Tratamiento de XML en Android

Existen distintas posibilidades a la hora de trabajar con datos en formato XML desde la plataforma Android.

A día de hoy, en casi todas las grandes plataformas de desarrollo existen varias formas de leer y escribir datos en formato XML. Los dos modelos más extendidos son **SAX** (*Simple API for XML*) y **DOM** (*Document Object Model*). Posteriormente, han ido apareciendo otros tantos, con más o menos éxito, entre los que destaca **StAX** (*Streaming API for XML*). Pues bien, Android no se queda atrás en este sentido e incluye estos tres modelos principales para el tratamiento de XML, o para ser más exactos, los dos primeros como tal y una versión análoga del tercero (***XmlPull***). Por supuesto con cualquiera de los tres modelos podemos hacer las mismas tareas, pero ya veremos cómo dependiendo de la naturaleza de la tarea que queramos realizar va a resultar más eficiente utilizar un modelo u otro.

Resumiendo:

1.- Si trabajamos con **SAX** tendremos que recorrer todo el fichero. Lo utilizaremos cuando el fichero ocupe mucho espacio y haya que recorrer todo el fichero.

2.- Si trabajamos con **DOM** cargaremos la información en memoria y podremos acceder a la información sin volverla a leer completamente. Es recomendable que el fichero no sea grande.

3.- Trabajaremos con **PULL** cuando nos interese acceder a parte de la información. Por ejemplo los 30 primeros items.

Antes de empezar, unas anotaciones respecto a los ejemplos que vamos a utilizar. Estas técnicas se pueden utilizar para tratar cualquier documento XML, tanto online como local, pero por utilizar algo conocido por la mayoría, todos los ejemplos van a trabajar sobre los datos XML de un documento RSS online, y ahora utilizaremos como ejemplo el canal RSS de portada de *europapress.com.*

Un documento RSS de este *feed* tiene la estructura siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
    <channel>
        <title>Portada</title>
        <link>https://www.europapress.es/</link>
        <description>Portada</description>
        <image>
            <url>https://s01.europapress.net/imagenes/estaticos/eplogo.gif</url>
            <title>Portada</title>
            <link>https://www.europapress.es/</link>
        </image>
        <language>es-ES</language>
        <copyright>(C) 2020 Europa Press.</copyright>
        <pubDate>Fri, 10 Jan 2020 18:01:41 GMT</pubDate>
        <lastBuildDate>Fri, 10 Jan 2020 17:31:12 GMT</lastBuildDate>
        <atom:link href="www.europapress.es" rel="self"
                   type="application/rss+xml" />
```

```
        <ttl>2</ttl>
        <item>
            <title> Titulo de la nocicia 1 </title>
            <link>https:// link_de_la_noticia1.es </link>
            <description> Descripción de la noticia 1. </description>
            <enclosure url="https://link_imagen_relacionada_con_la_noticia1.jpg"
                       length="4096" type="image/jpeg" />
            <guid isPermaLink="true">https://identificador_noticia1.html</guid>
            <pubDate> Fecha Publicacion de la noticia 1</pubDate>
            <category>Categoria de la noticia 1</category>
        </item>
        <item>
            <title> Titulo de la nocicia 2 </title>
            <link>https:// link_de_la_noticia2.es </link>
            <description> Descripción de la noticia 2. </description>
            <enclosure url="https://link_imagen_relacionada_con_la_noticia1.jpg"
                       length="4096" type="image/jpeg" />
            <guid isPermaLink="true">https://identificador_noticia2.html</guid>
            <pubDate> Fecha Publicacion de la noticia 1</pubDate>
            <category>Categoria de la noticia 2</category>
        </item>
        . . . .
        <item>
            . . .
        </item>
    </channel>
</rss>
```

Como puede observarse, se compone de un elemento principal `<channel>` seguido de varios datos relativos al canal y posteriormente una lista de elementos `<item>` para cada noticia con sus datos asociados.

Veamos cómo leer este XML mediante cada una de las tres alternativas citadas, y para ello lo primero que vamos a hacer es definir una **clase java** para almacenar los datos de una noticia (`Noticia.java`). Nuestro objetivo final será devolver **una lista de objetos de este tipo**, con la información de todas las noticias. Por comodidad, vamos a almacenar todos los datos como cadenas de texto:

```
public class Noticia {
    private String titulo;
    private String link;
    private String descripcion;
    private String guid;
    private String fecha;

    //Definimos los getter y los setter
    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
```

```
    }

    public String getLink() {
        return link;
    }

    public void setLink(String link) {
        this.link = link;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public String getGuid() {
        return guid;
    }

    public void setGuid(String guid) {
        this.guid = guid;
    }

    public String getFecha() {
        return fecha;
    }

    public void setFecha(String fecha) {
        this.fecha = fecha;
    }
}
```

Una vez conocemos la estructura del XML a leer y definida la clase auxiliar que nos hace falta para almacenar los datos, pasamos a comentar el primero de los modelos de tratamiento de XML.

1.- SAX en Android

En el modelo SAX, el tratamiento de un XML se basa en un analizador (*parser*) que a medida que lee secuencialmente el documento XML va generando diferentes eventos con la información de cada elemento leido. Asi, por ejemplo, a medida que lee el XML, si encuentra el comienzo de una etiqueta, por ejemplo `<title>`, generará un evento de comienzo de etiqueta, `startElement()`, con su información asociada, si después de esa etiqueta encuentra un fragmento de texto generará un evento `characters()` con toda la información necesaria, y así sucesivamente hasta el final del documento. Nuestro trabajo consistirá por tanto en implementar las acciones necesarias a ejecutar para cada uno de los eventos posibles que se pueden generar durante la lectura del documento XML.

Los principales eventos que se pueden producir son los siguientes (consultar aquí la lista completa):

- **`startDocument()`:** comienza el documento XML.
- **`endDocument()`:** termina el documento XML.
- **`startElement()`:** comienza una etiqueta XML.
- **`endElement()`:** termina una etiqueta XML.
- **`characters()`:** fragmento de texto.

Todos estos métodos están definidos en la clase `org.xml.sax.helpers.DefaultHandler`, de la cual deberemos derivar una clase propia donde se sobrescriban los eventos necesarios.

En nuestro caso vamos a llamarla **`RssHandler`** **(**`RssHandler.java`**)**

```
public class RssHandler extends DefaultHandler {

    private List<Noticia> noticias; //Lista de noticias
    private Noticia noticiaActual;  //Noticia Actual
    private StringBuilder sbTexto;  //Texto del elemento

    public List<Noticia> getNoticias() {
        return noticias;
    }

    @Override
    public void startDocument() throws SAXException {
        super.startDocument();

        noticias = new ArrayList<Noticia>();
        sbTexto = new StringBuilder();
    }

    @Override
    public void startElement(String uri, String localName, String qName,
                        Attributes attributes) throws SAXException {
        super.startElement(uri, localName, qName, attributes);

        if (localName.equals("item")){
            noticiaActual = new Noticia();
        }
    }

    @Override
    public void characters(char[] ch, int start, int length)
                                                throws SAXException {
        super.characters(ch, start, length);

        if (this.noticiaActual != null)
            sbTexto.append(ch,start,length);
    }

    @Override
    public void endElement(String uri, String localName,
                           String qName) throws SAXException {
        super.endElement(uri, localName, qName);
```

```
            if (this.noticiaActual != null) {
                if (localName.equals("title")){
                    noticiaActual.setTitulo(sbTexto.toString());
                } else if (localName.equals("link")){
                    noticiaActual.setLink(sbTexto.toString());
                } else if (localName.equals("description")){
                    noticiaActual.setDescripcion(sbTexto.toString());
                } else if (localName.equals("guid")){
                    noticiaActual.setGuid(sbTexto.toString());
                } else if (localName.equals("pubDate")){
                    noticiaActual.setFecha(sbTexto.toString());
                } else if (localName.equals("item")){
                    noticias.add(noticiaActual);
                }

                sbTexto.setLength(0);
            }
        }
    }
```

Como se puede observar en el código anterior, lo primero que haremos será incluir como miembro de la clase la lista de noticias que pretendemos construir, `List<Noticia> noticias`, y un método `getNoticias()` que permita obtenerla tras la lectura completa del documento. **Tras esto, implementamos directamente los eventos SAX necesarios.**

Comencemos por `startDocument()`, este evento indica que se ha comenzado a leer el documento XML, por lo que lo aprovecharemos para inicializar la lista de noticias y las variables auxiliares.

Tras éste, el evento `startElement()` se lanza cada vez que se encuentra una nueva etiqueta de apertura. En nuestro caso, la única etiqueta que nos interesará será `<item>`, momento en el que inicializaremos un nuevo objeto auxiliar de tipo `Noticia` donde almacenaremos posteriormente los datos de la noticia actual.

El siguiente evento relevante es `characters()`, que se lanza cada vez que se encuentra un fragmento de texto en el interior de una etiqueta. La técnica aquí será ir acumulando en una variable auxiliar, `sbTexto`, todos los fragmentos de texto que encontremos hasta que se detecte una etiqueta de cierre.

Por último, en el evento de cierre de etiqueta, `endElement()`, lo que haremos será almacenar en el atributo apropiado del objeto `noticiaActual` (que conoceremos por el parámetro `localName` devuelto por el evento) el texto que hemos ido acumulando en la variable `sbTexto` y limpiaremos el contenido de dicha variable para comenzar a acumular el siguiente dato. El único caso especial será cuando detectemos el cierre de la etiqueta `<item>`, que significará que hemos terminado de leer todos los datos de la noticia y por tanto aprovecharemos para añadir la noticia actual a la lista de noticias que estamos construyendo.

Una vez implementado nuestro *handler*, vamos a crear una nueva clase que haga uso de él para parsear mediante SAX un documento XML concreto. A esta clase la llamaremos **`RssParserSax`** (`RssParserSax.java`). Más adelante crearemos otras clases análogas a ésta que hagan lo mismo pero utilizando los otros dos métodos de tratamiento de XML ya mencionados. Esta clase tendrá únicamente un constructor que reciba como parámetro la URL del documento a parsear,

y un método público llamado `parse()` para ejecutar la lectura del documento, y que devolverá como resultado una lista de noticias. Veamos cómo queda esta clase:

```
public class RssParserSAX {

    private URL rssUrl;

    public RssParserSAX(String url) {
        try {
            this.rssUrl = new URL (url);
        } catch (MalformedURLException e) {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse() {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        try {
            SAXParser parser = factory.newSAXParser();
            RssHandler handler = new RssHandler();
            parser.parse(this.getInputStream(), handler);
            return handler.getNoticias();
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    private InputStream getInputStream (){
        try {
            return rssUrl.openConnection().getInputStream();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Como se puede observar en el código anterior, el constructor de la clase se limitará a aceptar como **parámetro la URL** del documento XML a parsear y controlar la validez de dicha URL, generando una excepción en caso contrario.

Por su parte, el método `parse()` será el encargado de crear un nuevo parser SAX mediante su *fábrica* correspondiente (lo que se consigue obteniendo una instancia de la fábrica con `SAXParserFactory.newInstance()` y creando un nuevo parser con `factory.newSaxParser()`) y de iniciar el proceso pasando al parser una instancia del *handler* que hemos creado anteriormente y una referencia al documento a parsear en forma de *stream*.

Para esto último, nos apoyamos en un método privado auxiliar `getInputStream()`, que se encarga de abrir la conexión con la URL especificada (mediante `openConnection()`) y obtener el *stream* de entrada (mediante `getInputStream()`).

Con esto ya tenemos nuestra aplicación Android preparada para parsear un documento XML online utilizando el modelo SAX. Veamos lo simple que sería ahora llamar a este parser por ejemplo desde nuestra actividad principal. Como ejemplo de tratamiento de los datos obtenidos mostraremos los titulares de las noticias en un cuadro de texto (`txtResultado`):

```
public class MainActivity extends AppCompatActivity {

    public String url = "https://www.europapress.es/rss/rss.aspx";

    private TextView txtResultado;
    private List<Noticia> noticias;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        txtResultado = (TextView)findViewById(R.id.txtResultado);
    }

    //Con la propiedad onClick en los botones
    public void cargarXMLConSAX (View v){
        //Sin Tarea Asincrona
        RssParserSAX saxparser = new RssParserSAX(url);
        List<Noticia> noticias= saxparser.parse();

        //Tratamos la lista de noticias
        //Por ejemplo: Escribimos los titulos en pantalla
        txtResultado.setText("");
        for (int i=0; i<=noticias.size(); i++) {
            txtResultado.setText(txtResultado.getText().toString() +
                    System.getProperty("line.separator") +
                    noticias.get(i).getTitulo());
        }
    }

}
```

Las líneas marcadas del código anterior son las que hacen toda la "*magia*". Primero creamos el parser SAX pasándole la URL del documento XML y posteriormente llamamos al método `parse()` para obtener una lista de objetos de tipo `Noticia` que posteriormente podremos manipular de la forma que queramos. Así de sencillo.

Adicionalmente, para que este ejemplo funcione debemos añadir previamente permisos de acceso a internet para la aplicación. Esto se hace en el fichero `AndroidManifest.xml`, que quedaría de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.lecturaxml">

    <uses-permission
        android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
```

```
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

En la linea marcada del código podéis ver cómo añadimos el permiso de acceso a la red mediante el elemento `<uses-permission>` con el parámetro `android.permission.INTERNET`

Pero hay algo más, si ejecutamos el código anterior en una versión de Android anterior a la 3.0 no tendremos ningún problema. La aplicación descargará el XML y lo parseará tal como hemos definido. Sin embargo, si utilizamos una versión de Android 3.0 o superior nos encontraremos con algo similar a esto:

Y si miramos el logcat de ejecución veremos que se ha generado una excepción del tipo `NetworkOnMainThreadException`. ¿Qué ha ocurrido? Pues bien, lo que ha ocurrido es que desde la versión 3 de Android se ha establecido una política que no permite a las aplicaciones ejecutar operaciones de larga duración en el hilo principal que puedan bloquear temporalmente la interfaz de usuario. En este caso, nosotros hemos intentado hacer una conexión a la red para descargarnos el XML y Android se ha quejado de ello generando un error.

¿Y cómo arreglamos esto? Pues la solución pasa por mover toda la lógica de descarga y lectura del XML a otro hilo de ejecución secundario, es decir, hacer el trabajo duro en segundo plano dejando libre el hilo principal de la aplicación y por tanto sin afectar a la interfaz. La forma más sencilla de hacer esto en Android es mediante la utilización de las llamadas `AsyncTask` o tareas asíncronas.

Lo que haremos será definir una nueva clase interna que extienda de `AsyncTask` y sobrescribiremos sus métodos `doInBackground()` y `onPostExecute()`. Al primero de ellos moveremos la descarga y parseo del XML, y al segundo las tareas que queremos realizar cuando haya finalizado el primero, en nuestro caso de ejemplo la escritura de los titulares al cuadro de texto de resultado. Quedaría de la siguiente forma:

```
//Tarea Asíncrona para cargar un XML en segundo plano
private class CargarXmlTask extends AsyncTask<String,Integer,Boolean> {

    protected Boolean doInBackground(String... params) {
                RssParserSAX saxparser = new RssParserSAX(params[0]);
```

```
                noticias = saxparser.parse();
            return true;
        }

        protected void onPostExecute(Boolean result) {
            //Tratamos la lista de noticias
            //Por ejemplo: escribimos los títulos en pantalla
            txtResultado.setText("");
            if (noticias !=  null) {
                for (int i = 0; i < noticias.size(); i++) {
                    txtResultado.setText(
                            txtResultado.getText().toString() +
                            System.getProperty("line.separator") +
                            noticias.get(i).getTitulo());
                }
            }
        }

    }
```

Por último, sustituiremos el código de nuestra actividad principal por una simple llamada para crear y ejecutar la tarea en segundo plano:

```
    public class MainActivity extends AppCompatActivity {

        public String url = "https://www.europapress.es/rss/rss.aspx";

        private TextView txtResultado;
        private List<Noticia> noticias;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);

            txtResultado = (TextView)findViewById(R.id.txtResultado);
        }

        public void cargarXMLConSAX (View v){
            //Carga de XML mediante Tarea Asincrona
            CargarXmlTask tarea = new CargarXmlTask();
            tarea.execute(url);
        }

    . . . .

    }
```

Y ahora sí, con esta ligera modificación del código nuestra aplicación se ejecutará correctamente en cualquier versión de Android.

2.- SAX Simplificado

En el ejemplo anterior hemos visto cómo realizar la lectura y tratamiento de un documento XML utilizando el modelo SAX clásico. Hemos visto cómo implementar un *handler* SAX, donde se definían las acciones a realizar tras recibirse cada uno de los posibles eventos generados por el *parser* XML.

Este modelo, a pesar de funcionar perfectamente y de forma bastante eficiente, tiene claras desventajas. Por un lado se hace necesario definir una clase independiente para el *handler*. Adicionalmente, la naturaleza del modelo SAX implica la necesidad de poner bastante atención a la hora de definir dicho *handler*, ya que los eventos SAX definidos no están ligados de ninguna forma a etiquetas concretas del documento XML sino que se lanzarán para todas ellas, algo que obliga entre otras cosas a realizar la distinción entre etiquetas dentro de cada evento y a realizar otros chequeos adicionales.

Estos problemas se pueden observar perfectamente en el evento `endElement()`. En primer lugar teníamos que comprobar la condición de que el atributo `noticiaActual` no fuera `null`, para evitar confundir el elemento `<title>` descendiente de `<channel>` con el del mismo nombre pero descendiente de `<item>`. Posteriormente, teníamos que distinguir con un **`if`** gigantesco entre todas las etiquetas posibles para realizar una acción u otra. Y todo esto para un documento XML bastante sencillo. No es difícil darse cuenta de que para un documento XML algo más elaborado la complejidad del handler podría dispararse rápidamente, dando lugar a posibles errores.

Para evitar estos problemas, Android propone una variante del modelo SAX que evita definir una clase separada para el handler y que permite asociar directamente las acciones a etiquetas concretas dentro de la estructura del documento XML, lo que alivia en gran medida los inconvenientes mencionados.

Veamos cómo queda el nuevo parser XML utilizando esta variante simplificada de SAX para Android a la que llamaremos **`RssParserSAXSimplificado`** (`RssParserSAXSimplificado.java`) y después comentaremos los aspectos más importantes del mismo.

```
public class RssParserSAXSimplificado {

    private URL rssUrl;
    private Noticia noticiaActual;

    public RssParserSAXSimplificado(String url) {
        try {
            this.rssUrl = new URL (url);
        } catch (MalformedURLException e) {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse(){

        final List <Noticia> noticias = new ArrayList<Noticia>();

        RootElement root = new RootElement("rss");
        Element channel = root.getChild("channel");
        Element item = channel.getChild("item");
```

```
item.setStartElementListener(new StartElementListener() {
    @Override
    public void start(Attributes attributes) {
        noticiaActual = new Noticia();
    }
});

item.setEndElementListener(new EndElementListener() {
    @Override
    public void end() {
        noticias.add(noticiaActual);
    }
});

item.getChild("title").setEndTextElementListener(
        new EndTextElementListener() {
            @Override
            public void end(String body) {
                noticiaActual.setTitulo(body);
            }
        });

item.getChild("link").setEndTextElementListener(
        new EndTextElementListener() {
            @Override
            public void end(String body) {
                noticiaActual.setLink(body);
            }
        });

item.getChild("description").setEndTextElementListener(
        new EndTextElementListener() {
    @Override
    public void end(String body) {
        noticiaActual.setDescripcion(body);
    }
});

item.getChild("guid").setEndTextElementListener(
        new EndTextElementListener() {
    @Override
    public void end(String body) {
        noticiaActual.setGuid(body);
    }
});

item.getChild("pubDate").setEndTextElementListener(
         new EndTextElementListener() {
    @Override
    public void end(String body) {
        noticiaActual.setFecha(body);
    }
});
```

```
            try {
                Xml.parse(this.getInputStream(),
                          Xml.Encoding.UTF_8,
                          root.getContentHandler());
            }catch (Exception e){
                throw new RuntimeException(e);
            }

            return noticias;
        }

        private InputStream getInputStream() {
            try {
                return rssUrl.openConnection().getInputStream();
            }catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
```

Debemos atender principalmente al método `parse()`. En el modelo SAX clásico nos limitamos a instanciar al *handler* definido en una clase independiente y llamar al correspondiente método `parse()` de SAX. Por el contrario, en este nuevo modelo SAX simplificado de Android, las acciones a realizar para cada evento las vamos a definir en esta misma clase y además asociadas a etiquetas concretas del XML. Y para ello lo primero que haremos será *navegar* por la estructura del XML hasta llegar a las etiquetas que nos interesa tratar y una vez allí, asignarle algunos de los *listeners* disponibles (de apertura (`StartElementListener`**)** o cierre (`EndElementListener`) de etiqueta) incluyendo las acciones oportunas. De esta forma, para el elemento `<item>` navegaremos hasta él obteniendo en primer lugar el elemento raíz del XML (`<rss>`) declarando un nuevo objeto `RootElement` y después accederemos a su elemento hijo `<channel>` y a su vez a su elemento hijo `<item>`, utilizando en cada paso el método `getChild()`. Una vez hemos llegado a la etiqueta deseada, asignaremos los *listeners* necesarios, en nuestro caso uno de apertura de etiqueta y otro de cierre, donde inicializaremos la noticia actual y la añadiremos a la lista final respectivamente, de forma análoga a lo que hacíamos para el modelo SAX clásico. Para el resto de etiquetas actuaremos de la misma forma, accediendo a ellas con `getChild()` y asignado los listeners necesarios.

Finalmente, iniciaremos el proceso de parsing simplemente llamando al método `parse()` definido en la clase `android.util.Xml`, al que pasaremos como parámetros el stream de entrada, la codificación del documento XML y un handler SAX obtenido directamente del objeto `RootElement` definido anteriormente.

Si queremos ejecutar nuestro parser sin errores en cualquier versión de Android (sobre todo a partir de la 3.0) deberemos hacerlo mediante una tarea asíncrona.

Como vemos, este modelo SAX alternativo simplifica la elaboración del handler necesario y puede ayudar a evitar posibles errores en el handler y disminuir la complejidad del mismo para casos en los que el documento XML no sea tan sencillo como el utilizado para el ejemplos. Por supuesto, el modelo clásico es tan válido y eficiente como éste, por lo que la elección entre ambos es cuestión de gustos.

```
public class MainActivity extends AppCompatActivity {

    public String url = "https://www.europapress.es/rss/rss.aspx";

    private TextView txtResultado;
    private List<Noticia> noticias;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        txtResultado = (TextView)findViewById(R.id.txtResultado);
    }


    public void cargarConSAXSimplificado(View v){
        //Carga del XML mediante tarea Asincrona
        CargarXmlTask tarea = new CargarXmlTask();
        tarea.execute(url);
    }


    //Tarea Asíncrona para cargar un XML en segundo plano
    private class CargarXmlTask extends AsyncTask<String,Integer,Boolean> {

        protected Boolean doInBackground(String... params) {
                  RssParserSAXSimplificado saxparserSimplificado =
                            new RssParserSAXSimplificado(params[0]);
                  noticias = saxparserSimplificado.parse();
            return true;
        }

        protected void onPostExecute(Boolean result) {
            //Tratamos la lista de noticias
            //Por ejemplo: escribimos los títulos en pantalla
            txtResultado.setText("");
            if (noticias !=  null) {
                for (int i = 0; i < noticias.size(); i++) {
                    txtResultado.setText(
                            txtResultado.getText().toString() +
                            System.getProperty("line.separator") +
                                    noticias.get(i).getTitulo());
                }
            }
        }
    }

}
```

3.- Tratamiento de XML DOM

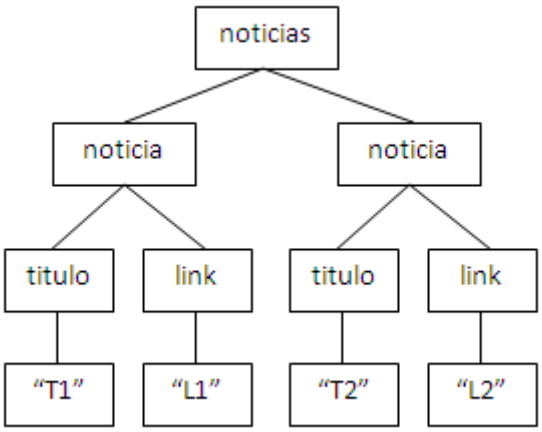
Ahora, vamos a centrarnos en DOM, otro de los métodos clásicos para la lectura y tratamiento de XML.

Cuando comentábamos la filosofía de SAX vimos cómo con dicho modelo el tratamiento del fichero XML se realizaba de forma secuencial, es decir, se iban realizando las acciones necesarias durante la propia lectura del documento. Sin embargo, con DOM la estrategia cambia radicalmente. Con *DOM*, *el documento XML se lee completamente antes de poder realizar ninguna acción en función de su contenido*. Esto es posible gracias a que, como resultado de la lectura del documento, *el parser DOM devuelve todo su contenido en forma de una estructura de tipo árbol*, donde los distintos elementos del XML se representa en forma de nodos y su jerarquía padre-hijo se establece mediante relaciones entre dichos nodos.

Como ejemplo, vemos un fichero XML sencillo y cómo quedaría su representación en forma de árbol:

```
<noticias>
   <noticia>
       <titulo>T1</titulo>
       <link>L1</link>
   </noticia>
   <noticia>
       <titulo>T2</titulo>
       <link>L2</link>
   </noticia>
<noticias>
```

Este XML se traduciría en un árbol parecido al siguiente:



Como vemos, este árbol conserva la misma información contenida en el fichero XML pero en forma de nodos y transiciones entre nodos, de forma que se puede navegar fácilmente por la estructura. Además, este árbol **se conserva persistente en memoria** una vez leído el documento completo, lo que permite procesarlo en cualquier orden y tantas veces como sea necesario (a diferencia de SAX, donde el tratamiento es secuencial y siempre de principio a fin del documento, no pudiendo volver atrás una vez finalizada la lectura del XML).

Para todo esto, el modelo DOM ofrece una serie de clases y métodos que permiten almacenar la información de la forma descrita y facilitan la navegación y el tratamiento de la estructura creada.

Veamos cómo quedaría nuestro parser utilizando el modelo DOM a cuya clase llamaremos **RssParserDOM** (RssParserDOM.java) y comentaremos los detalles más importantes.

```
public class RssParserDOM {

    private URL rssURL;

    public RssParserDOM(String url){
        try{
            this.rssURL =new URL (url);
        }catch (MalformedURLException e) {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse() {
        //Instanciamos la fabrica para DOM
        DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();
        List<Noticia> noticias = new ArrayList<Noticia>();

        try {
            //Creamos un nuevo parser DOM
            DocumentBuilder builder = factory.newDocumentBuilder();

            //Realizamos la lectura completa del XML
            Document dom = builder.parse(this.getInputStream());

            //Nos posicionamos en el nodo principal del árbol (<rss>)
            Element root = dom.getDocumentElement();

            //Localizamos todos los elemntos <item>
            NodeList items = root.getElementsByTagName("item");

            //Recorremos la lista de noticias
            for (int i=0; i<items.getLength(); i++){
                Noticia noticia = new Noticia();

                //Obtenemos la noticia actual
                Node item = items.item(i);

                //Obtenemos la lista de datos de la noticia actual
                NodeList datosNoticia = item.getChildNodes();

                //Procesamos cada dato de la noticia
                for (int j=0; j<datosNoticia.getLength(); j++){
                    Node dato = datosNoticia.item(j);
                    String etiqueta = dato.getNodeName();
```

```
                    if (etiqueta.equals("title")){
                        String texto = obtenerTexto(dato);
                        noticia.setTitulo(texto);
                    }
                    else if (etiqueta.equals("link")){
                        String texto = dato.getFirstChild().getNodeValue();
                        noticia.setLink(texto);
                        //noticia.setLink(dato.getFirstChild().getNodeValue());
                    }
                    else if (etiqueta.equals("guid")) {
                        noticia.setGuid(
                                dato.getFirstChild().getNodeValue());
                    }
                    else if (etiqueta.equals("pubDate")) {
                        noticia.setFecha(
                                dato.getFirstChild().getNodeValue());
                    }
                }
                noticias.add(noticia);
            }

        } catch (ParserConfigurationException e) {
            e.printStackTrace();
        }
        catch (Exception ex){
            throw new RuntimeException(ex);
        }

        return noticias;
    }


    public String obtenerTexto (Node dato) {
        StringBuilder texto = new StringBuilder();
        NodeList fragmentos = dato.getChildNodes();

        for (int k=0; k<fragmentos.getLength(); k++) {
            texto.append(fragmentos.item(k).getNodeValue());
        }
        return texto.toString();
    }


    private InputStream getInputStream() {
        try {
            return rssURL.openConnection().getInputStream();
        }
        catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Nos centramos una vez más en el método `parse()`. Al igual que hacíamos para SAX, el primer paso será instanciar una nueva fábrica, esta vez de tipo `DocumentBuilderFactory`, y posteriormente crear un nuevo parser a partir de ella mediante el método `newDocumentBuilder()`.

Tras esto, ya podemos realizar la lectura del documento XML llamando al método `parse()` de nuestro parser DOM, pasándole como parámetro el *stream* de entrada del fichero. Al hacer esto, el documento XML se leerá completo y se generará la estructura de árbol equivalente, que se devolverá como un objeto de tipo `Document`. Éste será el objeto que podremos navegar para realizar el tratamiento necesario del XML.

Para ello, lo primero será acceder al nodo principal del árbol (en nuestro caso, la etiqueta `<rss>`) utilizando el método `getDocumentElement()`. Una vez posicionados en dicho nodo, vamos a buscar todos los nodos cuya etiqueta sea `<item>`. Esto lo conseguimos utilizando el método de búsqueda por nombre de etiqueta, `getElementsByTagName("nombre_de_etiqueta")`, que devolverá una lista (de tipo `NodeList`) con todos los nodos hijos del nodo actual cuya etiqueta coincida con la pasada como parámetro.

Una vez tenemos localizados todos los elementos `<item>`, que representan a cada noticia, los vamos a recorrer uno a uno para ir generando todos los objetos `Noticia` necesarios. Para cada uno de ellos, se obtendrán los nodos hijos del elemento mediante `getChildNodes()` y se recorrerán éstos obteniendo su texto y almacenándolo en el atributo correspondiente del objeto `Noticia`. Para saber a qué etiqueta corresponde cada nodo hijo utilizamos el método `getNodeName()`.

Merece la pena pararnos un poco en comentar la forma de obtener el texto contenido en un nodo. Como hemos visto anteriormente en el ejemplo gráfico de árbol DOM, el texto de un nodo determinado se almacena a su vez como nodo hijo de dicho nodo. Este nodo de texto suele ser único, por lo que la forma habitual de obtener el texto de un nodo es obtener su primer nodo hijo y de éste último obtener su valor:

```
String texto = nodo.getFirstChild().getNodeValue();
```

Sin embargo, en ocasiones, el texto contenido en el nodo viene fragmentado en varios nodos hijos, en vez de sólo uno. Esto ocurre por ejemplo cuando se utilizan en el texto entidades HTML, como por ejemplo `&quot;` . En estas ocasiones, para obtener el texto completo hay que recorrer todos los nodos hijos e ir concatenando el texto de cada uno para formar el texto completo. Esto es lo que hace nuestra función auxiliar `obtenerTexto()`:

```
public String obtenerTexto (Node dato) {
    StringBuilder texto = new StringBuilder();
    NodeList fragmentos = dato.getChildNodes();

    for (int k=0; k<fragmentos.getLength(); k++) {
        texto.append(fragmentos.item(k).getNodeValue());
    }
    return texto.toString();
}
```

Como vemos, el modelo DOM nos permite localizar y tratar determinados elementos concretos del documento XML, sin la necesidad de recorrer todo su contenido de principio a fin. Además, a diferencia de SAX, como tenemos cargado en memoria el documento completo de forma

persistente (en forma de objeto `Document`), podremos consultar, recorrer y tratar el documento tantas veces como sea necesario sin necesidad de volverlo a parsear.

```
public class MainActivity extends AppCompatActivity {

    public String url = "https://www.europapress.es/rss/rss.aspx";

    private TextView txtResultado;
    private List<Noticia> noticias;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        txtResultado = (TextView)findViewById(R.id.txtResultado);
    }

    public void cargarXMLConDOM(View v){
        //Carga del XML mediante tarea Asincrona
        CargarXmlTask tarea = new CargarXmlTask();
        tarea.execute(url);
    }


    //Tarea Asíncrona para cargar un XML en segundo plano
    private class CargarXmlTask extends AsyncTask<String,Integer,Boolean> {

        protected Boolean doInBackground(String... params) {
                 RssParserDOM domParser = new RssParserDOM(params[0]);
                 noticias =domParser.parse();
            return true;
        }

        protected void onPostExecute(Boolean result) {
            //Tratamos la lista de noticias
            //Por ejemplo: escribimos los títulos en pantalla
            txtResultado.setText("");
            if (noticias !=  null) {
                for (int i = 0; i < noticias.size(); i++) {
                    txtResultado.setText(
                            txtResultado.getText().toString() +
                            System.getProperty("line.separator") +
                            noticias.get(i).getTitulo());
                }
            }
        }
    }
}
```

4.- Tratamiento de XML XmlPull

Por último estudiaremos el método menos conocido, aunque igual de válido según el contexto de la aplicación, llamado XmlPull. Este método es una versión similar al modelo StAX (Streaming API for XML), que en esencia es muy parecido al modelo SAX ya comentado. Y digo muy parecido porque también se basa en definir las acciones a realizar para cada uno de los eventos generados durante la lectura secuencial del documento XML. ¿Cuál es la diferencia entonces?

La diferencia radica principalmente en que, mientras que en SAX no teníamos control sobre la lectura del XML una vez iniciada (el parser lee automáticamente el XML de principio a fin generando todos los eventos necesarios), en el modelo XmlPull vamos a poder guiar o intervenir en la lectura del documento, **siendo nosotros los que vayamos pidiendo de forma explícita la lectura del siguiente elemento** del XML y respondiendo al resultado ejecutando las acciones oportunas.

Veamos cómo podemos hacer esto, para ello creamos la clase **`RssParserXMLPull`** (`RssParserXMLPull.java`):

```
public class RssParserXMLPull {
    private URL rssUrl;

    public RssParserXMLPull(String url) {
        try {
            this.rssUrl = new URL(url);
        }
        catch (MalformedURLException e) {
            throw new RuntimeException(e);
        }
    }

    public List<Noticia> parse() {

        List<Noticia> noticias = null;
        XmlPullParser parser = Xml.newPullParser();

        try {
            parser.setInput(this.getInputStream(), null);

            int evento = parser.getEventType();

            Noticia noticiaActual = null;

            while (evento != XmlPullParser.END_DOCUMENT) {
                String etiqueta = null;

                switch (evento) {
                    case XmlPullParser.START_DOCUMENT:
                        noticias = new ArrayList<Noticia>();
                        break;

                    case XmlPullParser.START_TAG:
                        etiqueta = parser.getName();
                        if (etiqueta.equals("item")) {
                            noticiaActual = new Noticia();
```

```
                        }
                        else if (noticiaActual != null) {
                            if (etiqueta.equals("link")) {
                                noticiaActual.setLink(parser.nextText());
                            }
                            else if (etiqueta.equals("description")) {
                                noticiaActual.setDescripcion(
                                        parser.nextText());
                            }
                            else if (etiqueta.equals("pubDate"))  {
                                noticiaActual.setFecha(parser.nextText());
                            }
                            else if (etiqueta.equals("title")) {
                                noticiaActual.setTitulo(parser.nextText());
                            }
                            else if (etiqueta.equals("guid")) {
                                noticiaActual.setGuid(parser.nextText());
                            }
                        }
                        break;

                    case XmlPullParser.END_TAG:
                        etiqueta = parser.getName();
                        if (etiqueta.equals("item") && noticiaActual !=null) {
                            noticias.add(noticiaActual);
                        }
                        break;
                }

                evento = parser.next();
            }
        }
        catch (Exception ex) {
            throw new RuntimeException(ex);
        }

        return noticias;
    }


    private InputStream getInputStream() {
        try {
            return rssUrl.openConnection().getInputStream();
        }
        catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Centrándonos una vez más en el método `parse()`, vemos que tras crear el nuevo parser XmlPull y establecer el fichero de entrada en forma de stream (mediante `XmlPull.newPullParser()` y `parser.setInput(…)`) nos metemos en un bucle en el que iremos solicitando al parser en cada paso el siguiente evento encontrado en la lectura del XML,

utilizando para ello el método `parser.next()`. Para cada evento devuelto como resultado consultaremos su tipo mediante el método `parser.getEventType()` y responderemos con las acciones oportunas según dicho tipo (`START_DOCUMENT`, `END_DOCUMENT`, `START_TAG`, `END_TAG`). Una vez identificado el tipo concreto de evento, podremos consultar el nombre de la etiqueta del elemento XML mediante `parser.getName()` y su texto correspondiente mediante `parser.nextText()`. En cuanto a la obtención del texto, con este modelo tenemos la ventaja de no tener que preocuparnos por "*recolectar*" todos los fragmentos de texto contenidos en el elemento XML, ya que `nextText()` devolverá todo el texto que encuentre hasta el próximo evento de fin de etiqueta (`ENT_TAG`).

Y sobre este modelo de tratamiento no queda mucho más que decir, ya que las acciones ejecutadas en cada caso son análogas a las que ya hemos visto.

```
public class MainActivity extends AppCompatActivity {

    public String url = "https://www.europapress.es/rss/rss.aspx";

    private TextView txtResultado;
    private List<Noticia> noticias;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        txtResultado = (TextView)findViewById(R.id.txtResultado);
    }

    public void cargarXMLConXMLPull (View v){
        //Carga del XML mediante tarea Asincrona
        CargarXmlTask tarea = new CargarXmlTask();
        tarea.execute(url);
    }


    //Tarea Asíncrona para cargar un XML en segundo plano
    private class CargarXmlTask extends AsyncTask<String,Integer,Boolean> {

        protected Boolean doInBackground(String... params) {
             RssParserXMLPull xmlPullParser = new RssParserXMLPull(params[0]);
             noticias = xmlPullParser.parse();
            return true;
        }

        protected void onPostExecute(Boolean result) {
            //Tratamos la lista de noticias
            //Por ejemplo: escribimos los títulos en pantalla
            txtResultado.setText("");
            if (noticias !=  null) {
                for (int i = 0; i < noticias.size(); i++) {
                    txtResultado.setText(
                            txtResultado.getText().toString() +
                            System.getProperty("line.separator") +
                            noticias.get(i).getTitulo());
```

```
                        }
                    }
                }
            }
        }
```

ACTIVIDADES XML

1.- Realizar un programa que obtenga una lista a partir de un fichero RSS con sus correspondientes enlaces a la URL.

2.- Visualiza el tiempo de tu ciudad a partir de los ficheros XML de AEMET.
