - Esta es la parte en la que nos centraremos en esta asignatura
- Generación de vistas (HTML, WML, PDF, etc.) a partir de documentos de datos
- Configuración de aplicaciones
- Bases de datos
- ...



5.2: El lenguaje XML



¿ Qué es XML ?

- XML (eXtensible Markup Language)
 - Lenguaje de tags (similar en sintaxis a HTML)
 - Estandarizado por el W3C (<http://www.w3.org>)
- Es extensible
 - XML no impone un conjunto de tags, sino sólo unas pocas normas sobre cómo usarlos
 - Los tags se abren (<tag>) y se cierran (</tag>) y en medio pueden tener otros tags anidados
 - Todos los documentos tienen un tag raíz
 - Los tags pueden tener atributos
 - Etc.
- Permite expresar información estructurada y fácilmente parseable
- Objetivo de este tema
 - **Entender los fundamentos básicos de XML necesarios para esta asignatura**



Información en XML (1)

- Las siguientes transparencias ilustran el uso de XML para expresar información de películas
 - Ese texto XML es el que devolvería la URL de nuestro servicio y que parsearía el portal del ejemplo de la introducción de este tema



Información en XML (2)

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<movies>
```

```
<!-- Dark Knight Rises Again. -->
```

```
<movie>
```

```
  <movieId>3</movieId>
```

```
  <title>Dark Knight Rises Again</title>
```

```
  <runtime>165</runtime>
```

```
  <releaseDate day="20" month="7" year="2012"/>
```

```
  <director>Cristopher Nolan</director>
```

```
  <actor>Christian Bale </actor>
```

```
  <actor>Morgan Freeman</actor>
```

```
  <genre>THR</genre>
```

```
  <price>4.99</price>
```

```
  <description> Ocho años después de los acontecimientos de The  
Dark Knight, Gotham se encuentra en un estado de paz. En virtud de  
los poderes otorgados por la Ley Dent, el comisario Gordon casi ha  
erradicado la violencia y el crimen organizado. Sin embargo, todavía  
se siente culpable por el encubrimiento de los crímenes de Harvey  
Dent.</description>
```

```
</movie>
```



Información en XML (y 3)

```
<!-- Con la Muerte en los Talones. -->
<movie>
  <movieId>4</movieId>
  <title>Con la Muerte en los Talones</title>
  <runtime>136</runtime>
  <releaseDate day="26" month="9" year="1959"/>
  <director>Alfred Hitchcock</director>
  <actor>Cary Grant</actor>
  <actor>Eve Marie Saint</actor>
  <actor>James Manson</actor>
  <genre>THR</genre>
  <genre>COM</genre>
  <price>3.99</price>
  <description> Roger O. Thornhill (Cary Grant) es un ejecutivo
publicitario de Nueva York al que unos espías confunden con un
agente del gobierno. Debe escapar, pero lo siguen de cerca. Durante
la fuga conoce a una atractiva mujer, Eve Kendall (Eva Marie Saint),
que lo ayuda. Algunas de sus escenas son recordadas por la magnitud
de las imágenes: la persecución de Thornhill por un campo sembrado
por una avioneta que quiere matarlo, una pelea en el Monte Rushmore
y la escena filmada de incógnito en la sede de la ONU en Nueva York.
</description>
</movie>
</movies>
```



Formato de un documento XML

- **Documento XML**

- Secuencia de caracteres (fichero, secuencia de caracteres que se envía por un socket, etc.) que tiene texto en formato XML

- **Aplicación XML**

- Conjunto particular de tags que permiten representar determinada información (e.g. la información sobre películas)

- Se dice que un documento XML está **bien formado** ("well-formed") si cumple con el conjunto de reglas que a continuación se expone

- Estas reglas permiten construir parsers eficientes

- Se distingue entre mayúsculas y minúsculas

- Comentarios

- Empiezan con `<!--` y terminan con `-->`
- Puede englobar varias líneas
- Dentro de un comentario no puede aparecer la secuencia `--`
- Comentarios tipo `<!--- Comentario ->` no son válidos



Declaración XML

- Declaración XML

- `<?xml version="1.0" encoding="UTF-8"?>`
- No es obligatoria (aunque sí aconsejable), pero si aparece tiene que aparecer justo al principio del documento

- **version**

- Indica la versión de la especificación XML (y no la versión del documento) a la que es conforme el documento

- **encoding**

- Indica la codificación de los caracteres del documento
- Cuando se utiliza XML para intercambio de información entre un cliente y un servidor, lo más normal es usar codificación UTF-8

Elementos y atributos (1)

- Todo documento debe tener un elemento raíz
 - **movies** en el ejemplo
- Entre el tag de inicio (**<tag>**) de un elemento y el de fin (**</tag>**) puede haber otros elementos o texto
 - NOTA: Por comodidad, utilizaremos las palabras "tag" y "elemento" de forma intercambiable
- No se puede mezclar el orden de los tags anidados
 - El primer elemento que se abre debe ser el último que se cierra
 - El siguiente ejemplo no estaría bien formado

```
<tag-1>
<tag-2>
</tag-1>
</tag-2>
```
- Un elemento puede tener atributos
 - El valor del atributo tiene que ir entrecomillado con comillas dobles (") o simples (')
 - Para un elemento dado, un atributo sólo puede tener un valor



Elementos y atributos (y 2)

- Elemento vacío

- No tiene elementos anidados ni texto

- Puede tener atributos

- Ejemplo:

- ```
<tag-1 attr-1="val1" attr-2="val2"></tag-1>
```

- Por comodidad, se puede usar la notación

- ```
<tag-1 attr-1="val1" attr-2="val2"/>
```



Elementos vs atributos (1)

- Podríamos haber pensando en múltiples alternativas para representar la misma información

```
<movie title="Dark Knight Rises Again" movieId="3" runtime="165"  
  releaseDay="20" releaseMonth="7" releaseYear="2012"  
  price="4.99">
```

```
  <director name="Cristopher Nolan"/>
```

```
  <actor name="Christian Bale"/>
```

```
  <actor name="Morgan Freeman"/>
```

```
  <genre name="THR"/>
```

```
  <description>Ocho años después de los acontecimientos de The  
Dark Knight, Gotham se encuentra en un estado de paz. En virtud de  
los poderes otorgados por la Ley Dent, el comisario Gordon casi ha  
erradicado la violencia y el crimen organizado. Sin embargo, todavía  
se siente culpable por el encubrimiento de los crímenes de Harvey  
Dent</description>
```

```
</movie>
```

Elementos vs atributos (y 2)

- En principio, se puede seguir la siguiente convención

- Usar elementos para datos multivaluados

```
<movie>  
  <actor>C. Bale</actor>  
  <actor>M. Freeman</actor>  
</movie>
```

```
<movie>  
  <actor name="C. Bale"/>  
  <actor name="M. Freeman"/>  
</movie>
```

- Usar contenido de elementos para datos de gran cantidad de texto
- Usar atributos o contenido de elementos en cualquier otro caso
- Ambas alternativas siguen esta convención
- **Se ha elegido la primera alternativa**
 - **Cualquiera de las alternativas (u otra distinta y correcta) es válida**



Referencias a entidades predefinidas

- Para poder incluir ciertos caracteres en el valor de un atributo o en el texto de un elemento, es preciso emplear "referencias a entidades"

Referencia	Carácter
<code>&lt;</code>	<code><</code>
<code>&amp;</code>	<code>&</code>
<code>&gt;</code>	<code>></code>
<code>&quot;</code>	<code>"</code>
<code>&apos;</code>	<code>'</code>

- Ejemplo

```
<tag char="&quot;">Si A &lt; B, entonces ... </tag>
```



Espacio de nombres (1)

- Un documento XML puede hacer uso de espacios de nombres
- Concepto similar al de paquete Java
- Permiten evitar conflictos de nombres cuando en un documento XML se usan elementos y atributos de distintas aplicaciones XML
- Cada espacio de nombres está asociado a una URI, que debe ser única
 - NOTA: Una URI es un identificador de un recurso, y típicamente es una URL
 - Se aconseja usar URLs (porque es una forma fácil de elegir nombres únicos)
 - No tienen por qué tener una existencia real (y de hecho, no suelen tenerla)



Espacio de nombres (2)

- Ejemplo. Supongamos que
 - Existe una aplicación XML que modela información de críticas de películas
 - Utiliza el espacio de nombres **`http://reviews.example.com`**
 - Dispone de varios elementos para modelar la información de la crítica incluyendo los elementos **`title`** y **`description`** que se refieren al título y resumen de la crítica
 - Nuestro servicio decide reutilizar estos elementos para incluir una crítica destacada junto con la información de la película
 - Así, si ya existe código que sepa generar/parsear los elementos de ese espacio de nombres, puede reusarse
 - El sitio Web especializado en cine decide que sus elementos pertenezcan al espacio de nombres **`http://ws.udc.es/movies/xml`** (buena práctica)

Espacio de nombres (3)

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<movies xmlns="http://ws.udc.es/movies/xml"
        xmlns:review="http://reviews.example.com">
```

```
  <!-- Dark Knight Rises Again -->
```

```
  <movie>
```

```
    <movieId>3</movieId>
```

```
    <title> Dark Knight Rises Again </title>
```

```
    <runtime>103</runtime>
```

```
    ...
```

```
    <description> Ocho años después ... </description>
```

```
    <review:title> Cualquiera puede ser un héroe </review:title>
```

```
    <review:description> Es difícil es hacer una crítica de una
    película tan compleja ... </review:description>
```

```
    <review:rating> 4 </review:rating>
```

```
    ...
```

```
  </movie>
```

```
  ...
```

```
</movies>
```

Espacio de nombres (y 4)

- `<movies xmlns="http://ws.udc.es/movies/xml" xmlns:review="http://reviews.example.com">`
- `xmlns="http://ws.udc.es/movies/xml"`
 - Espacio de nombres por defecto: todos los elementos contenidos dentro del elemento `movies` que no empiecen por `xxx:`, y el propio elemento `movies`, pertenecen al espacio de nombres `http://ws.udc.es/movies/xml`
- `xmlns:review="http://reviews.example.com"`
 - Todos los elementos y atributos contenidos dentro del elemento `movies` que empiezan por `review:` pertenecen al espacio de nombres `http://reviews.example.com`
 - `review` actúa sólo como un alias de la URI del espacio de nombres, es decir, el identificador del espacio de nombres no es `review`, sino la URI
- Conceptualmente se están importando dos "esquemas", uno para modelar información sobre películas y otro para modelar información sobre sus críticas



Validación de documentos XML (1)

- Existen varios tipos de mecanismos (esquemas) para expresar los elementos y atributos válidos de una aplicación XML y sus restricciones
 - Ejemplo: qué campos y de que tipo debe contener la información de una película
- Existen dos estándares del W3C para especificar esquemas XML
 - DTD (Document Type Definition)
 - Sencillo pero menos potente
 - Esquema XML (XML Schema)
 - Complejo
 - Semánticamente muy rico
 - Un esquema XML es a su vez un documento XML
- Un documento XML que cumple las restricciones de un esquema (DTD, esquema XML, etc.) se dice que es **válido**



Validación de documentos XML (2)

- Una aplicación que parsee un documento XML
 - Siempre comprueba que esté **bien formado**
 - **Puede** decidir comprobar si es válido. Veremos más adelante que validar tiene ventajas e inconvenientes
- En caso de decidir validarlo
 - El documento puede indicar (normalmente a través de URLs) cómo obtener los esquemas que utiliza
 - La aplicación también puede tener copias locales de los esquemas



Validación de documentos XML (y 3)

- A continuación, estudiamos los fundamentos básicos de los esquemas XML mediante la construcción de un esquema XML (**Movies.xsd**) para el ejemplo de las películas, asumiendo que
 - Se desea que los elementos pertenezcan al espacio de nombres **http://ws.udc.es/movies/xml**
 - Es buena práctica que cada esquema defina su propio espacio de nombres
 - NOTAS:
 - **.xsd** es la extensión que típicamente se utiliza para los esquemas XML (XSD = XML Schema Definition)
 - Buen tutorial de Esquemas XML: <http://www.xfront.com/xml-schema.html>



Movies.xsd (1)

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ws.udc.es/movies/xml"
  xmlns="http://ws.udc.es/movies/xml"
  elementFormDefault="qualified">
```

```
<!-- "ReleaseDate" type. -->
```

```
<xsd:complexType name="ReleaseDate">
```

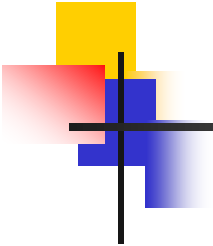
```
  <xsd:attribute name="day" type="xsd:short"/>
```

```
  <xsd:attribute name="month" type="xsd:short"/>
```

```
  <xsd:attribute name="year" type="xsd:short"/>
```

```
</xsd:complexType>
```

Movies.xsd (2)



```
<!-- "Genre" type.
```

```
Possible values of genre:
```

```
* COM: Comedy
```

```
* DRA: Drama
```

```
* HOR: Horror
```

```
* ROM: Romance
```

```
* SFI: Science fiction
```

```
* THR: Thriller
```

```
-->
```

```
<xsd:simpleType name="Genre">
```

```
  <xsd:restriction base="xsd:string">
```

```
    <xsd:enumeration value="COM"/>
```

```
    <xsd:enumeration value="DRA"/>
```

```
    <xsd:enumeration value="HOR"/>
```

```
    <xsd:enumeration value="ROM"/>
```

```
    <xsd:enumeration value="SFI"/>
```

```
    <xsd:enumeration value="THR"/>
```

```
  </xsd:restriction>
```

```
</xsd:simpleType>
```



Movies.xsd (3)

```
<!-- "Movie" type. -->
<xsd:complexType name="Movie">
  <xsd:sequence>
    <xsd:element name="movieId" type="xsd:long"/>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="runtime" type="xsd:short"/>
    <xsd:element name="releaseDate" type="ReleaseDate"/>
    <xsd:element name="director" type="xsd:string"
      minOccurs="1" maxOccurs="unbounded"/>
    <xsd:element name="actor" type="xsd:string"
      minOccurs="1" maxOccurs="unbounded"/>
    <xsd:element name="genre" type="Genre"
      minOccurs="1" maxOccurs="unbounded"/>
    <xsd:element name="price" type="xsd:float"/>
    <xsd:element name="description" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```



Movies.xsd (y 4)

```
<!-- "Movies" type. -->
<xsd:complexType name="Movies">
  <xsd:sequence>
    <xsd:element name="movie" type="Movie"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<!-- "movies" element (root element). -->
<xsd:element name="movies" type="Movies"/>

</xsd:schema>
```

Esquemas XML (1)

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ws.udc.es/movies/xml"
  xmlns="http://ws.udc.es/movies/xml"
  elementFormDefault="qualified">
```

- Los elementos y tipos (**schema**, **simpleType**, **short**, **string**, etc.) que se utilizan para definir un esquema XML están definidos en el esquema asociado al espacio de nombres **http://www.w3.org/2001/XMLSchema**
- **targetNamespace**: especifica la URI del espacio de nombres de los elementos definidos en este esquema
- **xmlns="http://ws.udc.es/movies/xml"**
 - Especifica el espacio de nombres por defecto para los elementos y tipos que no se prefijen (típicamente es el espacio de nombres especificado en **targetNamespace**)
- **elementFormDefault="qualified"**
 - En los documentos XML que utilicen este esquema, todos los elementos deberán escribirse cualificados (bien porque se ha utilizado el espacio de nombres por defecto o porque se les ha prefijado explícitamente)
 - En general, es buena práctica especificar siempre **qualified** (por defecto, el valor de **elementFormDefault** es **unqualified**)



Esquemas XML (2)

- Tipo simple (**simpleType**)
 - Tipos (de elementos o atributos) que tienen sólo valores y no otros atributos o elementos
 - Existen varios tipos predefinidos
 - **string**, **int**, **long**, **short**, **float**, **double**, **boolean**, **byte**, **dateTime**, etc.
- Restricciones de tipos simples
 - Permiten definir un tipo simple a partir de otro restringiendo los valores de este último
 - Las restricciones representan uno de los mecanismos disponibles para definir tipos derivados de otros tipos simples
 - El tipo **Genre** define un tipo **string** enumerado, es decir, sus valores posibles están restringidos a uno de los definidos en la enumeración



Esquemas XML (3)

- Restricciones de tipos simples (cont)
 - Existen muchas otras posibilidades de restricciones
 - Definir un tipo derivado de un **string** que restringe la longitud máxima de sus posibles valores
 - Definir un tipo derivado de un **string** que restringe sus valores mediante una expresión regular
- Tipo complejo (**complexType**)
 - Tipos (de elementos) que tienen atributos y/o elementos
 - El tipo **ReleaseDate** define un tipo de elemento que sólo contiene los atributos **day**, **month** y **year**, siendo los tres de tipo **short** (simple predefinido)



Esquemas XML (4)

Tipo complejo (**complexType**) [cont]

- El tipo **Movie** contiene sólo elementos
 - Usa **sequence** para declarar sus elementos
 - **sequence** es un "compositor" que define una secuencia ordenada de elementos
 - Existen otros compositores
 - **all**: los elementos pueden aparecer en cualquier orden
 - Tiene restricciones específicas (e.g. los elementos deben tener **maxOccurs="1"**)
 - **choice**: sólo puede aparecer uno de los elementos
 - Cada elemento se define dando
 - Su nombre
 - Su tipo
 - Opcionalmente, el número mínimo y máximo de ocurrencias posible
 - Por defecto => **minOccurs="1"** y **maxOccurs="1"**



Esquemas XML (y 5)

- Tipo complejo (**complexType**) [cont]
 - Es posible definir tipos complejos de atributos y elementos
 - La lista de atributos se especifica al final, es decir, después del compositor usado para definir los elementos
 - También es posible definir tipos complejos por derivación (se usan restricciones)
- El ejemplo termina definiendo el elemento raíz (**movies**)
- Existen herramientas online que permiten verificar si un documento XML es válido conforme a un esquema
 - <http://www.utilities-online.info/xsdvalidation>



Referencias a esquemas XML

- Si se desea, un documento XML puede incluir la URI del esquema XML asociado a cada espacio de nombres que disponga de esquema XML
- No veremos cómo hacerlo



5.3: Parsing de documentos XML



Introducción (1)

- Un documento XML se apoya en dos ideas
 - **[Obligatoria] Tiene que estar bien formado**, y en consecuencia, estar construido en base a las normas de XML (los tags se abren y cierran, los tags se anidan de manera jerárquica, los atributos tienen que ir entrecomillados, etc.)
 - **[Opcional]** El conjunto de elementos y atributos, y sus restricciones, pueden estar descritas formalmente en algún tipo de esquema (esquema XML, DTD, etc.) de manera que se pueda comprobar que el documento es **válido**
- Consecuencias
 - Es posible construir parsers genéricos que comprueban que el documento (1) está bien formado, y si se desea, (2) es válido
 - Existen parsers para los lenguajes más usuales
 - En un lenguaje orientado a objetos, un parser XML es una librería de clases



Introducción (2)

- Tipos de parsers
 - Parsers tipo DOM
 - Parsers tipo "streaming"
- Parsers tipo DOM (Document Object Model)
 - Construyen un árbol en memoria equivalente al documento XML (pueden hacerlo, dado que la sintaxis de XML sigue un modelo jerárquico)
 - Ventajas
 - Sencillos de utilizar: para acceder a la información del documento, basta recorrer el árbol
 - Suelen permitir modificar/crear un árbol y generar XML a partir de él
 - Desventajas
 - Consumo de memoria alto en aplicaciones servidoras que reciben muchas peticiones que involucran parsear/generar documentos XML grandes (normalmente las aplicaciones servidoras sirven cada petición en un thread)



Introducción (y 3)

- Parsers tipo "streaming"
 - No construyen un árbol en memoria, sino que procesan secuencialmente el documento en bloques
 - Ventajas
 - Mínimo consumo de memoria
 - Especialmente útiles en las situaciones en las que los parsers de tipo DOM son prohibitivos
 - Desventajas
 - No todos tienen soporte para generar XML
 - Más difíciles de usar que los parsers de tipo DOM



Parsers Java (1)

- SAX (Simple API for XML)
 - Parser tipo streaming
 - Forma parte de Java SE (Standard Edition)
 - API: familia de paquetes **org.xml.sax**
 - Es un pequeño framework basado en eventos
 - El programador proporciona uno o varios objetos callback a los que el parser llamará cada vez que ocurra un evento de interés (apertura de un tag, cierre de un tag, un error, etc.)
 - No tiene soporte para generación de XML
 - Disponible también para otros lenguajes

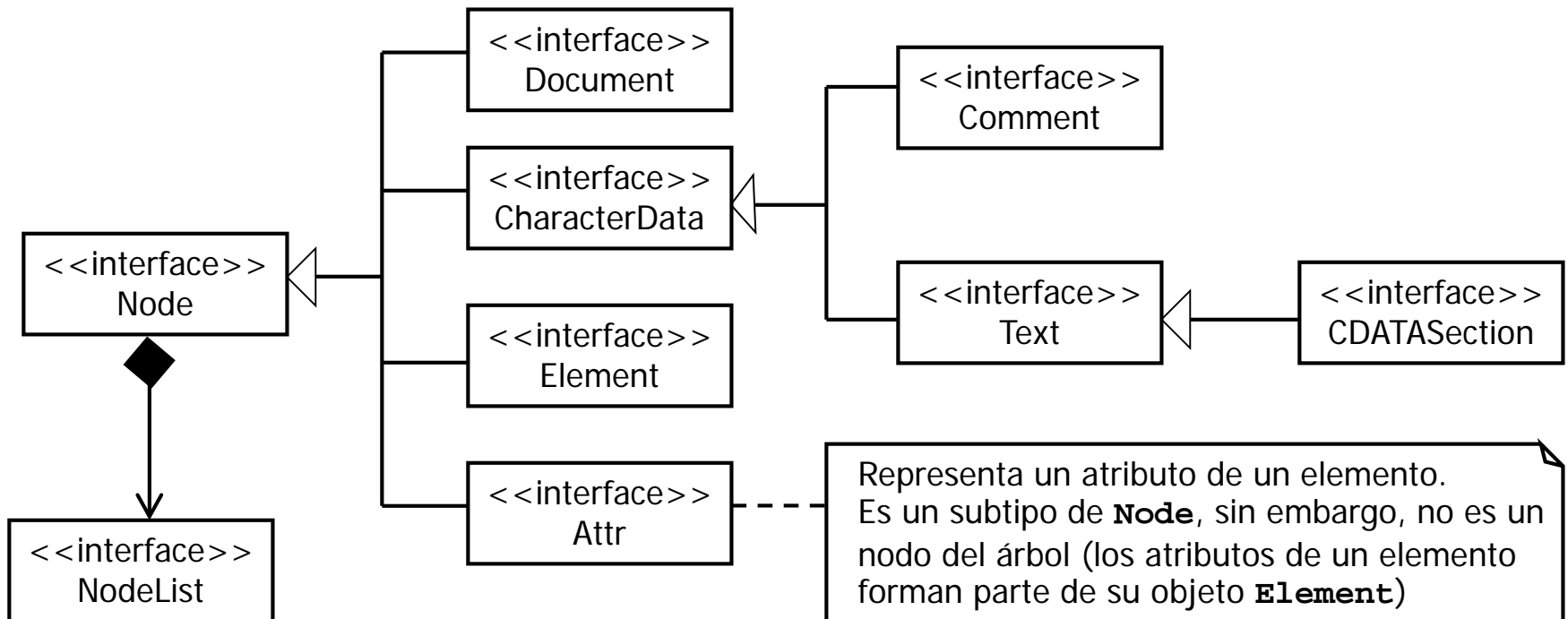


Parsers Java (2)

- DOM (Document Object Model)
 - Parser tipo DOM
 - La API está estandarizada por el W3C (<http://www.w3.org>)
=> disponible para varios lenguajes
 - Forma parte de Java SE
 - API: familia de paquetes **org.w3c.dom**
 - Tiene soporte para generación de XML

Parsers Java (3)

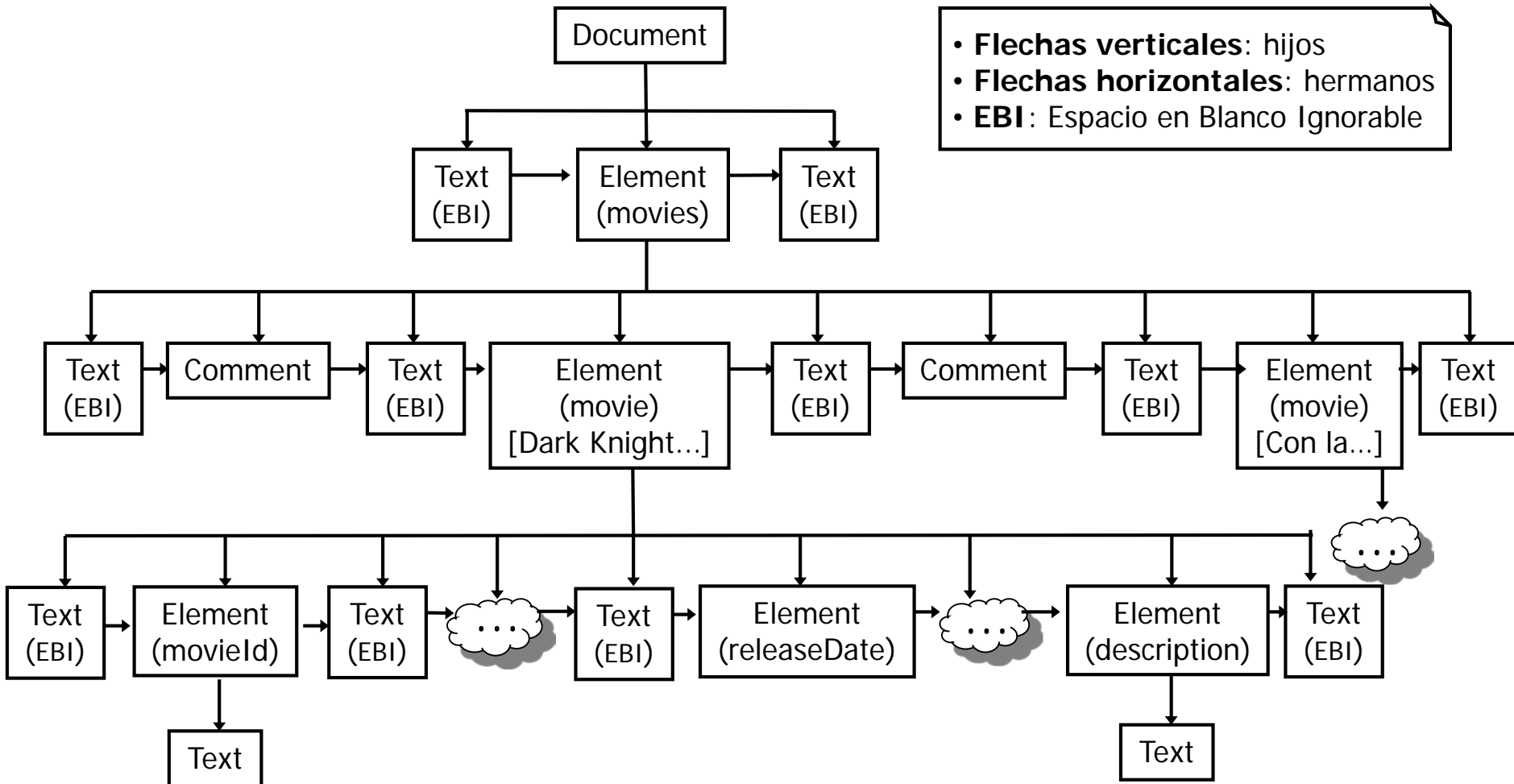
- DOM: tipos principales de nodos
 - Paquete `org.w3c.dom`



Parsers Java (4)

■ DOM: Ejemplo

- Representación DOM de **Movies.xml** (apartado 5.1)

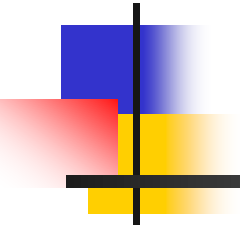




Parsers Java (y 5)

- JDOM (<http://www.jdom.org>)
 - Alternativa famosa a DOM
 - Open Source
 - Utiliza un modelo similar a DOM
 - Tiene soporte para generación de XML
 - Pensado para el lenguaje Java y más fácil de usar que la API de DOM
 - La API de DOM es demasiado tediosa de usar porque es de muy bajo nivel (no automatiza aspectos comunes)
 - No está acoplada con el lenguaje Java (e.g. no utiliza `java.util.List`, sino que dispone de sus propias listas)
 - Internamente JDOM utiliza la API de SAX y/o DOM

5.4: El lenguaje JSON





¿ Qué es JSON ?

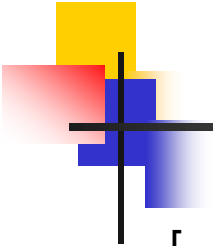
- **JavaScript Object Notation (JSON)** está basado en un subconjunto del lenguaje de programación JavaScript (estándar ECMA-262)
- Estandarizado en 2013 en el RFC 7158 y ECMA-404
 - El estándar ECMA describe solamente la sintaxis, mientras que el RFC cubre adicionalmente algunos aspectos de seguridad e interoperabilidad
 - El último RFC (8259), que define el estándar fue publicado en 2017 y sigue siendo consistente con ECMA-404
- Permite expresar información estructurada y fácilmente parseable
 - Se basa en dos estructuras soportadas por cualquier lenguaje de programación: objetos (registros, structs) y arrays
- Objetivo de este tema
 - **Entender los fundamentos básicos de JSON necesarios para esta asignatura**



Información en JSON (1)

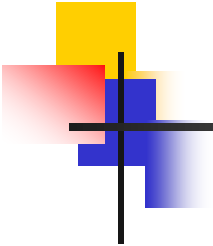
- Las siguientes transparencias ilustran el uso de JSON para expresar información de películas
 - Ese texto JSON es el que devolvería la URL de nuestro servicio y que parsearía el portal del ejemplo de la introducción de este tema

Información en JSON (2)



```
[
  {
    "movieId": 3,
    "title": "Dark Knight Rises Again",
    "runtime": 165,
    "releaseDate": { "day": 20, "month": 7, "year": 2012 },
    "directors": [ "Cristopher Nolan" ],
    "actors": [ "Christian Bale", "Morgan Freeman" ],
    "genres": [ "THR" ],
    "price": 4.99,
    "description": "Ocho años después de los acontecimientos de
The Dark Knight, Gotham se encuentra en un estado de paz. En virtud de
los poderes otorgados por la Ley Dent, el comisario Gordon casi ha
erradicado la violencia y el crimen organizado. Sin embargo, todavía se
siente culpable por el encubrimiento de los crímenes de Harvey Dent."
  },
]
```

Información en JSON (3)



```
{
  "movieId": 4,
  "title": "Con la Muerte en los Talones",
  "runtime": 136,
  "releaseDate": { "day": 26, "month": 9, "year": 1959 },
  "directors": [ "Alfred Hitchcock" ],
  "actors": [ "Cary Grant", "Eve Marie Saint", "James Manson" ],
  "genres": [ "THR", "COM" ],
  "price": 3.99,
  "description": "Roger O. Thornhill (Cary Grant) es un
ejecutivo publicitario de Nueva York al que unos espías confunden con
un agente del gobierno. Debe escapar, pero lo siguen de cerca. Durante
la fuga conoce a una atractiva mujer, Eve Kendall (Eva Marie Saint),
que lo ayuda."
}
```

]



Formato de un documento JSON (1)

■ Documento JSON

- Secuencia de caracteres (fichero, secuencia de caracteres que se envía por un socket, etc.) que tiene texto en formato JSON
- Codificación de caracteres UTF-8
- Al igual que con documentos XML, diremos que un documento JSON está **bien formado** ("well-formed") si cumple con el conjunto de reglas que a continuación se exponen
 - Estas reglas permiten construir parsers eficientes
- Un documento JSON contiene un valor que puede ser de cualquiera de los siguientes 6 tipos
 - Un objeto
 - Un array
 - Una cadena (string)
 - Un número
 - Un booleano: *true* o *false*
 - *null*
- Se distingue entre mayúsculas y minúsculas



Formato de un documento JSON (2)

- Objeto

- Se especifica entre llaves: { }
- Conjunto sin orden de pares campo-valor
 - Las claves son cadenas (strings)
 - Los valores pueden ser de cualquier tipo JSON válido
 - Las claves se separan de los valores por dos puntos
 - Los pares campo-valor se separan por comas

- Array

- Se especifican entre corchetes: []
- Colección ordenada de valores
 - Pueden ser de cualquier tipo válido
 - Se separan por comas



Formato de un documento JSON (3)

- Cadena (string)
 - Entre comillas dobles
 - Cualquier carácter Unicode excepto " o \ o caracteres de control
 - No pueden contener caracteres de retorno de carro, nueva línea, tabulador, etc.
 - El carácter \ se utiliza para
 - Escapar " o \
 - Especificar retornos de carro (\r), nueva línea (\n), tabulador (\t), etc.
 - Especificar un carácter a partir de su código Unicode: \u + 4 dígitos (hexadecimal)
- Se pueden insertar "espacios en blanco" (secuencias de espacio en blanco, retorno de carro, nueva línea y tabulador) entre dos tokens cualesquiera



Formato de un documento JSON (4)

- Algunas consideraciones respecto a XML
 - No hay comentarios
 - No existe el concepto de atributo
 - No existe el concepto de espacio de nombres
 - Los objetos y arrays "no tienen nombre" si no son el valor de un campo de un objeto
 - En XML cuando tenemos un elemento con un único subelemento no podemos distinguir si se trata de un array o no (a no ser que tengamos un esquema contra el que validarlo)



Validación de Documentos JSON (1)

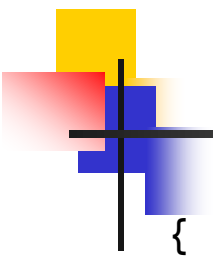
- Los esquemas JSON (JSON Schema) permiten expresar restricciones sobre el contenido de un documento JSON
 - <https://json-schema.org>
 - No está estandarizado
 - Septiembre 2019: Internet Draft 2019-09 (o Draft #8)
 - Siguiente paso: Drafts adoptados por un IETF Working Group
 - Objetivo: RFC
 - Ejemplo: especificar con qué tipo JSON se va a representar la información de una película o de una lista de películas
- Un documento JSON que cumple las restricciones de un esquema JSON se dice que es **válido**
- Una aplicación que parsee un documento JSON
 - Siempre comprueba que esté **bien formado**
 - Puede decidir comprobar si es **válido**. Veremos más adelante que validar tiene ventajas e inconvenientes



Validación de documentos JSON (y 2)

- Los esquemas JSON son a su vez documentos JSON
- A diferencia de XML, no existe una forma estandarizada para que un documento indique cómo obtener el esquema que utiliza
- A continuación, estudiamos los fundamentos básicos de los esquemas JSON mediante la construcción de un esquema JSON (**MoviesSchema.json**) para el ejemplo de las películas

MoviesSchema.json (1)



```
{  
  "$schema": "https://json-schema.org/draft/2019-09/schema",  
  "$id": "http://ws.udc.es/movies/json",  
  "type": "array",  
  "items": {  
    "type": "object",  
    "properties": {  
      "movieId": {"type": "integer"},  
      "title": {"type": "string"},  
      "runtime": {"type": "integer"},  
      "releaseDate": {  
        "type": "object",  
        "properties": {  
          "day": {"type": "integer"},  
          "month": {"type": "integer"},  
          "year": {"type": "integer"},  
        },  
        "additionalProperties": false,  
        "required": ["day", "month", "year"]  
      },  
      "additionalProperties": false,  
      "required": ["day", "month", "year"]  
    },  
  },  
}
```



MoviesSchema.json (y 2)

```
"directors": {
  "type": "array",
  "items": {"type": "string"},
  "minItems": 1
},
"actors": {
  "type": "array",
  "items": {"type": "string"},
  "minItems": 1
},
"genres": {
  "type": "array",
  "items": { "enum": ["COM", "DRA", "HOR", "ROM", "SFI",
                    "THR"] },
  "minItems": 1
},
"price": {"type": "number"},
"description": {"type": "string"}
},
"additionalProperties": false,
"required": ["movieId", "title", "runtime", "releaseDate",
             "directors", "actors", "genres", "price", "description"]
}
```



Esquemas JSON (1)

- Un esquema JSON contiene un objeto con campos que imponen un conjunto de restricciones
 - El esquema más simple (`{ }`) no impone ninguna restricción y por tanto cualquier documento JSON es válido según ese esquema
- Existen una serie de campos que son solamente descriptivos y no expresan restricciones
 - **\$schema**
 - Indica que el documento JSON es un esquema, y además puede indicar conforme a qué versión (de momento qué draft)
 - No es obligatorio pero es una buena práctica utilizarlo
 - **\$id**
 - Para asignar un identificador único al esquema
 - Tiene otros usos que no veremos
 - Tampoco es obligatorio pero es una buena práctica utilizarlo
 - **\$comment**
 - Para añadir comentarios



Esquemas JSON (2)

- Campo **type**

- Para restringir los documentos válidos a uno o varios de los 6 tipos existentes en JSON (aunque para numéricos hay dos tipos diferentes)
- Su valor puede ser
 - Una cadena especificando el nombre de un tipo
 - Numérico (**integer** o **number**), **string**, **object**, **array**, **boolean**, **null**
 - Por ejemplo: `{"type" : "string"}`
 - Un array de cadenas especificando varios nombres de tipos, en cuyo caso pasaría la validación si fuese de cualquiera de esos tipos
 - Por ejemplo: `{"type" : ["string", "null"]}`
 - El uso del tipo **null** en combinación con cualquier otro tipo es útil para valores que pueden ser nulos



Esquemas JSON (3)

- Tipos "básicos"

- String (`{ "type": "string" }`)

- Restricciones

- Longitud máxima (`maxLength`) y mínima (`minLength`)
 - Patrón (`pattern`)
 - Formatos o patrones predefinidos (`format`): `date`, `time`, `date-time`, `email`

- Tipos numéricos (`{ "type": "integer" }` o `{ "type": "number" }`)

- Dos tipos: `integer` (enteros) y `number` (enteros o decimales)

- Restricciones

- Múltiplos (`multipleOf`)
 - Rango (`maximum`, `minimum`, `exclusiveMaximum`, `exclusiveMinimum`)

- Boolean (`{ "type": "boolean" }`)



Esquemas JSON (4)

- Tipo array (`{ "type": "array" }`)
 - Por defecto los elementos pueden ser de cualquier tipo
 - Para restringir el tipo de los elementos se usa el campo **items**
 - Se especifica como un objeto que a su vez es un JSON schema
 - Para restringir el tamaño del array se puede usar **minItems** y **maxItems**
- Tipo object (`{ "type": "object" }`)
 - Por defecto, cualquier objeto es válido
 - Para restringir las propiedades que puede tener el objeto, se usa el campo **properties**
 - El valor es un objeto en el que los nombres de los campos especifican el nombre de la propiedad y el valor es un objeto (a su vez un JSON Schema) que especifica las restricciones del valor de la propiedad
 - Por defecto, solo se valida que si el objeto tiene esas propiedades, entonces tienen el tipo adecuado
 - **additionalProperties** se utiliza para especificar si el objeto puede tener o no propiedades que no estén en el esquema
 - **required** se utiliza para especificar qué propiedades son obligatorias



Esquemas JSON (y 5)

- Enumerados

- Se especifican con el campo enum
- El valor es un array con al menos un elemento y sus valores deben ser únicos
 - Pueden ser de diferentes tipos

- Existen herramientas online que permiten verificar si un documento JSON es válido conforme a un esquema

- <https://www.jsonschemavalidator.net/>



5.4: Parsing de documentos JSON



Introducción (1)

- Al igual que un documento XML, un documento JSON se apoya en dos ideas
 - **[Obligatoria] Tiene que estar bien formado**, y en consecuencia, estar construido en base a las normas de JSON
 - **[Opcional]** El contenido del documento puede estar descrito formalmente en un esquema JSON de manera que se pueda comprobar que el documento es **válido**
- Consecuencias
 - Es posible construir **parsers** genéricos que comprueban que el documento está bien formado, y **validadores** genéricos para comprobar, si se desea, que es válido
 - Existen parsers y validadores para los lenguajes más usuales



Introducción (y 2)

- Tipos de parsers
 - Parsers tipo "Modelo de árbol" o "Modelo de objetos" (equivalentes a los parsers XML tipo DOM)
 - Parsers tipo "streaming" (equivalentes a los parsers XML tipo streaming)
- Parsers tipo modelo de árbol o modelo de objetos
 - Construyen un árbol en memoria equivalente al documento JSON (pueden hacerlo, dado que JSON se basa en arrays y objetos/registros)
 - Mismas ventajas y desventajas que los parsers XML de tipo DOM
- Parsers tipo "streaming"
 - No construyen un árbol en memoria, sino que procesan secuencialmente el documento en bloques
 - Mismas ventajas y desventajas que los parsers XML de tipo streaming



Parsers Java

- JSON-P (JSON Processing)
 - API estándar para parsear, generar, transformar y consultar documentos JSON
 - Incluye tanto un parser tipo streaming como tipo modelo de árbol
 - Forma parte de Java EE / Jakarta EE
 - Existen pocas implementaciones
- Existen multitud de librerías alternativas al estándar que implementan parsers JSON de ambos tipos
 - Jackson, GSON, etc.
 - Mucho más utilizadas
 - En la asignatura utilizaremos Jackson
 - Y dentro de Jackson el parser tipo modelo de árbol



Validadores Java

- No existen APIs estándar para validar que un documento JSON es conforme a un esquema JSON
- Existen algunas librerías que implementan validadores
 - JSON Schema Validator
 - <https://github.com/java-json-tools/json-schema-validator>
 - everit.org/json.schema
 - <https://github.com/everit-org/json-schema>
 - Justify (usa JSON-P)
 - <https://github.com/leadpony/justify>
 - Lista de librerías para diferentes lenguajes: <https://json-schema.org/implementations.html>

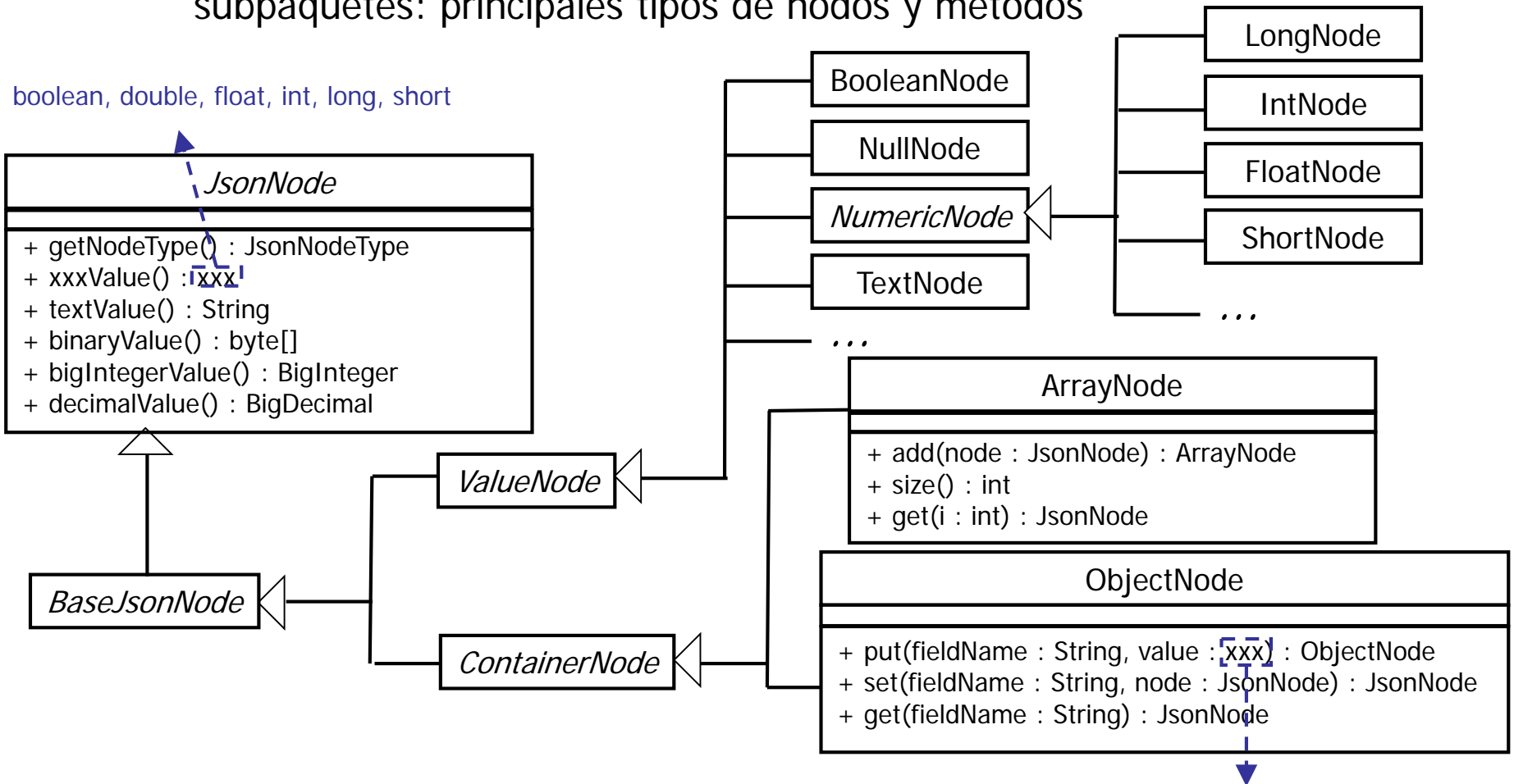


Jackson (1)

- El módulo `jackson-databind` contiene entre otras cosas
 - Las clases que modelan los distintos tipos de nodos de un árbol Jackson
 - Las clases que permiten construir un árbol Jackson a partir de un documento JSON en formato texto
 - Las clases que permiten construir un documento JSON en formato texto a partir de un árbol Jackson
 - Paquete **`com.fasterxml.jackson.databind`**
 - Internamente utiliza el módulo `jackson-core`, que define
 - La API de streaming
 - Abstracciones básicas compartidas por diferentes módulos

Jackson (2)

- Visión global de `com.fasterxml.jackson.databind` y subpaquetes: principales tipos de nodos y métodos



- `BigInteger`, `BigDecimal`, `byte[]`, `String`
- Tipos básicos y contrapartidas objetuales

Jackson (3)

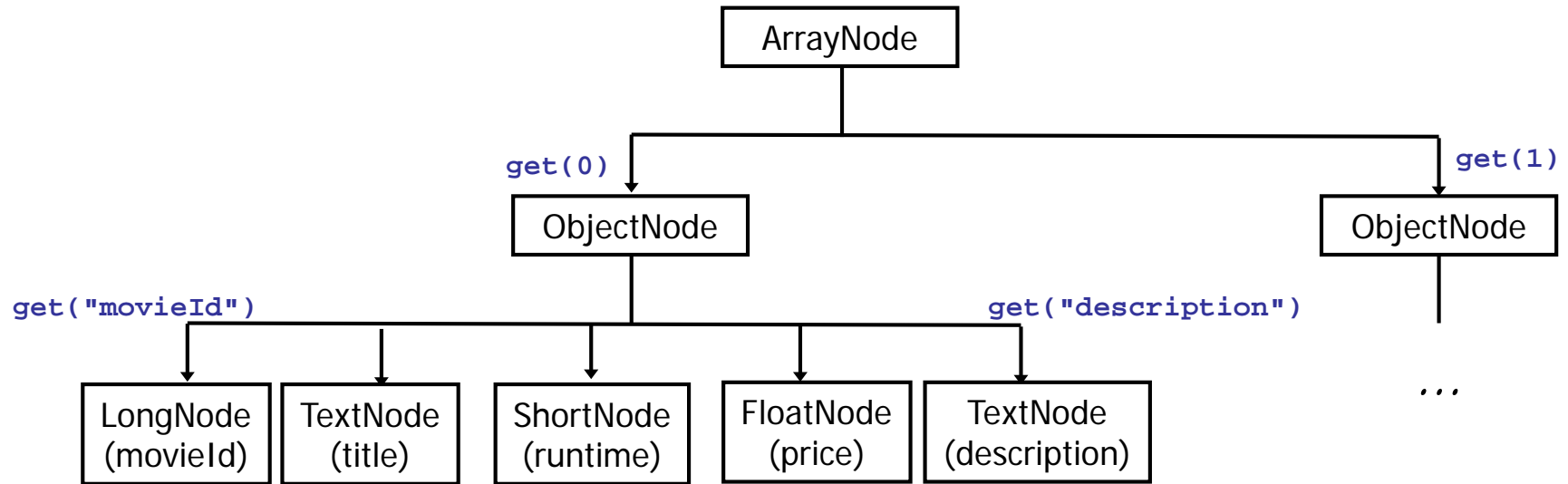
Ejemplo: documento JSON del ejemplo ws-movies

- Para cada película
 - No se expone fecha de creación (el diseñador ha decidido que no es relevante para aplicaciones externas)
 - Elemento **movieId** es opcional

```
[
  {
    "movieId": 3,
    "title": "Dark Knight Rises Again",
    "runtime": 165,
    "price": 4.99,
    "description": "Ocho años después de ... "
  },
  {
    "movieId": 4,
    "title": "Con la Muerte en los Talones",
    "runtime": 136,
    "price": 3.99,
    "description": " Roger O. Thornhill (Cary Grant) es un ... "
  }
]
```


Jackson (4)

- Representación del documento JSON



Jackson (5)

■ Comentarios

- Los métodos **xxxValue** de **JsonNode** no lanzan una excepción si el valor del nodo no se puede convertir al tipo correspondiente
 - Devuelven un valor por defecto (por ejemplo, **0** para los tipos numéricos, **null** para **String** y **byte[]**, **false** para **boolean**)
- Los métodos **put** de **ObjectNode** mostrados en el diagrama
 - Crean el nodo adecuado según el tipo del valor que reciben
 - Devuelven la misma instancia sobre la que se invocan, lo que permite "encadenar" llamadas
- **JsonNodeType** es un enumerado que tiene un valor por cada tipo de nodo

Jackson (6)

- Visión global de **com.fasterxml.jackson.databind**:
Transformaciones texto JSON <-> árbol Jackson

ObjectMapper
+ ObjectMapper() + readTree(in : InputStream) : JsonNode + writer(pp : PrettyPrinter) : ObjectWriter

ObjectWriter
+ writeValue(out : OutputStream, obj : Object) : void

- El método **readTree** parsea la entrada y devuelve el nodo raíz del árbol
- El método **writer** lo invocaremos pasándole una instancia de **DefaultPrettyPrinter** para que devuelva una instancia de **ObjectWriter** que genere JSON en formato "bonito"
 - El método **writeValue** escribe el JSON correspondiente al objeto que recibe (en nuestro caso un **JsonNode**, correspondiente al nodo raíz del árbol)

Jackson (7)

- Visión global de `com.fasterxml.jackson.databind`:
Creación de nodos objeto y array vacíos

JsonNodeFactory
+ <u>instance</u> : JsonNodeFactory
+ objectNode() : ObjectNode + arrayNode() : ArrayNode

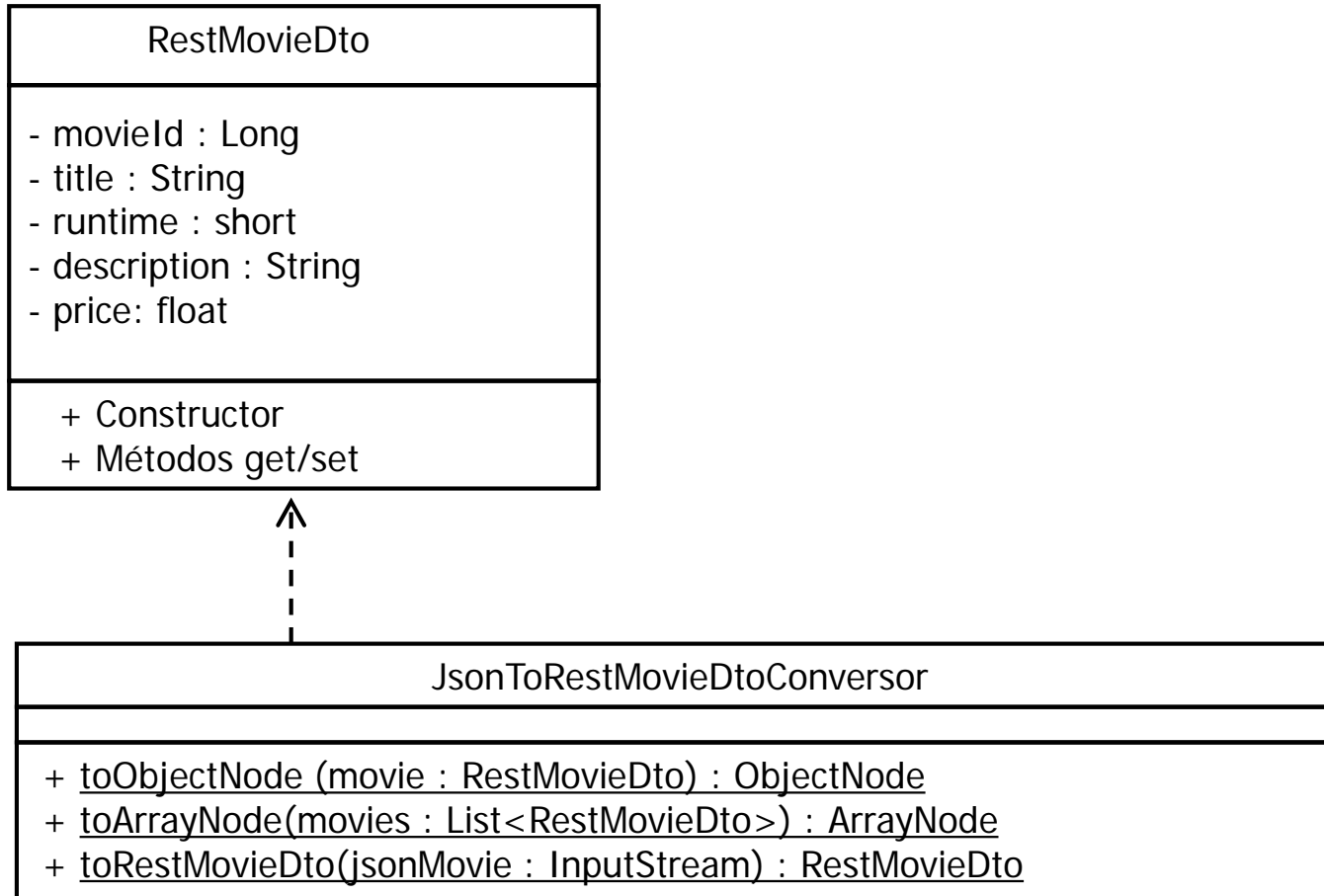
- Dado que no tiene estado se puede usar la misma instancia siempre
 - Se obtiene a través del atributo **instance**



Jackson (8)

- Mostraremos como ejemplo parte de la clase **JsonToRestMovieDtoConversor** de los ejemplos de la asignatura
 - Forma parte del caso de estudio del tema 6
 - Dispone de métodos para convertir instancias de **RestMovieDto** a JSON, y viceversa
 - **toObjectNode(RestMovieDto) : ObjectNode**
 - Recibe un objeto **RestMovieDto** y genera un árbol Jackson con la misma información
 - **toArrayNode(List<RestMovieDto>) : ArrayNode**
 - Recibe una lista de objetos **RestMovieDto** y genera un árbol Jackson con la misma información (delega en el método anterior)
 - **toRestMovieDto(InputStream) : RestMovieDto**
 - Recibe el texto de un documento JSON que tiene la información de una película y crea un objeto **RestMovieDto** con la misma información

Jackson (y 9)





es.udc.ws.movies.restservice.json.JsonToRestMovieDtoConversor (1)

```
package es.udc.ws.movies.restservice.json;

...
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ArrayNode;
import com.fasterxml.jackson.databind.node.JsonNodeFactory;
import com.fasterxml.jackson.databind.node.JsonNodeType;
import com.fasterxml.jackson.databind.node.ObjectNode;

import es.udc.ws.movies.restservice.dto.RestMovieDto;
import es.udc.ws.util.json.ObjectMapperFactory;
import es.udc.ws.util.json.exceptions.ParsingException;

public class JsonToRestMovieDtoConversor {
```

```
public static ObjectNode toObjectNode(RestMovieDto movie) {

    ObjectNode movieObject = JsonNodeFactory.instance.objectNode();

    if (movie.getMovieId() != null) {
        movieObject.put("movieId", movie.getMovieId());
    }
    movieObject.put("title", movie.getTitle()).
        put("runtime", movie.getRuntime()).
        put("price", movie.getPrice()).
        put("description", movie.getDescription());

    return movieObject;
}

public static ArrayNode toArrayNode(List<RestMovieDto> movies) {

    ArrayNode moviesNode = JsonNodeFactory.instance.arrayNode();
    for (int i = 0; i < movies.size(); i++) {
        RestMovieDto movieDto = movies.get(i);
        ObjectNode movieObject = toObjectNode(movieDto);
        moviesNode.add(movieObject);
    }

    return moviesNode;
}
```

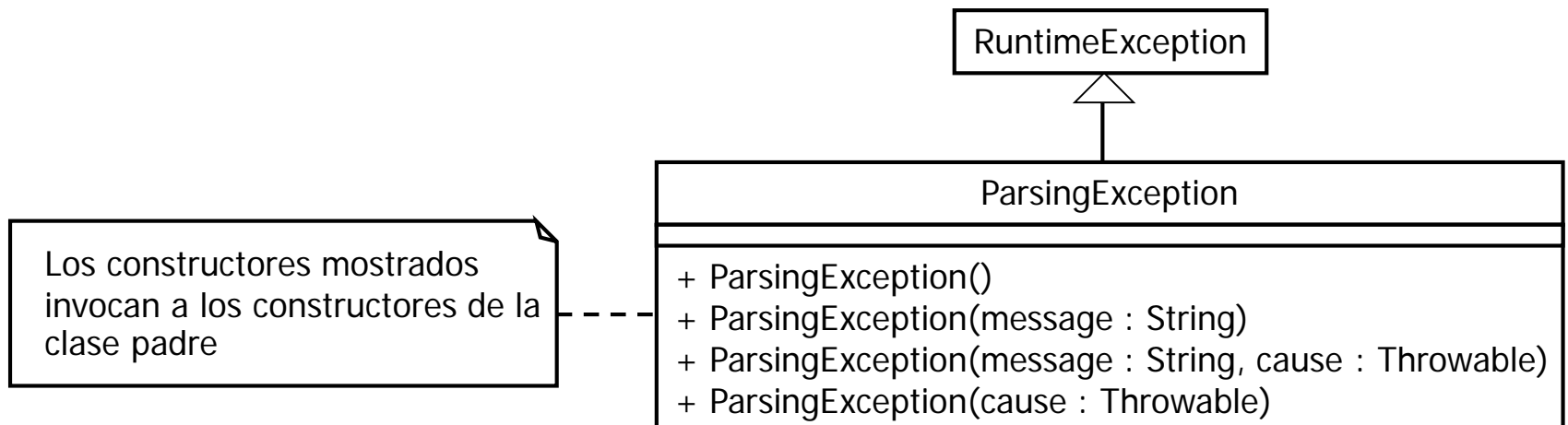


```
public static RestMovieDto toRestMovieDto(InputStream jsonMovie) throws
    ParsingException {
    try {
        ObjectMapper objectMapper = ObjectMapperFactory.instance();
        JsonNode rootNode = objectMapper.readTree(jsonMovie);
        if (rootNode.getNodeType() != JsonNodeType.OBJECT) {
            throw new ParsingException("Unrecognized JSON (object expected)");
        } else {
            ObjectNode movieObject = (ObjectNode) rootNode;
            JsonNode movieIdNode = movieObject.get("movieId");
            Long movieId =
                (movieIdNode != null) ? movieIdNode.longValue() : null;
            String title = movieObject.get("title").textValue().trim();
            String description =
                movieObject.get("description").textValue().trim();
            short runtime = movieObject.get("runtime").shortValue();
            float price = movieObject.get("price").floatValue();

            return
                new RestMovieDto(movieId, title, runtime, description, price);
        }
    } catch (ParsingException ex) {
        throw ex;
    } catch (Exception e) {
        throw new ParsingException(e);
    }
}
```

Comentarios (1)

- `es.udc.ws.movies.json.exceptions.ParsingException`
 - Los métodos públicos de la clase `JsonToRestMovieDtoConverter` devuelven esta excepción cuando hay algún problema durante el proceso de parsing
 - Forma parte del subsistema de utilidades de los ejemplos de la asignatura
 - Extiende a `RuntimeException`
 - Representa un error grave
 - Sólo es preciso capturarla en los lugares en los que explícitamente se quiere tratar
 - En el resto de sitios, la excepción "fluye" hacia arriba
 - Como cualquier tipo de excepción (`java.lang.Throwable`), permite especificar un mensaje y/o encapsular una excepción (la que produjo el problema)





Comentarios (y 2)

- **es.udc.ws.movies.json.ObjectMapperFactory**
 - Forma parte del subsistema de utilidades de los ejemplos de la asignatura
 - Clase utilidad que devuelve siempre la misma instancia de **ObjectMapper**
 - **ObjectMapper** es thread-safe
- Como justificaremos en el tema 6, no siempre queremos que los clientes y/o servidores validen los documentos que reciben, sino simplemente que comprueben que
 - Los documentos estén bien formados
 - Los elementos que se requieran estén presentes
- NOTA: **JsonToRestMovieDtoConverter** no comprueba que los elementos requeridos tengan valores válidos (se comprueba en la capa modelo)