# Unidad 4. Definición de esquemas y vocabularios en xml.

## 1. Documento XML. Estructura y sintaxis.

Al crear un documento XML es importante garantizar que la estructura de datos del documento es la que se desea y no otra, además de asegurar que todos los documentos del mismo tipo mantienen la misma estructura. Es decir, se trata de asegurar una normalización en el formato del documento XML. Por ejemplo, si definimos una estructura XML para guardar los datos de una empresa y queremos guardar en 100 documentos XML de ese tipo los datos de 100 empresas, sería importante normalizar esa estructura XML, para que los 100 ficheros XML cumplan las mismas normas; si al crear uno de esos documentos XML no se cumple alguna norma, el validador mostrará un aviso, así se obliga al usuario a respetar esas normas.

En la primera unidad vimos que un documento XML básico estaba formado por un prólogo y un ejemplar:

- Prólogo: informa al intérprete encargado de procesar el documento (que puede ser el navegador web) de todos aquellos datos que necesita para realizar su trabajo. Consta de dos partes:
    - Definición de XML: donde se indica la versión de XML que se utiliza, el código (encoding) de los datos a procesar y la autonomía (standalone) del documento. Este último dato hasta ahora siempre ha sido "yes" ya que los documentos generados eran independientes. Por ejemplo:

      <?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>

    - Declaración del tipo de documento: que es <!DOCTYPE ejemplar>, donde ejemplar es el elemento raíz del documento XML (nosotros hasta el momento no hemos utilizado esta declaración).

- Ejemplar: es el elemento que contiene todos los datos del documento, el elemento raíz, que debe ser único. Es decir, en el documento XML habrá elementos que mantienen una estructura de árbol, en la que el elemento raíz es el ejemplar y las hojas los elementos terminales, que son aquellos que no contienen otros elementos. Los elementos pueden tener a su vez atributos. Por ejemplo:

```
<!DOCTYPE stock>
<stock>
        Aquí van todos los elementos y sus atributos
</stock>
```

### 1.1. Tipo de documento.

El tipo de documento permite definir restricciones y características del documento XML, o sea que el documento deberá cumplir una serie de normas, que serán previamente definidas.

En la declaración del tipo de documento podemos distinguir entre:

- La declaración del tipo de documento en sí misma, que se escribe después del prólogo y será:

  <!DOCTYPE ejemplar>

  Como vemos, el nombre del tipo de documento debe coincidir con el ejemplar del documento XML. Por tanto, el documento XML quedaría:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE ejemplar>
<ejemplar>
  ...
</ejemplar>
```

- La definición del tipo de documento, que permite asociar al documento XML una serie de cualidades (normas a cumplir); en este apartado se definen los tipos de los elementos, atributos y notaciones que se pueden utilizar en el documento, así como las restricciones del documento, valores por defecto, etc. Es decir, se indican las normas que debe cumplir el documento XML. Esto se denomina Definición de Tipo de Documento, DTD (Document Type Definition).

Para escribir el DTD, XML está provisto de ciertas estructuras llamadas declaraciones de marcado (restricciones o normas a cumplir), las cuales pueden ser internas o externas. Para un documento XML se puede crear una mezcla de declaraciones de marcado internas y externas. Por ejemplo, se puede crear un documento DTD externo, que sea compartido por varios documentos XML, y después de forma interna en cada documento XML se pueden definir declaraciones DTD internas específicas para el documento XML en cuestión. Por tanto, el conjunto de declaraciones de marcado puede tener dos subconjuntos, el interno y el externo:

- Subconjunto interno: contiene las declaraciones que pertenecen exclusivamente a un documento y no es posible compartirlas; se localizan dentro de unos corchetes que siguen a la declaración de tipo del documento, del modo:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<!DOCTYPE ejemplar [
        Aquí van las declaraciones DTD internas
]>
<ejemplar>
        etc...
</ejemplar>
```

- Subconjunto externo: las declaraciones de marcado externas se escriben en un fichero con extensión .dtd que puede situarse en el mismo directorio que el documento XML. Habitualmente son declaraciones que pueden ser compartidas entre múltiples documentos XML que pertenecen al mismo tipo. En este caso la declaración de documento autónomo ha de ser negativa (standalone="no"), ya que es necesario el fichero del subconjunto externo para la correcta interpretación del documento. Se suele definir con la palabra clave SYSTEM, del modo siguiente:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
```

```
<!DOCTYPE ejemplar SYSTEM "URI">
<ejemplar>
        etc...
</ejemplar>
```

Donde URI es la localización del documento externo DTD, es decir del fichero con extensión .dtd, por ejemplo:

```
<!DOCTYPE ejemplar SYSTEM "fichero.dtd">
```

Por tanto, para aplicar a un documento XML declaraciones de marcado internas y también externas, se usan ambas sintaxis mezcladas, de la forma:

```
<!DOCTYPE ejemplar SYSTEM "fichero.dtd" [
        Aquí van las declaraciones DTD internas
]>
<ejemplar>
        etc...
</ejemplar>
```

Una ventaja de utilizar declaraciones externas, en un fichero .dtd, en lugar de declaraciones internas, dentro de DOCTYPE, es que el mismo fichero .dtd podría aplicarse a muchos ficheros .xml. En el caso de utilizar declaraciones internas, deberían repetirse las mismas declaraciones en cada fichero .xml donde se deseen aplicar.

Se puede decir que las declaraciones de tipo de documento pueden tener parte interna y parte externa, ya que incluyen las definiciones del tipo de documento.

**1.2. Definición de la sintaxis de documentos XML.**

De la Unidad 1 ya sabemos que un elemento de un documento XML es un grupo formado por una etiqueta de apertura, otra de cierre y el contenido que hay entre ambas. La distribución de los elementos está jerarquizada según una estructura de árbol, lo que implica que es posible anidarlos pero no entrelazarlos. Por tanto, el entrelazado de elementos no se puede usar para construir elementos. El orden es importante para los elementos, pero no para los atributos. Además los atributos no pueden tener nodos que dependan de ellos, por tanto solo pueden corresponder con hojas de la estructura de árbol que jerarquiza los datos. Las hojas pueden ser atributos o elementos terminales, que son los que no contienen otros elementos.

No existe una norma precisa para decidir si un dato debe guardarse en un elemento o en un atributo, pero se pueden usar los siguientes criterios orientativos:

- El dato se guarda en un elemento si cumple alguna de las siguientes condiciones:
  - Es de un tamaño considerable.
  - Su valor cambia frecuentemente.
  - Contiene elementos.
  - Su valor va a ser mostrado a un usuario o aplicación.

- El dato se guarda en un atributo si cumple alguna de las siguientes condiciones:
  - El dato es de pequeño tamaño y su valor raramente cambia.
  - El dato solo puede tener unos cuantos valores fijos.
  - El dato guía el procesamiento XML pero no se va a mostrar.

También en la Unidad 1 se tratan los espacios de nombres o namespaces, que permiten:

- Diferenciar entre los elementos y atributos de distintos vocabularios con diferentes significados que comparten nombre.

- Agrupar todos los elementos y atributos relacionados de una aplicación XML para que el software pueda reconocerlos con facilidad.

Un espacio de nombres se declara como:

xmnls:prefijo="URI_namespace"

donde "prefijo" informa sobre cuál es el vocabulario al que está asociada esa definición. URI_namespace es la localización del conjunto del vocabulario del espacio de nombres al que se hace referencia. Por ejemplo:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<asistentes xmlns:alumnos="http://ASIR/alumnos"
            xmlns:profesores="http://ASIR/profesores">

<alumnos:nombre>Fernando Fernández González</alumnos:nombre>
<alumnos:nombre>Isabel González Fernández</alumnos:nombre>
<alumnos:nombre>Ricardo Martínez López</alumnos:nombre>
<profesores:nombre>Pilar Ruiz Pérez</profesores:nombre>
<profesores:nombre>Tomás Rodríguez Hernández</profesores:nombre>

</asistentes>
```

En ese ejemplo el ejemplar es "asistentes". Se usan dos espacios de nombres, cuyos prefijos son: "alumnos" y "profesores". Para hacer referencia al elemento "nombre" debemos indicar a qué espacio de nombres (o vocabulario) nos referimos, ya que "nombre" existe en los dos vocabularios (el alumno tiene nombre y el profesor también); para ello se pone el prefijo antes de "nombre", del modo: alumnos:nombre o profesores:nombre.

**2. DTD: Definiciones de Tipo de Documento.**

Un DTD es el método más sencillo para normalizar documentos XML. No es un lenguaje XML, por lo que no respeta la sintaxis XML, pero tiene una sintaxis sencilla.

Se trata de crear una relación precisa de los elementos que pueden aparecer en un documento XML y dónde pueden aparecer, así como el contenido y los atributos del mismo. El DTD impone unas restricciones a los datos, que éstos deben cumplir.

Como veremos más adelante, un documento DTD permite:

- Especificar la estructura del documento.
- Determinar los elementos y atributos que forman un elemento de un documento XML, así como el orden en el que han de aparecer.
- Reflejar restricciones de integridad referencial utilizando ID e IDREF.
- Definir restricciones que los datos del documento XML deben cumplir.
- Utilizar unos pequeños mecanismos de abstracción comparables a las macros, que son las entidades.

- Incluir documentos externos.

Sin embargo, los DTD tienen los siguientes inconvenientes:

- Su sintaxis no es XML.
- No soportan espacios de nombres.
- No definen tipos para los datos. Solo hay un tipo de elementos terminales, que son los datos textuales.
- No permite las secuencias no ordenadas.
- No es posible formar claves o identificadores a partir de varios atributos o elementos.
- Una vez que se define un DTD no es posible añadir nuevos vocabularios.

En la primera unidad comprobábamos si un documento XML estaba "Bien-Formado". Ahora que el documento XML lleva asociado DTD, podremos comprobar si el documento XML es "Válido". Hay programas que analizan y validan (parser) el documento XML, avisando cuando éste no cumple las normas indicadas en el DTD.

Nosotros usaremos el programa "XML Copy Editor", que puede descargarse de http://xml-copy-editor.sourceforge.net de forma gratuita (licencia GNU). Además de ofrecer un editor de código XML, también permite comprobar si el fichero .xml es correcto sintácticamente, es decir si está "Bien-Formado" (well-formed). También permite "Validar", es decir comprobar si el .xml cumple las condiciones (restricciones) impuestas en un fichero .dtd (o .xsd, como veremos más adelante). Tenemos:

- Marca azul de la izquierda o tecla <F2>: "Comprobar Bien-Formado".
- Marca verde de la derecha o tecla <F5>: "Validar".

Algunos navegadores no revisan correctamente los ficheros .xml que tienen las definiciones en un fichero aparte .dtd (o .xsd como se verá posteriormente). Por ello, conviene realizar esta revisión del fichero .xml validándolo con "XML Copy Editor", en lugar de utilizar el navegador.

Con el programa "XML Copy Editor" se puede mantener abierto el fichero .xml y el .dtd simultáneamente; para el fichero .xml se puede utilizar la opción "Comprobar Bien-Formado" y la opción "Validar". Si hay algún error, el programa indica la línea y columna donde se encuentra, para ayudar a resolverlo.

Sin embargo, al fichero .dtd no se puede aplicar ni la opción "Comprobar Bien-Formado" ni la opción "Validar", ya que no respeta la sintaxis XML.

Los DTD no admiten tipos de datos, es decir que no se puede indicar que un elemento del fichero .xml debe ser de un tipo concreto: numérico entero, numérico con decimales, de texto, etc. Con DTD todos los datos son cadenas de caracteres.

Veamos un ejemplo muy simple de fichero.xml con definiciones de tipo en fichero.dtd:

Fichero ejemplo.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE datos SYSTEM "fichero.dtd">
```

```
<datos>
  <nombrealumno>Juan Pinto</nombrealumno>
  <nombrealumno>Elena Pons</nombrealumno>
</datos>
```

Fichero fichero.dtd

```
<!ELEMENT datos (nombrealumno+)>
<!ELEMENT nombrealumno (#PCDATA)>
```

Como vemos, en fichero.dtd se especifican algunas condiciones o restricciones que deben cumplir los elementos usados (datos y nombrealumno) en el fichero .xml.

**2.1. Declaraciones de tipos de elementos terminales (no contienen otros elementos).**

El elemento persona del siguiente ejemplo no es elemento terminal; los elementos nombre y apellidos sí son terminales.

```
<persona>
  <nombre>Juan</nombre>
  <apellidos>Ramos</apellidos>
</persona>
```

Para declarar un elemento terminal en DTD se usa la sintaxis:

<!ELEMENT elemento tipo>

Los tipos de elementos terminales son:

- EMPTY: un elemento terminal de tipo EMPTY estará vacío, es decir no guardará ningún dato (no puede haber nada entre la etiqueta de apertura y la de cierre), pero sí puede tener atributos; por ejemplo:

```
<alumno Nombre="Roberto" />
<alumno Nombre="Juan" />
<alumno Nombre="Alberto" />
```

El elemento alumno podría declararse en el .dtd del siguiente modo:

<!ELEMENT alumno EMPTY>

Al escribir esa línea en el fichero .dtd, estamos obligando a que los elementos "alumno" en el .xml estén vacíos; si al crear el .xml por error se escribe algo en el elemento alumno, el programa validador avisará del error. Por ejemplo, si se escribe:

<alumno Nombre="Roberto">Ramos</alumno>

Al "Validar" daría error.

Otro ejemplo de EMPTY podría ser:

Si en el .xml tenemos: <fecha_inicio dia="1" mes="2" anio="2016"/>

Para fecha_inicio, en el .dtd tendríamos: <!ELEMENT fecha_inicio EMPTY>

- ANY: puede contener cualquier cosa, es decir puede incluir otros elementos y también estar vacío; no se analiza qué contiene (no hay un modelo concreto, como sí lo hay para #PCDATA); se suele usar para realizar pruebas, pero no para publicar documentos xml.

- (#PCDATA), Parsed Character DATA: texto plano analizado como tal; el elemento puede contener un conjunto de caracteres, pero no otros elementos (es un tipo para elemento terminal). Un caso puede verse en el ejemplo del apartado anterior:

```
<!ELEMENT nombrealumno (#PCDATA)>
```

Deben respetarse las mayúsculas; es decir, si se escribe (#pcdata) dará error. Además, después del nombre del elemento, que en este caso es "nombrealumno", debe insertarse siempre un espacio en blanco antes de abrir el paréntesis.

**2.2. Declaraciones de tipos de elementos no terminales (contienen otros elementos).**

Los elementos no terminales contienen otros elementos, que pueden ser a su vez no terminales o terminales.

Veamos un ejemplo de fichero.dtd:

```
<!ELEMENT alumno (nombre, apellidos)>      <!-- alumno: elemento no terminal -->
<!ELEMENT apellidos (apell1, apell2)>      <!-- apellidos: elemento no terminal -->
<!ELEMENT apell2 (#PCDATA)>                <!-- apell2: elemento terminal -->
<!ELEMENT apell1 (#PCDATA)>                <!-- apell1: elemento terminal -->
<!ELEMENT nombre (#PCDATA)>                <!-- nombre: elemento terminal -->
```

Las normas o restricciones que imponen esas declaraciones son: el elemento no terminal alumno se forma obligatoriamente por los elementos nombre y apellidos; si a un alumno en el fichero .xml no le pones apellidos (o nombre), dará error al "Validar". Además, el elemento apellidos es también no terminal (como alumno) y está formado por los elementos apell1 y apell2. Finalmente se declaran los elementos terminales apell1, apell2 y nombre, todos de tipo (#PCDATA).

Con ese fichero.dtd anterior, en su fichero .xml asociado debe escribirse primero el nombre y después los apellidos (la etiqueta <nombre> debe estar antes que la etiqueta <apellidos>), es decir que debe respetarse el orden indicado en la declaración del elemento no terminal; si se pone primero apellidos dará error al "Validar" el .xml. Por ejemplo, el siguiente fichero.xml daría error de validación, por tener el nombre después de los apellidos (no cumple la restricción del .dtd):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE alumno SYSTEM "fichero.dtd">
<alumno>
  <apellidos>
    <apell1>Pinto</apell1>
    <apell2>Ramos</apell2>
  </apellidos>
  <nombre>Juan</nombre>
</alumno>
```

Igualmente, si en el .xml se escribe primero apell2 y después apell1, también dará error de validación. Sin embargo, como vemos en el fichero.dtd, el elemento terminal "nombre" puede declararse al final del .dtd (con <!ELEMENT nombre (#PCDATA)>), ya que el orden de las declaraciones no importa; también apell2 está declarado antes que apell1 sin dar errores.

Los elementos que componen un elemento no terminal deben ponerse entre paréntesis, por ejemplo (nombre, apellidos) del siguiente ejemplo:

<!ELEMENT alumno (nombre, apellidos)>

Si un elemento se compone sólo de otro elemento (caso que no suele darse; por ejemplo, una agenda de una sola persona), también deben utilizarse esos paréntesis:

<!ELEMENT agenda (persona)>

Para indicar las veces que puede aparecer un elemento en el fichero .xml se utilizan los operadores. Si no se utiliza ningún operador, el elemento indicado en el fichero .dtd debe aparecer obligatoriamente en el .xml una y sólo una vez. Para modificar esa situación se pueden aplicar los operadores siguientes:

- Operador ? El elemento no es obligatorio, es decir que puede aparecer una vez o ninguna.

<!ELEMENT telefono (teltrabajo?, telcasa )>

El elemento telefono puede tener un elemento teltrabajo (y no más de uno) y tiene obligatoriamente un elemento telcasa (y sólo uno). Con los siguientes ficheros .xml tendríamos:

```
<telefono>
        <teltrabajo>11111111</teltrabajo>
</telefono>
```
INCORRECTO; debe haber un elemento telcasa obligatoriamente.

```
<telefono>
        <telcasa>11111111</telcasa>
</telefono>
```
CORRECTO; tiene un elemento telcasa y teltrabajo no es obligatorio.

```
<telefono>
        <telcasa>11111111</telcasa>
        <telcasa>22222222</telcasa>
</telefono>
```
INCORRECTO; no puede haber más de un telcasa.

```
<telefono>
        <teltrabajo>11111111</teltrabajo>
        <telcasa>11111111</telcasa>
</telefono>
```
CORRECTO; tiene un teltrabajo y un telcasa.

```
<telefono>
        <telcasa>11111111</telcasa>
        <teltrabajo>11111111</teltrabajo>
</telefono>
```
INCORRECTO; teltrabajo debe estar antes que telcasa.

El operador debe escribirse pegado al elemento, es decir no se puede dejar un espacio en blanco entre ellos, por lo que lo siguiente sería erróneo:

<!ELEMENT telefono (teltrabajo ?, telcasa )>

Un operador puede aplicarse a un conjunto de elementos, que estarán entre paréntesis, por lo que el operador se colocará fuera del paréntesis, del modo siguiente:

<!ELEMENT persona (telefono, (nombre, apell)? )>

En ese ejemplo, en el .xml una "persona" tiene un "telefono", pero la pareja "nombre" con "apell" puede tenerla una o ninguna. El ? afecta a los elementos dentro del paréntesis en su conjunto (no por separado), por lo que no puede haber una persona que tenga "nombre" sin "apell" ni al revés; si tiene un "nombre", tendrá también un "apell", y si no tiene "nombre" tampoco tendrá "apell". Para aplicarlo por separado "nombre" y "apell" sería:

<!ELEMENT persona (telefono, nombre?, apell? )>

En ese caso, la persona puede tener nombre sin apell, así como apell sin nombre. Dicho de otro modo, la persona tendrá obligatoriamente un "telefono", tendrá un "nombre" o ninguno y tendrá un "apell" o ninguno.

- Operador + El elemento debe aparecer en el fichero .xml al menos una vez, es decir que debe aparecer una o más veces. Por ejemplo:

<!ELEMENT agenda (persona)+ >

en este caso también se puede poner el signo + dentro del paréntesis

<!ELEMENT agenda (persona+) >

El elemento "agenda" está formado por una "persona" o más. La agenda no puede estar vacía.

Otro ejemplo:

<!ELEMENT direccion (calle, numero)+ >

El elemento direccion está formado por la pareja calle y número, pero esta pareja podría aparecer más de una vez; sin embargo, no puede aparecer la calle sin número, ni tampoco el número sin la calle. Tendríamos por tanto en el xml:

```
<direccion>
        <calle>Mayor</calle>            INCORRECTO: falta el número
</direccion>

<direccion>
        <numero>14</numero>             INCORRECTO: falta la calle
</direccion>

<direccion>                             INCORRECTO: la pareja calle, número debe
</direccion>                            aparecer al menos una vez
```

```
<direccion>
        <calle>Mayor</calle>
        <numero>14</numero>
</direccion>
```

CORRECTO: aparece la pareja calle, número al menos una vez, en este caso es 1 vez.

```
<direccion>
        <calle>Mayor</calle>
        <numero>14</numero>
        <calle>Mayor</calle>
        <numero>14</numero>
</direccion>
```

CORRECTO: aparece la pareja calle, número al menos una vez, en este caso son 2 veces.

Otro ejemplo:

```
<!ELEMENT provincia (nombre, (cp, ciudad)+) >
```

El + afecta a todos los elementos que estén dentro del paréntesis anterior. Es decir, el elemento provincia tiene obligatoriamente un nombre y una pareja de elementos cp y ciudad, pero esta pareja podrá aparecer más de una vez. Según ese ejemplo, no puede haber provincia que tenga cp y no tenga ciudad, ni provincia que tenga ciudad y no tenga cp; al poner los paréntesis de la forma (cp, ciudad) ambos elementos deben ir en pareja, cp con ciudad. Al poner el + después del paréntesis del modo (cp, ciudad)+ afecta a la pareja en su conjunto. Si en el XML aparece cp tres veces, la ciudad también debe estar las tres veces, formando 3 parejas de cp, ciudad. El siguiente XML cumple correctamente la declaración anterior:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE provincia SYSTEM "fichero.dtd">
<provincia>
  <nombre>BADAJOZ</nombre>
  <cp>06001</cp>
  <ciudad>Badajoz</ciudad>
  <cp>06300</cp>
  <ciudad>Zafra</ciudad>
</provincia>
```

Otro ejemplo:

```
<!ELEMENT provincia (nombre, cp, ciudad+) >
```

En este caso la provincia tendrá un nombre, un cp y una o más ciudades.

- Operador * El elemento aparecerá cero 0 o más veces. Por ejemplo:

```
<!ELEMENT agenda (persona)* >
```

en este caso también se puede poner el asterisco * dentro del paréntesis

```
<!ELEMENT agenda (persona*) >
```

El elemento "agenda" está formado por ninguna "persona" o más. Ahora la agenda sí puede estar vacía.

Otro ejemplo:

<!ELEMENT provincia (nombre, (cp, ciudad)*) >

El elemento provincia tiene obligatoriamente un nombre y la pareja de elementos cp y ciudad puede que no aparezca ni una sola vez, aunque también esa pareja puede aparecer muchas veces dentro de cada provincia. Pero igual que antes, si aparece dos veces el cp, deberá aparecer dos veces la ciudad (van en pareja). El siguiente XML no cumple la declaración anterior y daría error de validación, por tener un cp sin ciudad:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE provincia SYSTEM "fichero.dtd">
<provincia>
  <nombre>BADAJOZ</nombre>
  <cp>06001</cp>
  <cp>06300</cp>
  <ciudad>Zafra</ciudad>
</provincia>
```

- Operador |  Es el operador "elección". Se escribe entre dos elementos o más (en lugar de la coma) para indicar que debe elegirse uno de los elementos en cuestión (el carácter | se escribe con la tecla del número 1). Los elementos junto con el operador | deben ir entre paréntesis. Por ejemplo:

<!ELEMENT ciudad (cp | nombre) >

En el .xml el elemento ciudad tiene un elemento cp o un elemento nombre, pero no ambos, debe elegirse uno de los dos. Si no tiene ninguno de los dos también dará error; debe tener siempre uno y sólo uno de los dos

Otro ejemplo:

<!ELEMENT alumno (tfno | (nombre, apell) )>

El elemento alumno tiene un tfno o la pareja nombre con apell, obligatoriamente uno de los dos, pero no ambas cosas; si tiene tfno no tendrá nombre ni apell; si tiene nombre y apell no tendrá tfno. Por supuesto, nombre y apell van en pareja (tienen paréntesis), por lo que no puede haber nombre sin apell, ni al revés.

Podría usarse el operador | para elegir uno entre tres o más elementos, del modo:

<!ELEMENT alumno (tfnocasa | tfnotrabajo | tfnomovil)>

El elemento alumno puede tener un teléfono, ya sea el de casa, el del trabajo o el móvil, pero no puede tener más de uno, ni menos; un teléfono de esos tres debe tener obligatoriamente, y sólo uno.

Si no se utiliza ningún operador, será obligatorio que el elemento aparezca una y sólo una vez. Por ejemplo:

<!ELEMENT alumno (nombre, apellidos)>

En ese ejemplo, si a un alumno no se le pone nombre dará error de validación; igualmente si no se pone apellidos también. El alumno debe tener obligatoriamente un elemento nombre y un elemento apellidos

Veamos un ejemplo completo de un .xml con su .dtd:

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE datos SYSTEM "fichero.dtd">
<datos>   <!-- Este es el ejemplar -->
<alumno>
  <nombre>Pedro</nombre>
  <apellidos>Garrido</apellidos>
  <direccion>Mayor, 20</direccion>
</alumno>
<alumno>
  <nombre>Ana</nombre>
  <apellidos>Pando</apellidos>
</alumno>
</datos>
```

fichero .dtd

```
<!ELEMENT datos (alumno+)>
<!ELEMENT alumno (nombre, apellidos, direccion?)>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT apellidos (#PCDATA)>
<!ELEMENT direccion (#PCDATA)>
```

Veamos otro ejemplo completo de un .xml con su .dtd:

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE agenda SYSTEM "fichero.dtd">
<agenda>
  <persona>
   <nombre>Ana</nombre>
   <apellidos>Cobos Ramos</apellidos>
   <direccion>
    <calle>Grande</calle>
    <numero>41</numero>
    <cp>28010</cp>
   </direccion>
  </persona>
</agenda>
```

fichero .dtd

```
<!ELEMENT agenda (persona+)>
        <!-- El elemento ejemplar agenda se forma con una o más personas -->
<!ELEMENT persona (nombre, apellidos, direccion)>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT apellidos (#PCDATA)>
```

```
<!ELEMENT direccion (calle, numero, cp)>
<!ELEMENT numero (#PCDATA)>
<!ELEMENT cp (#PCDATA)>
<!ELEMENT calle (#PCDATA)>
    <!-- Como vemos, la declaración del elemento calle puede ponerse después de numero
    y cp; el orden que importa es el establecido en la
    línea  <!ELEMENT direccion (calle, numero, cp)>    -->
```

**2.3. Declaraciones de atributos.**

Hasta ahora hemos hablado sobre los elementos, pero como sabemos en un documento XML algunos elementos pueden tener atributos para guardar algunos datos, por ejemplo la línea xml siguiente:

```
<alumno edad="20" nota="8">Juan Ramos</alumno>
```

El elemento alumno guarda su nombre en el propio elemento "alumno", pero su edad y su nota las guarda en los atributos "edad" y "nota". Como ya sabemos, el elemento alumno se pondría en el dtd del modo siguiente:

```
<!ELEMENT alumno (#PCDATA)>
```

El elemento alumno es elemento terminal, aunque tiene atributos. En el fichero .dtd hay que indicar los atributos, que se declaran con ATTLIST y CDATA en lugar de ELEMENT y #PCDATA. Para el ejemplo anterior sería:

```
<!ATTLIST alumno edad CDATA #IMPLIED>
<!ATTLIST alumno nota CDATA #IMPLIED>
```

Por tanto, la línea xml anterior quedaría en el dtd como sigue:

```
<!ELEMENT alumno (#PCDATA)>
<!ATTLIST alumno edad CDATA #IMPLIED>
<!ATTLIST alumno nota CDATA #IMPLIED>
```

Es decir, en la línea ATTLIST se debe escribir el elemento al que pertenece el atributo y después el atributo. También se podría utilizar la siguiente sintaxis, que es más corta ya que asocia a un elemento varios de sus atributos en una sola línea, quedando las dos líneas ATTLIST anteriores como sigue:

```
<!ATTLIST alumno edad CDATA #IMPLIED nota CDATA #IMPLIED>
```

Con CDATA se indica que el tipo del atributo es una cadena de caracteres (un texto). Esta cadena de texto de un atributo CDATA puede contener varias palabras, sin embargo con el tipo NMTOKEN obligas a que el atributo esté formado por una sola palabra. Para comprobar si el valor del atributo tiene varias palabras, el validador busca espacios en blanco, comas, etc., pero no los dos puntos ni el punto. Por ejemplo:

```
"hola adios"  el validador considera más de una palabra.
"hola;adios"  el validador considera más de una palabra.
"hola:adios"  el validador considera una palabra.
"hola.adios"  el validador considera una palabra.
```

Realmente lo que hace el tipo NMTOKEN es que sólo admite guardar en un atributo los siguientes caracteres:

- Letras (no admite el espacio en blanco): desde a hasta z y desde A hasta Z
- Dígitos: de 0 a 9
- El carácter punto .
- El carácter guión -
- El carácter subrayado o guión bajo _
- El carácter dos puntos :

Por tanto, el valor de un atributo de tipo NMTOKEN debe cumplir las reglas de XML para nombrar etiquetas. Además, con NMTOKEN se obliga a que el atributo tenga un valor, no se puede dejar vacío, del modo atributo="" (con las 2 comillas seguidas); si se deja vacío, al validar el .xml dará error. Sin embargo, con CDATA sí se puede dejar un atributo vacío.

Hemos indicado los tipos CDATA y NMTOKEN para los atributos, pero existe otro tipo, que es la enumeración (en el que se utiliza el operador | elección), donde se especifica una lista de valores, de forma que el atributo en el .xml deberá tener obligatoriamente uno de esos valores (se elige uno de los valores). Por ejemplo:

<!ATTLIST fecha dia (lunes|martes|miércoles|jueves|viernes|sábado|domingo) #REQUIRED>

El atributo dia del elemento fecha puede tomar uno de los siete valores indicados (como vemos, se escriben separados por el carácter | en el .dtd).

<!ATTLIST alumno nota (0|1|2|3|4|5|6|7|8|9|10) #IMPLIED>

El atributo nota del elemento alumno puede tomar uno de los 11 valores indicados y separados por el carácter |.

Otro tipo para los atributos es ID, que obliga a que el atributo tenga un valor único en el fichero .xml.

Además, las líneas de los atributos (ATTLIST) admiten los siguientes modificadores:

- Modificador #IMPLIED: el atributo es opcional; podría haber un elemento en el fichero .xml que no tenga el atributo en cuestión. Por ejemplo:

  En el .dtd: <!ATTLIST alumno edad NMTOKEN #IMPLIED nota NMTOKEN #IMPLIED>
  Los atributos edad y nota pueden no aparecer en un alumno en el .xml.

  En el .xml: <alumno>Juan Ramos</alumno>
  El elemento alumno no tiene edad ni nota, lo cual será validado correctamente, porque llevan el modificador #IMPLIED.

  En el .xml: <alumno edad="22">Juan Ramos</alumno>
  El elemento alumno tiene edad y no tiene nota, lo cual será validado correctamente, porque llevan el modificador #IMPLIED.

- Modificador #REQUIRED: el atributo es obligatorio, es decir, debe aparecer obligatoriamente en el .xml.

  En el .dtd: <!ATTLIST alumno edad NMTOKEN #REQUIRED nota NMTOKEN #IMPLIED>
  El atributo edad debe aparecer y el atributo nota puede no aparecer en un alumno en

el .xml.

En el .xml: <alumno edad="22">Juan Ramos</alumno>
El elemento alumno tiene edad y no tiene nota, lo cual será validado correctamente.

En el .xml: <alumno nota="9">Juan Ramos</alumno>
El elemento alumno no tiene edad, lo cual no será validado correctamente; la edad es obligatoria, por llevar #REQUIRED.

En el .xml: <alumno edad="">Juan Ramos</alumno>
El elemento alumno tiene el atributo edad (aunque con valor vacío), por lo que REQUIRED no dará errores de validación, sin embargo como tiene valor vacío no será validado correctamente, porque NMTOKEN no permite dejarlo vacío.

- Modificador #FIXED: el atributo debe valer obligatoriamente el valor indicado después de #FIXED. El uso del modificador #FIXED va seguido de un valor entre comillas, es decir se indica del siguiente modo:

  <!ATTLIST alumno grupo CDATA #FIXED "Primero">

  En ese ejemplo, el elemento alumno tiene el atributo grupo de tipo CDATA cuyo valor debe ser "Primero" siempre. Si en el fichero .xml se asigna al atributo grupo un valor distinto de "Primero", dará error al "Validar", por ejemplo:

  <alumno grupo="Segundo">  Esto daría error de validación.

  Es decir que con #FIXED se indica un valor que debe escribirse obligatoriamente en el fichero .xml en el atributo en cuestión. Además, si el elemento no lleva ese atributo, no dará error de validación y automáticamente se le añade el atributo con el valor indicado en el #FIXED; no se obliga a poner el atributo en el XML, pero si se pone deberá llevar obligatoriamente el valor indicado en #FIXED.

  En el fichero.dtd: <!ATTLIST alumno grupo CDATA #FIXED "Primero">

  En el fichero .xml: <alumno>
  No dará error de validación, además el atributo grupo será añadido con el valor "Primero".

  Para comprobar con el navegador web que ese atributo es añadido automáticamente deben ponerse las declaraciones DTD internas. Por ejemplo:

  ```
  <?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
  <!DOCTYPE alumno [
          <!ELEMENT alumno (#PCDATA)>
          <!ATTLIST alumno grupo CDATA #FIXED "Primero">
  ]>
  <alumno>
  </alumno>
  ```

  Si se abre ese documento .xml con un navegador web observaremos que al alumno se la añade el atributo grupo con el valor "Primero", aunque no esté escrito.

- Modificador "Literal": se indica el valor por defecto que toma el atributo cuando no se especifique en el .xml; si el atributo lleva un valor en el .xml se respeta, pero si no se pone atributo tomará el valor "Literal". Por ejemplo:

  En el fichero.dtd: <!ATTLIST alumno grupo NMTOKEN "Primero">

  En el fichero .xml: <alumno>
  Como no se especifica el atributo grupo, tomará el valor "Primero", por lo quedaría <alumno grupo="Primero">

  Para probar esto deben ponerse las declaraciones DTD internas en el fichero .xml (entre los corchetes) y abrir ese fichero .xml con el navegador web.

  Otro ejemplo:

  En el fichero .dtd: <!ATTLIST alumno edad NMTOKEN "18">

  En el fichero .xml: <alumno edad="20">
  En ese ejemplo, el atributo edad valdrá 20, ya que el valor 18 indicado en ATTLIST se utiliza sólo cuando en el .xml no se especifica el atributo, pero en este caso sí se ha especificado.

Veamos un ejemplo completo de un .xml con su .dtd, que valida correctamente (el .xml cumple todas las restricciones o normas incluidas en el .dtd):

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE alumnos SYSTEM "fichero.dtd">
<alumnos>
    <alumno especialidad="Ciencias">
      <nombre>Olga</nombre>
      <apellidos>Ramos</apellidos>
      <direccion tipo="Calle">Mayor 4</direccion>
      <contacto>
        <tel1>11111111</tel1>
        <tel2>22222222</tel2>
      </contacto>
      <ingles>nivel alto</ingles>
    </alumno>
     <alumno edad="16" especialidad="Letras">
      <nombre>Vicente</nombre>
      <apellidos>Parras</apellidos>
      <direccion>El Paseo 13</direccion>
      <contacto>
        <tel1>333333333</tel1>
        <tel2>444444444</tel2>
      </contacto>
      <correo>correo1@gmail.com</correo>
      <frances>nivel bajo</frances>
    </alumno>
     <alumno edad="14" especialidad="Ciencias">
      <nombre>Manuel</nombre>
```

```
    <apellidos>Correa</apellidos>
    <direccion>Villa 20</direccion>
    <contacto>
      <tel1>555555555</tel1>
      <tel2>666666666</tel2>
    </contacto>
    <correo>correo2@gmail.com</correo>
    <aleman>nivel medio</aleman>
   </alumno>
</alumnos>
```

fichero .dtd

```
<!ELEMENT alumnos (alumno+)>
<!ELEMENT alumno (nombre, apellidos, direccion, contacto,correo?,(ingles|frances|aleman))>
<!ATTLIST alumno edad NMTOKEN "10" especialidad (Ciencias|Letras) #IMPLIED>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT apellidos (#PCDATA)>
<!ELEMENT direccion (#PCDATA)>
<!ATTLIST direccion tipo CDATA #FIXED "Calle">
<!ELEMENT contacto (tel1, tel2)>
<!ELEMENT tel1 (#PCDATA)>
<!ELEMENT tel2 (#PCDATA)>
<!ELEMENT correo (#PCDATA)>
<!ELEMENT ingles (#PCDATA)>
<!ELEMENT frances (#PCDATA)>
<!ELEMENT aleman (#PCDATA)>
```

**2.4. Declaraciones de entidades.**

Las entidades nos permiten definir constantes en un documento XML.

Existen distintos tipos de entidades. Al usar una entidad en el fichero .xml, siempre se escribe con el símbolo & delante y con el símbolo ; (punto y coma) detrás.

Las entidades predefinidas (o integradas en el XML) se usan para escribir dentro de un dato en el fichero .xml uno de los caracteres siguientes: < > " ' &

Si se escribe en un .xml la línea

```
<empresa>Comer & Asociados</empresa>
```

el validador dará error, porque el carácter & tiene un significado especial para XML. Se debe escribir la entidad que corresponde al carácter &, que es:

```
<empresa>Comer  &amp;  Asociados</empresa>
```

De este modo no dará error y se visualizará el texto "Comer & Asociados".

Las entidades predefinidas o internas son:

| | |
|---|---|
| < | Se corresponde con el signo "menor que", < |
| > | Se corresponde con signo "mayor que", > |
| " | Se corresponde con las comillas rectas dobles, '' |

'  Se corresponde con el apóstrofe o comilla simple, '
&  Se corresponde con el et o ampersand, &

Como vemos, todas llevan & delante y ; detrás quedando: &Entidad;

Las entidades correspondientes a caracteres especiales (como el símbolo de euro €, el símbolo del copyright ©, etc. ) tienen un número asociado (código UNICODE, que puede verse en http://unicode-table.com), como:

€  Se corresponde con el euro €
©  Se corresponde con el copyright ©

En el fichero .xml pueden escribirse esas entidades del modo:

<autor>Pedro Contreras © </autor>
Se visualizará Pedro Contreras ©

Aparte de esas entidades predefinidas, también se pueden crear entidades propias, con ENTITY, para definir constantes en el .dtd, que se podrán utilizar en un fichero .xml, por ejemplo:

En el fichero .dtd:  <!ENTITY IVA "21">
(la constante IVA tiene el valor 21)

En el fichero .xml:  <impuesto> &IVA; </impuesto>

Se visualizará <impuesto> 21 </impuesto>

Para comprobar estas entidades con el navegador web, deben ponerse las declaraciones DTD internas (en el propio fichero .xml). Por ejemplo:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<!DOCTYPE ejemplar [
        <!ENTITY IVA "21">
        <!ELEMENT ejemplar (#PCDATA)>
]>
<ejemplar>
El iva es  &IVA;  por ciento.
</ejemplar>
```

Al abrir ese .xml con el navegador web, se mostrará:

<ejemplar>El iva es 21 por ciento.</ejemplar>

La ventaja de usar estas constantes radica en que si se desea cambiar su valor, sólo es necesario modificar una sola línea del fichero .dtd, la línea ENTITY, por lo que el fichero .xml no sería necesario modificarlo. Por ejemplo, cuando el IVA pase a ser 22, cambiamos el 21 por el 22 en la línea <!ENTITY IVA "22"> del .dtd, permaneciendo igual el .xml: <impuesto> &IVA; </impuesto>.

Se pueden utilizar estas entidades de constantes para los caracteres especiales indicados anteriormente (como el euro, el copyright, etc.), para utilizar en el .xml un nombre de entidad más legible. Por ejemplo:

En el fichero .dtd: <!ENTITY euro "€">

En el fichero .xml: <precio> 58 € </precio>
Visualizará: 58 €

En el fichero .dtd: <!ENTITY copyri "©">

En el fichero .xml: <autor> Pedro Contreras &copyri; </autor>
Visualizará: Pedro Contreras ©

Otro tipo de entidad es la entidad de parámetro interno (sólo para uso interno en el .dtd), que se utiliza para asignar un nombre a un trozo de código dentro del fichero .dtd. Si ese trozo de código se repite muchas veces dentro del fichero .dtd, será más rápido, cómodo y legible utilizar el nombre asignado (la entidad) en el lugar donde iría el trozo de código. Estas entidades no afectan al fichero .xml; tanto la definición como el uso están en el fichero .dtd. Por ejemplo:

```
<!ELEMENT residencia (nombre, calle, cp, ciudad, tfno)>
<!ELEMENT apartamento (nombre, calle, cp, ciudad, tfno)>
<!ELEMENT oficina (nombre, calle, cp, ciudad, tfno)>
<!ELEMENT tienda (nombre, calle, cp, ciudad, tfno)>
```

Como en ese .dtd hay un trozo de código que se repite varias veces, que es (nombre, calle, cp, ciudad, tfno), se puede definir una entidad para asignar un nombre a ese trozo de código, del modo siguiente (ahora se usa el % antes del nombre de la entidad):

```
<!ENTITY  % datos  "(nombre, calle, cp, ciudad, tfno)">
```

Le hemos dado el nombre "datos" a esa entidad. Debe destacarse que después del % y antes de datos debe ponerse un espacio en blanco; el trozo de código debe ponerse entre comillas.

Así el .dtd quedaría con menos código:

```
<!ENTITY  % datos  "(nombre, calle, cp, ciudad, tfno)">
<!ELEMENT residencia %datos;>
<!ELEMENT apartamento %datos;>
<!ELEMENT oficina %datos;>
<!ELEMENT tienda %datos;>
```

Como vemos en ese ejemplo, al definir la entidad con ENTITY se usa el símbolo % y un espacio antes del nombre de la entidad, que es % datos en este caso; sin embargo, al aplicar la entidad se pone el símbolo % pegado al nombre de la entidad y el símbolo ; (punto y coma) detrás, quedando %datos; en este caso.

La línea que define el ENTITY, es decir la línea
<!ENTITY  % datos  "(nombre, calle, cp, ciudad, tfno)">

puede colocarse en cualquier punto del fichero.dtd, pero siempre antes de utilizar dicha entidad, es decir antes de la línea <!ELEMENT residencia %datos;> en este caso.

No deben confundirse las entidades (ENTITY) con las notaciones (NOTATION). Mediante las notaciones se puede determinar la aplicación que se hará cargo de un fichero binario incluido en el DTD como entidad. Nosotros no usaremos notaciones.

Por otro lado, también existen las entidades (ENTITY) de parámetro externas, que permiten incluir un DTD (o varios DTD) en otro documento DTD.

**3. XML Schema.**

Los XML Schema (ficheros XSD), al igual que los DTD, también se utilizan para aplicar restricciones a los ficheros XML, de forma que no se permita escribir datos en los XML que no cumplan una serie de condiciones indicadas en el fichero XSD (en el Schema). El lenguaje XSD ofrece más posibilidades que el DTD en esa tarea, por ejemplo permite definir tipos de datos.

Usaremos también el programa "XML Copy Editor" para escribir los ficheros .xsd.

Si queremos asociar un fichero .xsd a un fichero .xml, es decir si queremos crear un documento XML que utiliza un fichero XSD, la estructura general del XML debe ser:

fichero.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<ejemplar xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                    xs:noNamespaceSchemaLocation="fichero.xsd">

   ... los distintos elementos del documento ...

</ejemplar>
```

Como vemos en esa estructura, en el fichero.xml debe indicarse el nombre del fichero XSD (en este caso fichero.xsd) que contiene las definiciones con un namespaces, que es siempre el mismo. Un fichero XSD tiene la estructura general siguiente:

fichero.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="ejemplar"  type="....." />

        ... más definiciones ...

</xs:schema>
```

En ese fichero .xsd se indica en name el nombre del ejemplar del fichero .xml; en type se indica el tipo de elemento, que se irá viendo a continuación.

Como vemos, un fichero .xsd también respeta la sintaxis XML: la primera línea es ?xml, tiene ejemplar (o elemento raíz) que es xs:schema, cada elemento tiene su apertura y cierre, etc. Por tanto, se le puede aplicar la opción "Comprobar Bien-Formado" del programa "XML Copy Editor" para ver si tiene errores sintácticos. La opción "Validar" en un fichero XSD no tiene sentido, pero un fichero XML que utilice XSD se valida con la misma opción vista para DTD (tecla <F5>).

Veamos un ejemplo muy sencillo de XML con XSD:

fichero.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<alumno xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xs:noNamespaceSchemaLocation="fichero.xsd">
    Pedro Contreras Pinto
</alumno>
```

fichero.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Elemento raíz del esquema-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!--Definición de un elemento simple llamado alumno que es una cadena de caracteres-->
  <xs:element name="alumno"  type="xs:string" />
</xs:schema>
```

Si el ejemplar tuviera algún atributo, se puede poner como en un XML normal. Por ejemplo, si el alumno anterior tuviera el atributo edad, sería:

fichero.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<alumno edad="22" xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xs:noNamespaceSchemaLocation="fichero.xsd">
    Pedro Contreras Pinto
</alumno>
```

Como vemos, se pone en la línea  <alumno edad="22"  xmlns:xs="etc etc....">

El fichero xsd para ese caso lo veremos más adelante.

**3.1. Tipos de datos base (predefinidos en XSD, built-in).**

El tipo de datos de un elemento indica el conjunto de valores que puede tomar ese elemento. Por ejemplo, si un elemento es de tipo decimal, podrá guardar números con decimales (como 9.65 u otro cualquiera); si un elemento es de tipo integer, podrá guardar números enteros (positivos o negativos pero sin decimales); si un elemento es de tipo string, podrá guardar cualquier cadena de caracteres

Como vemos en el apartado anterior, con xs:element se puede indicar el tipo de datos (type) que debe tener un elemento del fichero XML. El formato de xs:element es:

<xs:element name="Nombre del elemento del fichero XML"  type="...un tipo de datos...">

En ese ejemplo, en name hemos puesto "alumno", que es el elemento o etiqueta del XML (en este caso es el ejemplar o etiqueta raíz); en type se ha utilizado un tipo de los que ofrece XSD, como es xs:string. Estos tipos que ya vienen incluidos o predefinidos en XSD (built-in) se llaman tipos de datos base. Las opciones posibles para ese tipo (en type) son:

- string, se corresponde con una cadena de caracteres UNICODE (cadena de texto).

- boolean, un elemento de tipo boolean solo puede tomar 2 valores: true (cierto) o false (falso); por ejemplo:

En fichero .xsd: <xs:element name="MayorEdad"  type="xs:boolean" />
En fichero .xml: <MayorEdad>true</MayorEdad>  Correcto
En fichero .xml: <MayorEdad>false</MayorEdad>  Correcto
En fichero .xml: <MayorEdad>True</MayorEdad>  Incorrecto, la T debe ser t.
En fichero .xml: <MayorEdad>falso</MayorEdad>  Incorrecto, debe ser false.
En fichero .xml: <MayorEdad>cierto</MayorEdad>  Incorrecto, debe ser true.

- integer, número entero positivo o negativo.

- positiveInteger, número entero positivo; la I de Integer está en mayúsculas, es decir que deben respetarse las mayúsculas y minúsculas al escribir los códigos.

- negativeInteger, número entero negativo.

- decimal, número decimal; por ejemplo, 8.97. Como vemos, para separar los decimales debe utilizarse el punto decimal, no la coma; el valor decimal 8,97 dará error de validación.

- dateTime, un elemento de este tipo podrá guardar una fecha y hora absolutas, con el formato "YYYY-MM-DDThh:mm:ss", por ejemplo el valor "2014-01-23T14:57:38" cumple el tipo dateTime; si en un elemento de tipo dateTime escribimos el valor "2014-01-34T14:57:70" daría error al validar el fichero XML, ya que el día 34 no existe y los segundos 70 tampoco. También el validador comprueba los días que tiene cada mes e incluso si el año es bisiesto o no, por ejemplo dará error si ponemos el día 31 de abril (no será válido: 20140431T12:12:12) o el 29 de febrero de un año no bisiesto; será correcto 20120229T12:12:12, porque el 2012 fue año bisiesto (existe 29 de febrero).

- duration, un elemento de este tipo podrá guardar una duración de tiempo expresado en años, meses, días, horas, minutos segundos. El formato utilizado es: PnYnMnDTnHnMnS. Por ejemplo para representar una duración de 2 años, 4 meses, 3 días, 5 horas, 6 minutos y 10 segundos habría que poner: P2Y4M3DT5H6M7S. Se pueden omitir los valores nulos, luego una duración de 2 años será P2Y. Para indicar una duración negativa se pone un signo – precediendo a la P.

- time, hora en el formato "hh:mm:ss"; se comprueba que los valores sean correctos, es decir el validador dará error si se pone el minuto o segundo 61 y si se pone la hora 25 también.

- date, fecha en formato YYYY-MM-DD.

- gYearMonth, representa un mes de un año determinado mediante el formato YYYY-MM.

- gYear, indica un año gregoriano, el formato usado es YYYY.

- gMonthDay, representa un día de un mes mediante el formato --MM-DD (delante de las MM se ponen dos guiones); en el fichero .xml un dato de este tipo podría ser --08-15, es decir: 2 guiones, el mes con dos dígitos (entre 1 y 12) y el día con 2 dígitos (entre 1 y 31). Por supuesto, si MM es 02 (febrero), DD deberá estar entre 1 y 29; si MM es 04 (abril), DD estará entre 1 y 30; es decir, que el validador comprueba los días que tiene cada mes. Por ejemplo:

En fichero .xsd: <xs:element name="MesYDia"  type="xs:gMonthDay" />
En fichero .xml: <MesYDia>--03-29</MesYDia>  Será correcto; dará ".xml is valid".
Pero si:

En fichero .xml: <MesYDia>--13-29</MesYDia>  Será incorrecto (mes 13 no existe)

- gDay, indica el ordinal del día del mes mediante el formato ---DD, es decir deben ponerse tres guiones antes de los dos dígitos; por ejemplo, en el .xml el valor ---31 será correcto y ---32 incorrecto.

- gMonth, representa el mes mediante el formato --MM (con dos guiones delante). Por ejemplo, febrero es --02; el valor --2 dará error, porque deben utilizarse 2 dígitos.

- language, representa los identificadores de lenguaje, sus valores están definidos en RFC 1766; sirve para describir un lenguaje.

**3.2. Tipo de datos simple (simpleType).**

Aparte de esos tipos base (ya predefinidos en XSD, built-in), se pueden crear tipos propios, definidos por el programador. Un tipo de datos propio puede ser simple. Más adelante veremos que también existen los tipos complejos.

Un elemento de tipo simple (simpleType) no puede contener otros elementos ni atributos. Si un elemento contiene otros elementos y/o atributos, debe ser de tipo complejo, como veremos posteriormente.

Con un tipo simple (simpleType) sólo se puede indicar, a través de la etiqueta xs:restriction, el tipo base del dato (string, decimal, etc.) y las restricciones o características que debe cumplir.

El formato para declarar un tipo simple es el siguiente:

```
<xs:element name="alumno" type="tipo_propio"/>
<xs:simpleType name="tipo_propio">
    ... Aquí se indican las características de ese tipo_propio ...
</xs:simpleType>
```

Con ese formato, el tipo "tipo_propio" puede declararse en cualquier punto del fichero .xsd, no es obligatorio ponerlo debajo de la línea <xs:element name etc etc etc />, siempre que se respete la sintaxis de las estructuras xsd, es decir que ese tipo se declara fuera de cualquier otro tipo y elemento, como un bloque independiente. Se pueden declarar varios elementos seguidos y después declarar sus tipos; cada elemento queda cerrado con la barra / al final de cada línea. Por ejemplo:

```
<xs:element name="elem1" type="tipo1"/>
<xs:element name="elem2" type="tipo2"/>
<xs:element name="elem3" type="tipo3"/>
<xs:simpleType name="tipo1">
    ... Aquí se indican las características de tipo1 ...
</xs:simpleType>
<xs:simpleType name="tipo2">
    ... Aquí se indican las características de tipo2 ...
</xs:simpleType>
<xs:simpleType name="tipo3">
    ... Aquí se indican las características de tipo3 ...
</xs:simpleType>
```

Los nombres que se le dan a los tipos, en este caso "tipo1", "tipo2" y "tipo3", deben ser identificadores únicos, es decir que no deben coincidir con los identificadores o nombres que se utilicen para otros elementos. Si un elemento se llama por ejemplo "direccion", el tipo de ese elemento no debe llamarse también "direccion"; podría llamarse "tipo_direccion" o con otro nombre que no coincida con ningún otro elemento y ningún otro tipo. El programador debe inventar identificadores únicos.

Un segundo formato para declarar el tipo es el siguiente:

```
<xs:element name="alumno">
  <xs:simpleType>
      ... Aquí se indican las características que debe cumplir el elemento "alumno"...
  </xs:simpleType>
</xs:element>
```

Como vemos, con este segundo formato para definir un tipo no se le da nombre al tipo, sino que se declara un elemento ("alumno") y directamente se define su tipo; la etiqueta del elemento se cierra al final con </xs:element>, después de declarar el tipo. Se puede decir que el tipo se declara dentro del propio elemento, entre las etiquetas <xs:element ...> y </xs:element>. De este modo, el tipo debe escribirse justo después de la línea <xs:element ...> (no en otro sitio). Un inconveniente de este formato es que el tipo del elemento "alumno" no se puede utilizar para otros elementos que sean del mismo tipo; si otros elementos son de ese mismo tipo, debe repetirse el código si usamos esta notación. Con el primer formato visto para declarar un tipo, en el que se le da un nombre al tipo (en este caso es "tipo_propio"), puede aplicarse ese tipo a otros elementos, no sólo a "alumno". Por ejemplo, los siguientes elementos alumno y profesor tienen el mismo tipo de datos:

```
<xs:element name="alumno" type="tipo_propio"/>
<xs:element name="profesor" type="tipo_propio"/>
<xs:simpleType name="tipo_propio">
      ... Aquí se indican las características de ese tipo_propio ...
</xs:simpleType>
```

Es conveniente usar esa notación para definir el tipo, en la que se da un nombre al tipo (tipo_propio en ese caso) y se define en un bloque aparte del elemento (no dentro del elemento, sino fuera de cualquier otro bloque del programa). Debe tenerse en cuenta que en ese caso el tipo debe definirse fuera de todos los bloques del programa, como un bloque independiente (ver ejemplos posteriores). De ese modo, el programa queda mejor estructurado y más legible. Si se define el tipo dentro del propio elemento, al anidar varios elementos, se anidan sus tipos, de forma que resulta una estructura más compleja.

Las características de ese "tipo_propio" (el nombre lo elige el programador) se definen como se indica a continuación. Veamos un ejemplo:

fichero.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- "dato" es el ejemplar del fichero .xml  y de tipo "estado", que lo define el programador -->
<xs:element name="dato" type="estado"/>
```

```
<!-- El tipo "estado" es un tipo simple, que se define como un tipo base, xs:string en este caso (cadena de caracteres), al que se le aplican unas restricciones o condiciones que debe cumplir el dato que sea de este tipo; estas restricciones son que el dato debe tener como máximo 9 caracteres y sólo puede tener 2 valores: conectado u ocupado. -->
<xs:simpleType name="estado">
    <xs:restriction base="xs:string">
      <xs:maxLength value="9"/>
      <xs:enumeration value="conectado"/>
      <xs:enumeration value="ocupado"/>
    </xs:restriction>
</xs:simpleType>
</xs:schema>
```

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<!-- Si dato tomara un valor con una longitud superior a 9 o distinto de conectado u ocupado el .xml no sería válido, dando error de validación en XML Copy Editor  -->
<dato xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xs:noNamespaceSchemaLocation="fichero.xsd">conectado</dato>

<!-- El valor escrito en el .xml, "conectado" en este caso, debe aparecer pegado a las etiquetas, sin insertar fin de línea y retorno de carro (sin pulsar Intro), ya que implicaría una longitud superior a 9 y quedaría invalidado el .xml -->
```

Vemos en el ejemplo que en la etiqueta xs:restriction se indica el tipo de datos base de los que ofrece el XSD (string, boolean, integer, decimal, etc.); las restricciones incluidas en xs:restriction se aplican a un tipo base del XSD, es decir:

```
<xs:restriction base="xs:tipo_base">
        ...restricciones...
</xs:restriction>
```

El tipo_base puede ser string, integer, etc.

**3.3. Facetas de los tipos de datos.**

Como vemos en ese ejemplo, dentro del elemento xs:simpleType (dentro de sus etiquetas) se usa la etiqueta xs:restriction en la que se incluyen unos elementos que sirven para aplicar restricciones a los valores que pueden escribirse en el fichero .xml. Esas restricciones son las llamadas facetas. En este ejemplo las restricciones impuestas, que se escriben entre <xs:restriction> y <xs:/restriction>, son:

- Debe ser una cadena de caracteres (xs:string).
- El número máximo de caracteres será de 9, <xs:maxLength value="9"/>.
- Los valores posibles sólo son "conectado" y "ocupado".

Como se deduce del ejemplo, con el elemento xs:enumeration se puede crear una lista de valores posibles. En el fichero .xml sólo podrán aparecer los valores indicados en xs:enumeration.

Algunas facetas que se pueden utilizar son:

- length: longitud que debe tener el dato; si éste tiene otra longitud, el fichero .xml dará error de validación; por ejemplo:

```
<xs:element name="clave" type="tipo_clave" />

<xs:simpleType name="tipo_clave">
  <xs:restriction base="xs:string">
    <xs:length value="8"/>
  </xs:restriction>
</xs:simpleType>
```

El elemento "clave" debe ser una cadena de 8 caracteres. En el .xml el dato debe escribirse pegado a las etiquetas, del modo siguiente:

```
<clave>12345678</clave>        CORRECTO
```

Si se añaden líneas adicionales, del modo siguiente, dará error, ya que la longitud sería mayor de 8 (los saltos de línea cuentan como caracteres):

```
<clave>
12345678                INCORRECTO
</clave>
```

- minLength y maxLength (la L es mayúscula): longitud mínima y máximo que debe tener el dato; por ejemplo, si el elemento "codigo" es una cadena que puede tener entre 4 y 8 caracteres, se podría definir como sigue:

```
<xs:element name="codigo" type="tipo_cod" />

<xs:simpleType name="tipo_cod">
  <xs:restriction base="xs:string">
    <xs:minLength value="4"/>
    <xs:maxLength value="8"/>
  </xs:restriction>
</xs:simpleType>
```

- enumeration: se usa una etiqueta <xs:enumeration> por cada valor que puede tener el elemento.

```
<xs:element name="estadocivil" type="tipo_estadocivil" />

<xs:simpleType name="tipo_estadocivil">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Soltero"/>
    <xs:enumeration value="Casado"/>
    <xs:enumeration value="Viudo"/>
    <xs:enumeration value="Divorciado"/>
  </xs:restriction>
</xs:simpleType>
```

- whiteSpace (debe escribirse con la S mayúscula, ya que en caso contrario dará error sintáctico); los valores posibles para whiteSpace son: preserve (se respetan los espacios

escritos en el XML); replace (los saltos de línea, retornos de carro y tabuladores del XML se sustituyen por espacios); collapse (los espacios, saltos de línea, retornos de carro y tabuladores contiguos se quedan en un solo espacio en blanco; los aglutina todos en un espacio).

```
<xs:element name="frase" type="tipo_frase" />

<xs:simpleType name="tipo_frase">
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="collapse"/>
  </xs:restriction>
</xs:simpleType>
```

- maxInclusive, maxExclusive: indica el valor máximo permitido, incluyendo o excluyendo respectivamente ese valor.

```
<xs:element name="edad" type="tipo_edad" />

<xs:simpleType name="tipo_edad">
  <xs:restriction base="xs:integer">
        <xs:maxInclusive value="17"/>
            <!-- 17 será el valor máximo permitido, incluyendo el propio 17 -->
  </xs:restriction>
</xs:simpleType>
```

Por tanto, las dos líneas siguientes hacen lo mismo:

```
<xs:maxInclusive value="5"/>    Máximo el 5, incluido el 5.
<xs:maxExclusive value="6"/>    Máximo el 6, excluido el 6 (o sea, 5 será el máximo).
```

- minInclusive, minExclusive: indica el valor mínimo permitido, incluyendo (Inclusive) o excluyendo (Exclusive) respectivamente el propio valor.

```
<xs:minExclusive value="0"/>
0 será el valor mínimo permitido, excluyendo el propio 0, por lo que realmente el valor
mínimo permitido será el 1.
```

- totalDigits (la D es mayúscula): número total de dígitos permitidos; por ejemplo:

```
<xs:element name="edad" type="tipo_edad" />

<xs:simpleType name="tipo_edad">
  <xs:restriction base="xs:integer">
    <xs:totalDigits value="2"/>   <!-- Una edad de 100 o mayor no se admitirá -->
  </xs:restriction>
</xs:simpleType>
```

Si el número tiene decimales, el punto decimal no cuenta para totalDigits; por ejemplo, con un <xs:totalDigits value="4"/> el valor 10.54 (con el punto decimal son 5 caracteres) será admitido, sin embargo 108.54 no será admitido y dará error de validación.

- fractionDigits: número máximo de decimales permitidos; no es el número obligatorio de decimales, sino el número máximo, por lo que incluso un valor sin decimales en el XML sería permitido, estaría correctamente validado.

Ejemplo:

fichero .xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!--Elemento raíz -->
<xs:element name="nota" type="tipo_nota"/>

<xs:simpleType name="tipo_nota">
  <xs:restriction base="xs:integer">
    <xs:totalDigits value="2"/>
    <xs:minExclusive value="0"/>
    <xs:maxInclusive value="10"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!--
Creación del elemento "nota" de tipo "tipo_nota": dos dígitos cuyo valor es un número entero comprendido entre 1 y 10.
Sólo será un documento XML válido cuando la nota sea 1, 2, 3, 4, 5, 6, 7, 8, 9 ó 10 (el cero 0 no vale, ya que pone minExclusive). Cualquier otro número hará que este documento no sea válido.
-->
<nota xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xs:noNamespaceSchemaLocation="fichero.xsd">
10
</nota>
```

Otro ejemplo:

En este ejemplo, se permiten escribir en el fichero XML notas con decimales (xs:decimal) dentro del rango de 0 a 10 (ambos inclusive, el cero 0 también), con un máximo de 3 dígitos, de los cuales puede haber como máximo 2 decimales.

fichero .xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!--Elemento raíz -->
<xs:element name="nota" type="calificaciones"/>

<xs:simpleType name="calificaciones">
  <xs:restriction base="xs:decimal">
    <xs:totalDigits value="3"/>
    <xs:fractionDigits value="2"/>
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="10"/>
```

```
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<nota xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
            xs:noNamespaceSchemaLocation="fichero.xsd">
9.25
 <!-- La nota 10.5 no sería válida, ya que aunque tiene 3 dígitos, queda fuera del rango
    por ser mayor de 10.
    La nota 10.00 sí será válida, porque aunque tenga 4 dígitos, los ceros por la derecha en los
decimales no influyen (realmente el valor es 10)  -->
</nota>
```

- pattern: se utiliza para obligar a que los datos cumplan un patrón, es decir aplicar una máscara. Las opciones son:

| | |
|---|---|
| [A-Za-z] | Letra mayúscula o minúscula. |
| [A-Z] | Letra mayúscula. |
| [a-z] | Letra minúscula. |
| [0-9] | Dígitos decimales. |
| \D | Cualquier carácter excepto un dígito decimal. |
| (A) | Cadena que coincide con A. |
| A\|B | Cadena que es igual a la cadena A o a la B (el símbolo \| está en la tecla del número 1). |
| AB | Cadena que es la concatenación de las cadenas A y B. |
| A? | Cero o una vez la cadena A. |
| A+ | Una o más veces la cadena A. |
| A* | Cero o más veces la cadena A. |
| [abcd] | Alguno de los caracteres que están entre corchetes. |
| [^abcd] | Cualquier carácter que no esté entre corchetes. |
| \t | Tabulación. |

Por ejemplo, para obligar a cumplir el formato de un DNI sería:

fichero .xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!--Elemento raíz -->
<xs:element name="DNI" type="mascara"/>

<xs:simpleType name="mascara">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][A-Z]"/>
    <!--Este patrón indica que deben escribirse 8 dígitos de 0 a 9 cada uno y al final
       una letra mayúscula-->
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

```
<DNI xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xs:noNamespaceSchemaLocation="fichero.xsd">07222333C</DNI>
```

Debe tenerse en cuenta que si ponemos un salto de línea al final para poner la etiqueta </DNI> en la siguiente línea (el código xml quedaría más claro), del modo:

```
<DNI xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xs:noNamespaceSchemaLocation="fichero.xsd">07222333C
</DNI>
```

daría error de validación, ya que ese salto de línea no cumple el patrón indicado en pattern, queda fuera de la máscara; según el patrón antes de la etiqueta </DNI> debe haber una letra mayúscula, [A-Z], no un salto de línea. El salto de línea, aunque no se vea está ahí.

Un DNI del .xml que tenga sólo 7 dígitos debe escribirse con el 0 al principio, ya que el patrón utilizado obliga a tener 8 dígitos.

Ese patrón anterior se puede escribir de un modo más corto como sigue:

```
<xs:pattern value="[0-9]{8}[A-Z]"/>
```

Es decir, se pone entre paréntesis {...} el número de veces que debe aparecer el dígito entre 0 y 9. En este caso estamos diciendo que necesitamos 8 veces un dígito de 0 a 9 y después una letra mayúscula.

Si queremos admitir en el fichero .xml los DNI que tengan 7 dígitos más la letra, sin obligar a escribir el cero 0 al principio, sería con el patrón:

```
<xs:pattern value="[0-9]?[0-9][0-9][0-9][0-9][0-9][0-9][0-9][A-Z]"/>
```

o lo que es lo mismo

```
<xs:pattern value="[0-9]?[0-9]{7}[A-Z]"/>
```

Es decir, al poner el signo de interrogación ? estamos indicando que el primer dígito (que va de 0 a 9) puede aparecer una o cero veces. Así en el .xml el DNI "7222333C" sería validado correctamente (sin necesidad de rellenar con un cero 0 por la izquierda).

Debe recordarse que el tipo de dato para un DNI es xs:string, es decir cadena de caracteres, ya que un DNI, aunque tenga muchos dígitos, al llevar al final una letra no expresa una cantidad númerica, por lo que debe ser tratado como cadena (string). Suele ocurrir lo mismo con los teléfonos, los cuales aunque se forman con números, no indican cantidades numéricas, por lo que suelen ser string.

Con el patrón utilizado, la letra del DNI debe ser mayúscula [A-Z]; si queremos admitir mayúscula y minúscula, usaríamos el patrón:

```
<xs:pattern value="[0-9]?[0-9][0-9][0-9][0-9][0-9][0-9][0-9][A-Za-z]"/>
```

Otros ejemplos:

[XYZ] Este patrón indica que el carácter debe ser X o Y o Z.

[3-7XYZ] Este patrón indica que el carácter debe ser un dígito de 3 a 7 o una X o Y o Z.
[A-DZ] Este patrón indica que el carácter debe ser una letra de la A a la D o una Z.

Los corchetes indican que todo lo que se ponga dentro se utilizará para validar un solo carácter (o dígito) del dato a validar, es decir que se escoge un sólo carácter (no una cadena de 2 o más) de todos los que se incluyan en los corchetes. Por tanto:

[XYZ] Si el dato en el .xml vale "XY" dará error de validación, ya que ese patrón dice que el valor debe ser X o Y o Z; debe escogerse un solo carácter de los que estén entre corchetes.

Otros casos son:

<xs:pattern value="ABC"/> Obligamos a que el dato en cuestión valga obligatoriamente la cadena ABC (no se usan corchetes). Eso también puede indicarse poniendo la cadena entre paréntesis: <xs:pattern value="(ABC)">

<xs:pattern value="\D[0-9]"/> El dato debe comenzar por un carácter que no sea dígito (puede ser una letra mayúscula o minúscula, un signo de puntuación, la @, etc., pero no un dígito de 0 a 9) y después tendrá un dígito de 0 a 9. Debe notarse que la barra es invertida \ y la letra D es mayúscula.

<xs:pattern value="ABCD|1234"/> El dato debe valer la cadena ABCD o la cadena 1234; otros valores no validarán correctamente. Entre ABCD y el símbolo | no debe ponerse espacio en blanco; tampoco debe ponerse entre el símbolo | y 1234; si se pone algún espacio en blanco, significa que el espacio también será uno de los caracteres admitidos y validará correctamente en el xml.

<xs:pattern value="ABCD1234"/> El dato debe valer la cadena ABCD1234.

<xs:pattern value="ABCD?"/> El dato debe valer ABC o ABCD, es decir que la letra D puede aparecer una vez o ninguna, ya que lleva asociado el ? En este caso el ? sólo afecta a la D.

<xs:pattern value="(ABCD)?"/> El dato debe valer ABCD o nada, ya que el ? afecta a todo lo que esté entre los paréntesis ( ).

<xs:pattern value="ABCD+"/> El dato debe valer ABCD o ABCDD o ABCDDD, etc., ya que el + indica que la D puede aparece una vez o muchas.

<xs:pattern value="(ABCD)+"/> El dato debe valer ABCD o ABCDABCD o ABCDABCDABCD, etc., ya que el + se aplica al paréntesis completo, por lo que la cadena ABCD puede aparecer una vez o muchas.

En lugar del signo + se puede usar el asterisco * para indicar que la cadena puede aparecer cero veces o muchas (con el + obligas a que esté al menos una vez).

<xs:pattern value="[^xyz]"/> El dato debe ser cualquier carácter excepto la x, la y o la z. Para escribir ese símbolo ^ (a la derecha de la P en el teclado) debe pulsarse espacio en blanco una vez escrito dicho símbolo.

<xs:pattern value="[A-Z][^@]*"/> El dato debe tener un primer carácter que sea letra mayúscula y después admite cualquier cosa que no sea la @, incluso nada (ya que el * permite nada; si fuera + en lugar de * obliga a que haya al menos uno). Por tanto, los siguientes datos

sería válidos: A, A123abc, Z abcd123, BA. Sin embargo, los siguientes datos no validarían correctamente: a, A@, 1, b@, Z 123@ab.

**3.4. Elementos del lenguaje XSD.**

Ya se han estado utilizando distintos elementos del lenguaje XSD, como xs:schema, xs:element, xs:simpleType, xs:restriction, etc. Veamos ahora algunos otros.

Debemos saber que todos los elementos del fichero .xml deben estar definidos en el .xsd.

El fichero .xsd tendrá siempre una etiqueta xs:element en sus primeras líneas, del modo:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="alumnos" type="datosAlum"/>
        ...
</xs:schema>
```

En ese xs:element se indica el nombre del ejemplar del fichero .xml ("alumnos" en este caso) y el tipo de datos de ese ejemplar con type. Ese tipo puede ser un tipo base de XSD (string, decimal, etc.) o un tipo definido por el programador. Ese tipo definido puede ser simple (simpleType) o complejo (complexType).

En las etiquetas xs:element pueden utilizarse los atributos minOccurs y maxOccurs, para indicar el número mínimo y máximo de veces que puede aparecer el elemento en el fichero .xml. Por ejemplo:

```
<xs:element name="disco" type="tipodisco" minOccurs="0" maxOccurs="unbounded"/>
```

El valor "unbounded" significa "sin límite máximo", por lo que el fichero .xml puede contener muchos discos, sin limitar. Como el mínimo vale cero 0 (minOccurs="0"), el fichero .xml podría no tener ningún disco. Si se quiere obligar a que haya uno y sólo uno, ambos atributos se ponen con "1":

```
<xs:element name="profesor" type="tipoprofe" minOccurs="1" maxOccurs="1"/>
```

Pero realmente el valor por defecto para esos atributos minOccurs y maxOccurs es "1"; es decir, si no se especifica uno de ellos o ambos, es como si estuvieran escritos con el valor "1"; o sea que si se va a poner el valor "1", no es necesario escribirlo, así se ahorra código. Por ejemplo:

```
<xs:element name="profesor" type="tipoprofe" />
```

El elemento "profesor" debe aparecer en el fichero .xml una vez y sólo una. Por tanto, esa línea es equivalente a:

```
<xs:element name="profesor" type="tipoprofe" minOccurs="1" maxOccurs="1" />
```

Las dos líneas siguientes hacen lo mismo, el mínimo es 1 y el máximo es 5:

```
<xs:element name="profesor" type="tipoprofe" minOccurs="1" maxOccurs="5"/>
```

```
<xs:element name="profesor" type="tipoprofe" maxOccurs="5"/>
```

Lógicamente, para obligar a que un elemento aparezca en el fichero .xml un número concreto de veces, hay que poner ese número tanto en minOccurs como en maxOccurs. Por ejemplo, si queremos que haya 3 profesores (ni más ni menos) se pondría:

```
<xs:element name="profesor" type="tipoprofe" minOccurs="3" maxOccurs="3"/>
```

Como ya se ha indicado, un elemento de tipo simple (simpleType) no puede contener otros elementos ni atributos. Con un tipo simple (simpleType), a través de la etiqueta xs:restriction, sólo se puede indicar el tipo base del dato (string, decimal, etc.) y las restricciones que debe cumplir, como hemos estado haciendo en los ejemplos vistos hasta ahora. Es decir, un tipo simple es un tipo base que cumple unas restricciones.

**3.5. Tipo de datos complejo (complexType).**

Un elemento de tipo complejo (complexType) contiene otros elementos y/o atributos. Si un elemento de fichero .xml está formado por otros elementos, por ejemplo el elemento <agenda> puede estar formado por una lista de elementos <contacto>, se utilizará un tipo complejo (complexType) para agenda. También un elemento <direccion> puede estar formado por los elementos <calle>, <numero>, <poblacion>, <provincia>, etc., por lo que <direccion> también sería un elemento de tipo complejo. Por ejemplo:

En el .xml tenemos el siguiente elemento direccion:

```
<direccion>
        <calle>Mayor</calle>
        <numero>13</numero>
</direccion>
```

En el .xsd se pondría el tipo del elemento direccion del siguiente modo:

```
<xs:element name="direccion" type="TipoDireccion"/>

<xs:complexType name="TipoDireccion">
        <xs:sequence>
                <xs:element name="calle" type="xs:string"/>
                <xs:element name="numero" type="xs:positiveInteger"/>
        </xs:sequence>
</xs:complexType>
```

El nombre que se invente para el tipo, en este caso "TipoDireccion", debe ser identificador único, de forma que no coincida con otros nombres o identificadores de elementos o tipos.

Si un elemento tiene atributos, ese elemento debe ser declarado con tipo complejo (complexType). Para indicar el tipo de los atributos de un elemento se usa la etiqueta xs:attribute. Esta etiqueta xs:attribute debe incluirse dentro de la etiqueta complexType. Por ejemplo, si el elemento "persona" del XML es del tipo complejo "TipoContacto" y ese elemento "persona" tiene el atributo "tfno", escribiremos el xs:attribute dentro de la definición de ese tipo complejo, después de definir la secuencia de elementos que forman "persona", como sigue:

```
<xs:element name="persona" type="TipoContacto"/>

<xs:complexType name="TipoContacto">
	<xs:sequence>
		Secuencia de elementos ordenados que forman el elemento "persona".
	</xs:sequence>
	<xs:attribute name="tfno" type="xs:string"/>
</xs:complexType>
```

Con ese formato de declarar el tipo (en el que se le da un nombre al tipo, en este caso es TipoContacto), no es obligatorio poner la declaración del tipo justo debajo de la declaración del elemento, sino que el tipo podría declararse en cualquier punto del fichero .xsd, siempre que se respete la sintaxis de las estructuras xsd, es decir que el tipo debe declararse fuera de otros tipos y elementos, como un bloque independiente (como ya se ha indicado anteriormente para simpleType). Es decir, que podría ponerse algo como:

```
<xs:element name="persona" type="TipoContacto"/>
<xs:element name="edad" type="xs:positiveInteger"/>
<xs:element name="sueldo" type="xs:decimal"/>

<xs:complexType name="TipoContacto">
	<xs:sequence>
		Secuencia de elementos ordenados que forman el elemento "persona".
	</xs:sequence>
	<xs:attribute name="tfno" type="xs:string"/>
</xs:complexType>
```

En ese ejemplo, el TipoContacto no se declara debajo de la línea del elemento persona, lo cual es correcto.

Si un elemento tiene el tipo "Tipo01" (ya sea éste tipo simple o tipo complejo) y ese elemento forma parte de un tipo complexType, antes de definir el Tipo01 debe completarse la definición del complexType; el "Tipo01" se definirá en un bloque aparte, como:

```
<xs:element name="persona" type="TipoContacto"/>
<xs:complexType name="TipoContacto">
	<xs:sequence>
		<xs:element name="calle" type="Tipo01"/>
		<xs:element name="numero" type="xs:positiveInteger"/>
	</xs:sequence>
	<xs:attribute name="tfno" type="xs:string"/>
</xs:complexType>

<xs:complexType name="Tipo01">
	<xs:sequence>
		secuencia de elementos
	</xs:sequence>
</xs:complexType>
```

En ese código, el Tipo01 (ya sea simpleType o complexType) no puede declararse justo debajo del elemento calle, sino que antes debe completarse la definición de TipoContacto; el

Tipo01 se declara después, una vez cerrado el complexType. Por tanto, el siguiente código sería incorrecto:

```
<xs:element name="persona" type="TipoContacto"/>
<xs:complexType name="TipoContacto">
	<xs:sequence>
		<xs:element name="calle" type="Tipo01"/>
		<xs:complexType name="Tipo01">
			<xs:sequence>
				secuencia de elementos
			</xs:sequence>
		</xs:complexType>
		<xs:element name="numero" type="xs:positiveInteger"/>
	</xs:sequence>
	<xs:attribute name="tfno" type="xs:string"/>
</xs:complexType>
```

Ese código no es correcto, porque se define Tipo01 sin haber acabado de definir TipoContacto. Una vez completada la estructura del complexType de TipoContacto y cerrada con </xs:complexType>, después se puede declarar el Tipo01 (ya sea éste simple o complejo).

Como sabemos, existe otro formato para declarar el tipo, ya que se puede declarar directamente, sin ponerle nombre al tipo, escribiéndolo justo después de la línea <xs:element...> (no se puede poner en otro sitio), declarando el tipo dentro del propio elemento y cerrando el elemento "persona" al final con </xs:element> después de acabar la declaración del tipo, del modo:

```
<xs:element name="persona">
	<xs:complexType>
		<xs:sequence>
		    Secuencia de elementos ordenados que forman el elemento "persona".
		</xs:sequence>
		<xs:attribute name="tfno" type="xs:string"/>
	</xs:complexType>
</xs:element>
```

Como ocurre con los simpleType, en los complexType también es conveniente utilizar el método de declarar el tipo con su nombre, no declarado dentro del propio elemento, sino como un bloque aparte, independiente de todos los demás bloques, dentro sólo de xs:schema (pero fuera de otros elementos y tipos), de ese modo el fichero quedará más fácil de escribir, legible y estructurado. Además, el tipo en cuestión podría utilizarse para varios elementos, si fueran del mismo tipo. Por tanto, ese caso anterior quedaría mejor:

```
<xs:element name="persona" type="tipo_persona" />

<xs:complexType name="tipo_persona">
	<xs:sequence>
		Secuencia de elementos ordenados que forman el elemento "persona".
	</xs:sequence>
	<xs:attribute name="tfno" type="xs:string"/>
</xs:complexType>
```

Veamos ahora más sobre atributos. En ese ejemplo el atributo "tfno" no será obligatorio; es decir, si en el .xml se escribe un elemento "persona" sin el atributo "tfno" no dará error de validación, por lo tanto será correcto lo siguiente:

<persona>
elementos que forman el elemento "persona"
</persona>

CORRECTO, aunque no tiene atributo "tfno".

Para indicar que el atributo debe aparecer obligatoriamente en el xml se debe escribir en la línea xs:attribute la opción use="required", del modo:

```
<xs:attribute name="tfno" type="xs:string" use="required"/>
```

Con ese xs:attribute, el siguiente xml daría en la validación:

<persona>
elementos que forman el elemento "persona"
</persona>

INCORRECTO, porque no tiene atributo "tfno" y es obligatorio.

<persona tfno="924-11.11.11">
elementos que forman el elemento "persona"
</persona>

CORRECTO, tiene atributo "tfno" y debe tenerlo.

Hemos visto en los elementos anteriores que dentro de xs:sequence se ponen los elementos ordenados que componen otro elemento.

Si un elemento no contiene otros elementos, pero sí tiene atributos, no se pone xs:sequence dentro del complexType, pero se ponen los atributos con xs:attribute. Por ejemplo:

En el .xml tenemos: `<fecha_nacim dia="3" mes="8" anio="1975"/>`

En el .xsd se pondría:

```
<xs:element name="fecha_nacim" type="TipoFecha"/>

<xs:complexType name="TipoFecha">
        <xs:attribute name="dia" type="xs:positiveInteger"/>
        <xs:attribute name="mes" type="xs:positiveInteger"/>
        <xs:attribute name="anio" type="xs:positiveInteger"/>
</xs:complexType>
```

Como vemos, en la línea xs:attribute se indica el tipo del atributo con type. Ese tipo puede ser un tipo de datos base (string, integer, decimal, etc.); pero también un atributo puede ser de tipo simple (simpleType), si queremos que dicho atributo cumpla una serie de restricciones. Por ejemplo:

```
<xs:attribute name="tfno" type="TipoTelefono"/>

<xs:simpleType name="TipoTelefono">
        <xs:restriction base="xs:positiveInteger">
                <xs:length value="9"/>
```

```
        </xs:restriction>
    </xs:simpleType>
```

En ese ejemplo se obliga a que el valor del atributo teléfono (en el XML) sea un entero positivo de 9 dígitos.

El tipo del atributo, "TipoTelefono", debe declararse como un bloque aparte e independiente del resto de bloques del programa (fuera de cualquier otro elemento y tipo). Por ejemplo:

```
<xs:element name="persona" type="TipoPersona"/>

<xs:complexType name="TipoPersona">
    <xs:attribute name="codigo" type="TipoCodigo"/>
    <xs:attribute name="tfno" type="TipoTelefono"/>
</xs:complexType>

<xs:simpleType name="TipoCodigo">
    <xs:restriction base="xs:positiveInteger">
        <xs:length value="4"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TipoTelefono">
    <xs:restriction base="xs:positiveInteger">
        <xs:length value="9"/>
    </xs:restriction>
</xs:simpleType>
```

Por tanto, el tipo para un atributo se declara de la misma forma que el tipo para un elemento.

El ejemplar del fichero XML puede tener atributos. Veamos ese caso con un ejemplo, en el que el elemento ejemplar alumno tiene el atributo edad:

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<alumno edad="22" xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
            xs:noNamespaceSchemaLocation="fichero.xsd">
    <nombre>Antonio</nombre>
    <apellidos>Ramos Pinto</apellidos>
</alumno>
```

El fichero .xsd asociado será:

fichero .xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="alumno" type="TipoAlumno"/>

<xs:complexType name="TipoAlumno">
    <xs:sequence>
```

```
                <xs:element name="nombre" type="xs:string"/>
                <xs:element name="apellidos" type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="edad" type="xs:positiveInteger"/>
</xs:complexType>
```

Veamos ahora un ejemplo más extenso de un fichero .xml con su fichero .xsd asociado, donde se utilizan tipos complejos:

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!--
En este ejemplo sólo hay un alumno, pero podrían escribirse varios. El elemento "alumnos" (que es el ejemplar) está formado por una lista de elementos "alumno" y tiene el atributo cantidad.
-->
<alumnos cantidad="30" xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                    xs:noNamespaceSchemaLocation="fichero.xsd">
<alumno id="111">
  <nombre>Pedro</nombre>
  <apellidos>Contreras Ramos</apellidos>
  <edad>30</edad>

  <direccion>
   <domicilio>C/ Mayor, 21</domicilio>
   <codigo_postal cp="06010"/>
   <localidad>Villa Mayor</localidad>
   <provincia>Badajoz</provincia>
  </direccion>

  <contactar>
   <telf_casa>111111111</telf_casa>
   <telf_movil>222222222</telf_movil>
   <telf_trabajo>333333333</telf_trabajo>
   <email href="correo@gmail.com"/>
   <email href="otrocorreo@gmail.com"/>
  </contactar>
</alumno>
</alumnos>
```

Veamos el fichero.xsd asociado con las explicaciones aclaratorias incorporadas como comentarios (más adelante tenemos este fichero.xsd sin los comentarios):

fichero .xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="alumnos" type="datosAlum"/>
  <!-- En este xs:element se indica el ejemplar del fichero .xml ("alumnos") con su tipo -->

  <!-- Definición del nuevo "datosAlum", que es el tipo del ejemplar -->
  <xs:complexType name="datosAlum">
    <xs:sequence>
```

```
        <xs:element name="alumno" type="datos" maxOccurs="unbounded"/>
                <!--alumno aparecerá 1 o varias veces-->
    </xs:sequence>
    <xs:attribute name="cantidad" type="xs:positiveInteger"/>
                <!--El ejemplar alumnos tiene el atributo cantidad -->
  </xs:complexType>
```

<!-- Con ese tipo de datos complejo anterior indicamos que la secuencia de elementos (xs:sequence) que componen ese tipo es sólo un elemento, que es el elemento "alumno", cuyo tipo es "datos"; este elemento "alumno" puede aparecer en el fichero .xml al menos una vez (es como si estuviera puesto minOccurs="1") o muchas veces sin limitar (maxOccurs="unbounded"). También se indica que el tipo de datos para cada alumno se llama "datos", que se va a definir a continuación; en name debe indicarse el nombre del elemento usado en el fichero .xml, que en este caso es "alumno". Por otra parte, se incluye en ese tipo complejo el atributo cantidad del ejemplar alumnos -->

```
  <!-- Definicion del tipo "datos", que es el tipo de "alumno" -->
  <xs:complexType name="datos">
    <xs:sequence>
```

<!-- La secuencia de elementos (xs:sequence) que componen el elemento "alumno" es: nombre, apellidos, edad, direccion y contactar; si en el .xml se escribe antes el nombre que los apellidos dará error; con xs:sequence se obliga a respetar el orden de escritura del .xsd -->

```
        <xs:element name="nombre" type="xs:string"/>
        <xs:element name="apellidos" type="xs:string"/>
        <xs:element name="edad" type="tipoedad"/>
        <xs:element name="direccion" type="datosDireccion"/>
        <xs:element name="contactar" type="datosContactar"/>
      </xs:sequence>
```

<!-- Los elementos edad, direccion y contactar no son de tipo base, como string, integer, etc., sino que son de un tipo simple o complejo definido por el programador. El tipo para edad es "tipoedad", para direccion es "datosDireccion" y para contactar es "datosContactar".
Estos tipos están definidos más adelante, fuera de otros tipos y elementos -->

<!-- El atributo del elemento "alumno", llamado "id", debe definirse con xs:attribute, que debe colocarse fuera del xs:sequence, pero dentro de la etiqueta xs:complexType que define el tipo del alumno. Por eso se pone xs:attribute antes de cerrar la etiqueta xs:complexType, como se ve a continuación. -->

```
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>

  <xs:simpleType name="tipoedad">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="datosDireccion">
    <xs:sequence>
```

<!-- Como se usa xs:sequence, los elementos de la "direccion" deben escribirse en el .xml respetando el orden indicado aquí, en el .xsd: domicilio, codigo_postal, etc. -->

```
        <xs:element name="domicilio" type="xs:string" minOccurs="0"/>
        <xs:element name="codigo_postal" type="tipocodigop" minOccurs="0"/>
        <xs:element name="localidad" type="xs:string" minOccurs="0"/>
        <xs:element name="provincia" type="xs:string" minOccurs="0"/>
       </xs:sequence>
  </xs:complexType>

<!-- El tipo de codigo_postal se define a continuación, fuera de otros tipos y elementos. En este caso, codigo_postal no tiene otros elementos dentro, por lo que no tiene xs:sequence, sino que sólo lleva xs:attribute dentro de su tipo, para declarar el atributo cp.  -->
  <xs:complexType name="tipocodigop">
    <xs:attribute name="cp" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="datosContactar">
    <xs:sequence>
      <xs:element name="telf_casa" type="xs:string" minOccurs="0"/>
      <xs:element name="telf_movil" type="xs:string" minOccurs="0"/>
      <xs:element name="telf_trabajo" type="xs:string" minOccurs="0"/>
      <xs:element name="email" type="tipoem" minOccurs="0" maxOccurs="unbounded"/>
      <!-- Como vemos, email puede aparecer varias veces (unbounded) en el fichero .xml -->
    </xs:sequence>
  </xs:complexType>

 <xs:complexType name="tipoem">
   <xs:attribute name="href" type="xs:string"/>
 </xs:complexType>

</xs:schema>
```

Este fichero.xsd anterior sin los comentarios quedaría como sigue:

fichero .xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="alumnos" type="datosAlum"/>

  <xs:complexType name="datosAlum">
    <xs:sequence>
      <xs:element name="alumno" type="datos" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="cantidad" type="xs:positiveInteger"/>
  </xs:complexType>

  <xs:complexType name="datos">
    <xs:sequence>
      <xs:element name="nombre" type="xs:string"/>
      <xs:element name="apellidos" type="xs:string"/>
      <xs:element name="edad" type="tipoedad"/>
      <xs:element name="direccion" type="datosDireccion"/>
      <xs:element name="contactar" type="datosContactar"/>
     </xs:sequence>
```

```
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>

  <xs:simpleType name="tipoedad">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="datosDireccion">
    <xs:sequence>
      <xs:element name="domicilio" type="xs:string" minOccurs="0"/>
      <xs:element name="codigo_postal" type="tipocodigop" minOccurs="0"/>
      <xs:element name="localidad" type="xs:string" minOccurs="0"/>
      <xs:element name="provincia" type="xs:string" minOccurs="0"/>
     </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tipocodigop">
    <xs:attribute name="cp" type="xs:string"/>
  </xs:complexType>

  <xs:complexType name="datosContactar">
    <xs:sequence>
      <xs:element name="telf_casa" type="xs:string" minOccurs="0"/>
      <xs:element name="telf_movil" type="xs:string" minOccurs="0"/>
      <xs:element name="telf_trabajo" type="xs:string" minOccurs="0"/>
      <xs:element name="email" type="tipoem" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

 <xs:complexType name="tipoem">
   <xs:attribute name="href" type="xs:string"/>
 </xs:complexType>

</xs:schema>
```

Los elementos que forman parte de un elemento complejo pueden ser también complejos; no hay restricciones en ese sentido, cualquier elemento puede ser de tipo base, de tipo simple o de tipo complejo, aunque esté dentro de otro elemento. Eso puede verse en el ejemplo anterior, donde el elemento email es complejo y está dentro del elemento contactar, que también es complejo. Lo mismo ocurre con el elemento complejo codigo_postal, que está dentro del elemento direccion, que también es complejo. Otro ejemplo de esto puede ser:

En el .xml podemos tener:

```
<direccion ciudad="Zafra">
        <calle>
                <nombre>Mayor</nombre>
                <numero>21</numero>
```

```
        </calle>
        <cp>06310</cp>
</direccion>
```

En el .xsd podríamos tener:

```
<xs:element name="direccion" type="TipoDirec"/>
<xs:complexType name="TipoDirec">
        <xs:sequence>
                <xs:element name="calle" type="TipoCalle"/>
                <xs:element name="cp" type="TipoCp"/>
        </xs:sequence>
        <xs:attribute name="ciudad" type="TipoCiudad"/>
</xs:complexType>

<xs:complexType name="TipoCalle">
        <xs:sequence>
                <xs:element name="nombre" type="xs:string"/>
                <xs:element name="numero" type="xs:positiveInteger"/>
        </xs:sequence>
</xs:complexType>

<xs:simpleType name="TipoCp">
        <xs:restriction base="xs:string">
                <xs:length value="5"/>
        </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TipoCiudad">
        <xs:restriction base="xs:string">
                <xs:minLength value="3"/>
                <xs:maxLength value="20"/>
        </xs:restriction>
</xs:simpleType>
```

Si se define un tipo complejo sin darle nombre al mismo, es decir definiéndolo justo debajo del elemento, podría quedar un xs:sequence dentro de otro xs:sequence, lo cual es correcto y no da ningún tipo de problemas, pero se pierde legibilidad y queda el código peor estructurado. Por ejemplo:

```
<xs:element name="direccion">
    <xs:complexType>
        <xs:sequence>
                <xs:element name="calle" type="TipoCalle">
                   <xs:complexType>
                        <xs:sequence>
                                <xs:element name="nombre" type="xs:string"/>
                                <xs:element name="numero" type="xs:positiveInteger"/>
                        </xs:sequence>
                   </xs:complexType>
                </xs:element>
                <xs:element name="cp" type="xs:positiveInteger"/>
```

```
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Vemos en ese ejemplo que el elemento complejo "direccion" es una secuencia de los elementos "calle" y "cp"; a su vez el elemento complejo calle es una secuencia de los elementos "nombre" y "numero". Ha quedado una secuencia dentro de otra, pero es correcto.

Como ya se ha indicado, dentro de xs:sequence se escriben los elementos en un orden (es una lista ordenada), el cual debe respetarse en el fichero .xml. Sin embargo, si se utiliza xs:all en lugar de xs:sequence, se pueden escribir los elementos en el xml en distinto orden del indicado en el xs:all. Es decir, a través de xs:all se permite trabajar con secuencias o listas no ordenadas.

Los elementos dentro de xs:all deben tener minOccurs y maxOccurs entre 0 y 1, por lo que el valor "unbounded" dará error.

Por ejemplo, la lista de elementos (xs:all) que componen el elemento "direccion" pueden escribirse en el fichero .xml en un orden distinto del indicado en el siguiente fichero .xsd:

```
  <xs:complexType name="datosDireccion">
    <xs:all>
      <xs:element name="domicilio" type="xs:string" minOccurs="0" maxOccurs="1"/>
      <xs:element name="codigo_postal" type="tipocodp" minOccurs="0" maxOccurs="1"/>
      <xs:element name="localidad" type="xs:string" minOccurs="0" maxOccurs="1"/>
      <xs:element name="provincia" type="xs:string" minOccurs="0" maxOccurs="1"/>
     </xs:all>
  </xs:complexType>

  <xs:complexType name="tipocodp">
    <xs:attribute name="cp" type="xs:string"/>
  </xs:complexType>
```

Con ese código .xsd, en el fichero .xml se pueden escribir los datos en distinto orden del indicado en el .xsd, por ejemplo:

```
  <direccion>
   <codigo_postal cp="06010"/>
   <localidad>Villa Mayor</localidad>
   <provincia>Badajoz</provincia>
   <domicilio>C/ Mayor, 21</domicilio>
  </direccion>
```

Sabemos que maxOccurs="1" realmente no es necesario escribirlo, porque es el valor que toma por defecto.

Dentro de xs:all no pueden ponerse xs:sequence ni otras opciones (tampoco xs:choice, que veremos más adelante), sólo pueden ponerse elementos, con xs:element, y sus tipos, como se ve en ejemplo anterior.

También se puede utilizar xs:choice en lugar de xs:sequence o xs:all. Con xs:choice (elegir uno) se indica que debe elegirse uno y sólo uno de los tipos de elementos incluidos en el xs:choice. Por ejemplo:

```
<xs:element name="participante_curso" type="tipoparticipante" />

<xs:complexType name="tipoparticipante">
  <xs:choice>
   <xs:element name="profesor" type="tipoprofesor"/>
   <xs:element name="alumno" type="tipoalumno" maxOccurs="30"/>
  </xs:choice>
</xs:complexType>
```

Como vemos en ese ejemplo, el elemento complejo "participante_curso" estará formado por elementos de un solo tipo, que puede ser de tipo "tipoprofesor" o de tipo "tipoalumno", pero no por elementos de ambos tipos. Si el tipo es "tipoalumno", puede haber hasta un máximo de 30 elementos "alumno". O sea, en el .xml puede haber algo como:

```
<participante_curso>
        <profesor> ... </profesor>        ES CORRECTO
</participante_curso>

<participante_curso>
        <profesor> ... </profesor>        ES INCORRECTO; sólo puede haber un profesor.
        <profesor> ... </profesor>
</participante_curso>

<participante_curso>
        <profesor> ... </profesor>        ES INCORRECTO; o hay profesor o hay alumno, pero
        <alumno> ... </alumno>            no ambos mezclados.
</participante_curso>

<participante_curso>
        <alumno> ... </alumno>            ES CORRECTO; se permiten hasta 30 alumnos. Aunque
        <alumno> ... </alumno>            hay varios elementos, son todos del mismo tipo.
        <alumno> ... </alumno>
</participante_curso>
```

Otro ejemplo de xs:choice puede ser:

En el fichero .xsd podríamos tener

```
  <xs:complexType name="datosContactar">
    <xs:choice>
      <xs:element name="telf_casa" type="xs:string" minOccurs="0"/>
      <xs:element name="telf_movil" type="xs:string"/>
      <xs:element name="telf_trabajo" type="xs:string" maxOccurs="unbounded"/>
    </xs:choice>
  </xs:complexType>
```

En el fichero .xml podríamos tener

```
  <contactar>
   <telf_trabajo>333333333</telf_trabajo>
```

```
  <telf_trabajo>444444444</telf_trabajo>
 </contactar>
```

o también:

```
 <contactar>
  <telf_casa>111111111</telf_casa>
 </contactar>
```

o también:

```
 <contactar>
  <telf_movil>333333333</telf_movil>
 </contactar>
```

o también podríamos no tener ningún teléfono:

```
 <contactar>
 </contactar>
```

Es decir, en el fichero .xml dentro de la etiqueta <contactar> aparecerá sólo uno de los elementos escritos dentro del xs:choice del .xsd; pero telf_trabajo se admiten muchos, sin límite ("unbounded"). Aunque en este caso también podría no aparecer ninguno de los tres teléfonos, ya que el tfnocasa tiene minOccurs="0". Por tanto, debemos saber que si alguno de los elementos incluidos en el xs:choice tiene la opción minOccurs="0", puede que no aparezca en el xml ninguno de los elementos del xs:choice.

El xs:choice puede utilizarse dentro de xs:sequence. Por ejemplo, si una persona tiene varios elementos (nombre, edad, etc.) y entre ellos debe tener un teléfono, que puede ser el fijo o el móvil (sólo uno de los dos, nunca los dos), sería:

```
<xs:element name="persona" type="TipoPersona"/>
<xs:complexType name="TipoPersona">
	<xs:sequence>
		<xs:element name="nombre" type="xs:string"/>
		<xs:choice>
			<xs:element name="tfnofijo" type="xs:string"/>
			<xs:element name="tfnomovil" type="xs:string" minOcurrs="0"/>
		</xs:choice>
		<xs:element name="edad" type="xs:positiveInteger"/>
	</xs:sequence>
</xs:complexType>
```

En el xml, una persona tendrá un nombre, un tfnofijo o un tfnomovil o ninguno de los dos (nunca tendrá ambos) y finalmente una edad. Puede no tener ninguno de los dos teléfonos porque tenemos minOccurs="0" en uno de los elementos dentro del xs:choice.

También se puede dar la situación inversa, es decir tener un xs:sequence dentro de un xs:choice. Veamos un ejemplo:

```
<xs:element name="persona" type="TipoPersona"/>
<xs:complexType name="TipoPersona">
```

```
        <xs:sequence>
                <xs:element name="nombre" type="xs:string"/>
                <xs:choice>
                        <xs:element name="movil" type="xs:string"/>
                        <xs:sequence>
                                <xs:element name="fijocasa" type="xs:string"/>
                                <xs:element name="fijotrabajo" type="xs:string"/>
                        </xs:sequence>
                </xs:choice>
        </xs:sequence>
</xs:complexType>
```

Ese código quiere decir que la persona tiene un nombre; después la persona tiene movil o tiene la pareja fijocasa y fijotrabajo; si tiene movil, no puede tener dicha pareja; si tiene esa pareja, no puede tener movil.

El elemento xs:group tiene un uso muy diferente a los tres anteriores (xs:sequence, xs:all y xs:choice). Con xs:group simplemente se asigna un nombre a una agrupación de elementos (agrupación hecha con xs:sequence, xs:all o xs:choice), para después utilizar ese nombre en la definición de un tipo complejo complexType. Al asignarle ese nombre, se puede utilizar el mismo en distintos tipos complejos sin necesidad de repetir todo el código; por tanto, xs:group es simplemente una forma de ahorrar código. Por ejemplo:

```
<!-- Define un grupo formado por una secuencia de 4 elementos (Titulo, Editorial, Autor, Precio); a ese grupo le asigna el nombre "Libro" -->
<xs:group name="Libro">
 <xs:sequence>
  <xs:element name="Titulo" type="xs:string"/>
  <xs:element name="Editorial" type="xs:string"/>
  <xs:element name="Autor" type="xs:string"/>
  <xs:element name="Precio" type="xs:decimal"/>
 </xs:sequence>
</xs:group>

<!-- En la definición del siguiente tipo complejo se usa el grupo anterior poniendo la misma etiqueta xs:group pero con el atributo ref (en lugar de name); en esta etiqueta debe ponerse el nombre dado al grupo anteriormente con name, quedando ref="Libro", es decir:  -->
<xs:element name="Compra" type="TipoCompra"/>
<xs:complexType name="TipoCompra">
 <xs:group ref="Libro"/>
 <xs:attribute name="stock" type="xs:integer"/>
</xs:complexType>
<!-- Vemos en ese tipo complejo que además de incluir los 4 elementos del xs:group, se añade un atributo llamado "stock". Este atributo no estaba incluido en el grupo. Por tanto, el elemento "Compra" está formado por los elementos "Titulo", "Editorial", "Autor" y "Precio", además del atributo "stock".  -->

<!-- Aparte del elemento "Compra", en el xsd podríamos tener definir un elemento complejo que tuviera los mismos cuatro elementos de xs:group, sin incluir el citado atributo "stock".
Éste nuevo elemento se define como sigue: -->
<xs:element name="Venta" type="TipoVenta"/>
<xs:complexType name="TipoVenta">
```

```
 <xs:group ref="Libro"/>
</xs:complexType>
<!-- Este elemento "Venta" se compone de los elementos indicados en el group "Libro"; en este caso, "Venta" no tiene atributo "stock", que sí lo tenía "Compra".  -->
```

Con ese .xsd anterior, en el .xml podríamos tener algo como:

```
<!-- Una Compra tendrá el atributo "stock" y además los 4 elementos del xs:group -->
<Compra stock="9">
        <Titulo>Aquel</Titulo>
        <Editorial>Ed. Luna</Editorial>
        <Autor>Roberto Santana</Autor>
        <Precio>18.90</Precio>
</Compra>
<!-- Una Venta tendrá los 4 elementos del xs:group -->
<Venta>
        <Titulo>Cuestiones</Titulo>
        <Editorial>Edit. Sierra</Editorial>
        <Autor>Juan Pedros</Autor>
        <Precio>7.10</Precio>
</Venta>
```

Como vemos, el group "Libro" es simplemente un nombre (una referencia) que se ha dado a un grupo (sequence en este caso; podría ser all o choice) de 4 elementos. Ese nombre "Libro" puede ser utilizado para crear tipos complejos.

Si en el ejemplo anterior, el elemento "Compra" incluyera otro elemento (como el elemento "Nombre_Comprador"), además de los indicados en xs:group y del atributo "stock", su definición sería como sigue:

```
<xs:element name="Compra" type="TipoCompra"/>
<xs:complexType name="TipoCompra">
        <xs:sequence>
                <xs:group ref="Libro"/>
                 <xs:element name="Nombre_Comprador" type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="stock" type="xs:integer"/>
</xs:complexType>
```

Ahora el tipo "TipoCompra" lleva xs:sequence, dentro del cual se incluye el xs:group y el nuevo elemento, llamado "Nombre_Comprador". El xs:group tenía otro xs:sequence, por tanto queda un xs:sequence dentro de otro, lo cual no supone ningún tipo de problemas. Sin embargo, si el xs:group tuviera xs:all en lugar de xs:sequence, esa definición del tipo "TipoCompra" daría error, ya que queda un xs:all (el del xs:group) dentro de un xs:sequence, lo cual no es correcto (dentro de xs:sequence no puede haber xs:all).

Con ese .xsd anterior, en el xml un elemento "Compra" podría ser:

```
<Compra stock="9">
        <Titulo>Aquel</Titulo>
        <Editorial>Ed. Luna</Editorial>
        <Autor>Roberto Santana</Autor>
```

```
        <Precio>18.90</Precio>
        <Nombre_Comprador>Juan Pedros</Nombre_Comprador>
</Compra>
```

Debemos tener en cuenta que si varios elementos complejos se componen del mismo grupo o secuencia de elementos, no sería necesario utilizar xs:group, porque es más fácil definir un único tipo que se aplica a esos elementos complejos. Por ejemplo, si en el caso anterior "Compra" no tuviera atributo "stock" ni elemento "Nombre_Comprador", entonces "Compra" y "Venta" coincidirían en todo, estarían formados por la secuencia de 4 elementos "Titulo", "Editorial", "Autor" y "Precio"; en esas situaciones, en lugar de usar xs:group se haría lo siguiente:

```
<xs:element name="Compra" type="TipoCompraVenta"/>
<xs:element name="Venta" type="TipoCompraVenta"/>
<xs:complexType name="TipoCompraVenta">
        <xs:sequence>
                <xs:element name="Titulo" type="xs:string"/>
                <xs:element name="Editorial" type="xs:string"/>
                <xs:element name="Autor" type="xs:string"/>
                <xs:element name="Precio" type="xs:decimal"/>
        </xs:sequence>
</xs:complexType>
```

Como vemos, no se usa xs:group, sino que simplemente se define el tipo "TipoCompraVenta" que se usa tanto para el elemento "Compra" como para el elemento "Venta".

Por tanto, la ventaja de xs:group es que si se desea usar un mismo xs:group en varios tipos complejos distintos (deben ser distintos, que no coincidan en todo, si son iguales en todo no hace falta xs:group), basta con incluir la línea <xs:group ref="LaQueSea"/> en cada tipo complejo, por lo que se ahorra código.

Una vez vistas todas las opciones de tipos simples y complejos, vamos a exponer otro ejemplo con tipos complexType y simpleType:

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<alumnos xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                xs:noNamespaceSchemaLocation="fichero.xsd">
  <alumno id="001">
    <nombre>Alberto</nombre>
    <apellidos>Ramos Pinto</apellidos>
    <annos>14</annos>
  </alumno>

  <alumno id="002">
    <nombre>Ana</nombre>
    <apellidos>Prado Juan</apellidos>
    <annos>18</annos>
  </alumno>
</alumnos>
```

fichero .xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="alumnos" type="datosAlumnos"/>
  <!-- Definicion del tipo complejo "datosAlumnos": que está formado por elementos que
       también son de tipo complejo, el tipo "TipoAlumno" -->
  <xs:complexType name="datosAlumnos">
    <xs:sequence>
      <xs:element name="alumno" type="TipoAlumno" maxOccurs="unbounded"/>
       <!-- Un alumno o varios; al menos un alumno debe haber -->
    </xs:sequence>
  </xs:complexType>

  <!-- Definicion del tipo complejo "TipoAlumno": que está formado por unos elementos de
       tipo base del XSD (en este caso "string") y un elemento de tipo simple ("tipoedad").
       Además en este tipo complejo se define el tipo del atributo "id" -->
  <xs:complexType name="TipoAlumno">
    <xs:sequence>
      <xs:element name="nombre" type="xs:string" />
      <xs:element name="apellidos" type="xs:string" />
      <xs:element name="annos" type="tipoedad" />
     </xs:sequence>

    <!-- Atributo del elemento alumno: el tipo del atributo es un tipo base: string  -->
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>

<!-- Definición del tipo simple "tipoedad", que es un tipo base con restricciones, por lo que es
simpleType. En este caso el tipo base es positiveInteger y la restricción es obligar a que las
edades que se escriban en el .xml estén entre 12 y 18  -->
  <xs:simpleType name="tipoedad">
  <xs:restriction base="xs:positiveInteger">
    <xs:minInclusive value="12"/>
    <xs:maxInclusive value="18"/>
  </xs:restriction>
</xs:simpleType >
</xs:schema>
```

**3.6. Listas de valores.**

Con el elemento xs:list pueden crearse listas de valores, de forma que en el fichero .xml pueden escribirse una serie de valores separados por espacios. Por ejemplo:

```
<xs:simpleType name="lista_ingredientes">
 <xs:list itemType="xs:string"/>
</xs:simpleType>
```

Un elemento que sea de tipo "lista_ingredientes" podrá contener una lista de cadenas de caracteres separadas por espacios en blanco, como: "Agua Harina Aceite Azúcar".

En el atributo itemType puede indicarse un tipo base XSD (string, integer, etc.) o un tipo simple (simpleType).

**4. Herramientas de creación y validación.**

Existen muchas herramientas que facilitan el trabajo de edición y validación de los documentos XSD y XML.

Nosotros usaremos XML Copy Editor para crear y validar los documentos; también utilizaremos los navegadores web para mostrar los ficheros XML. A modo informativo, otros productos para estas tareas son:

- Editix XML Editor.
- XMLFox.
- Altova XML Spy Edición Estándar.
- Microsoft Core XML Services.
- Editor XML xmlBlueprint.
- Liquid XML Studio.
- Oxygen XML Editor.
- Exchanger XML Editor.

**5. Más ejemplos.**

Veamos algunos ejemplos completos:

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<alumnos xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                    xs:noNamespaceSchemaLocation="fichero.xsd">
<!-- El alumno tiene una serie de notas, las cuales se guardan todas en un solo elemento llamado <notas>; también el alumno tiene una serie de días de asistencia, que se guardan todos en el elemento <asiste>  -->
  <alumno id="001">
    <nombre>Marta Ramos</nombre>
    <notas>6 8 5 10</notas>
    <asiste>Lunes Martes Viernes</asiste>
  </alumno>
</alumnos>
```

fichero .xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="alumnos" type="datosAlum"/>

  <!-- Definicion del tipo "datosAlum", que es el tipo del elemento "alumnos" del fichero xml.
       Se indica simplemente que "alumnos" está formado por un elemento, "alumno", pero
       que puede aparecer muchas veces, "unbounded" -->
  <xs:complexType name="datosAlum">
    <xs:sequence>
      <xs:element name="alumno" type="datos" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <!-- Definicion del tipo "datos", que es el tipo del elemento "alumno" del fichero xml. Se indica que cada "alumno" está formado por un elemento nombre, un elemento notas y un elemento asiste; además tiene el atributo "id" -->
  <xs:complexType name="datos">
```

```
      <xs:sequence>
        <xs:element name="nombre" type="xs:string" />
        <xs:element name="notas" type="lista_notas" />
        <xs:element name="asiste" type="lista_dias" />
       </xs:sequence>
      <!-- Atributo "id"  del elemento alumno  -->
      <xs:attribute name="id" type="xs:string"/>
   </xs:complexType>

<!-- El elemento "notas" anterior es de tipo "lista_notas". Aquí se define ese tipo, de forma que
          se trata de una lista (xs:list) de valores enteros (xs:integer). Estos valores deben
          escribirse en el .xml separados por espacios. Como vemos con el atributo itemType se
          indica el tipo de los valores que se pueden escribir en el xml  -->
<xs:simpleType name="lista_notas">
  <xs:list itemType="xs:integer"/>
</xs:simpleType>

<!--  El elemento anterior llamado "asiste" tiene el tipo "lista_dias". Aquí se define este tipo,
          que también es una lista, pero la diferencia con la anterior es que los valores ahora no
          son de tipo base (integer, string, etc.), sino de un tipo simple propio (simpleType),
definido por el programador, llamado "dias_posibles" y que se define más abajo -->
<xs:simpleType name="lista_dias">
  <xs:list itemType="dias_posibles"/>
</xs:simpleType>
<!-- El tipo "dias_posibles" es un tipo simple; un elemento de este tipo sólo podrá valer
"Lunes", "Martes", "Miércoles", "Jueves" o "Viernes" -->
<xs:simpleType name="dias_posibles">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Lunes"/>
    <xs:enumeration value="Martes"/>
    <xs:enumeration value="Miércoles"/>
    <xs:enumeration value="Jueves"/>
    <xs:enumeration value="Viernes"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

Veamos otro ejemplo que incluye el elemento "web", que no guarda ningún dato, sino que la información se guarda en el atributo "href" de dicho elemento:

fichero .xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<datos_alumno clase="5" xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
                        xs:noNamespaceSchemaLocation="fichero.xsd">
<!--  El ejemplar es datos_alumno, que tiene el atributo tipo y los elementos: -->
<nombre>Juan</nombre>
<apellidos>Ramos Ruz</apellidos>
<web href="http://www.pagina1.com"/>
<web href="http://www.pagina2.com"/>
<web href="http://www.pagina3.com"/>
<web href="http://www.pagina4.com"/>
<web href="http://www.pagina5.com"/>
</datos_alumno>
```

fichero .xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- Para definir el tipo del ejemplar datos_alumno usamos el tipo complejo "tipoalumno" -->
<xs:element name="datos_alumno" type="tipoalumno"/>
 <xs:complexType name="tipoalumno">
  <xs:sequence>
     <xs:element name="nombre" type="xs:string" />
     <xs:element name="apellidos" type="xs:string" />
     <xs:element name="web" type="tipoweb" minOccurs="0" maxOccurs="5" />
         <!-- El elemento web puede aparecer en el fichero .xml desde 0 veces hasta 5 veces.-->
  </xs:sequence>
  <xs:attribute name="clase" type="xs:positiveInteger"/>
         <!-- El ejemplar datos_alumno tiene el atributo clase.-->
 </xs:complexType>

 <xs:complexType name="tipoweb">
       <xs:attribute name="href" type="xs:string" />
 </xs:complexType>

</xs:schema>
```

Si ponemos en ese.xml un alumno con 6 páginas webs, dará error de validación.