# PROCESAMIENTO DE INFORMACIÓN EN APLICACIONES TELEMÁTICAS

# PROCESAMIENTO DE DOCUMENTOS JSON EN JAVA.

# Índice de contenidos



# Índice de figuras



# Índice de código

# 1 Introducción

Las API Java para el procesamiento de información en formato JSON, proporcionan métodos que permiten el análisis, transformación y generación de una forma ágil.

Atendiendo al modelo de datos (estructura de los datos en memoria) y al tipo de procesamiento se pueden utilizar diferentes estrategias para el desarrollo de soluciones:

- **Modelo de datos basado en objetos Java**: la información se almacena en memoria en objetos JAVA. En operaciones de serialización (conversión de objetos Java a JSON), los datos en memoria se corresponden con la información a transformar a formato JSON; en operaciones de deserialización (conversión de JSON a objetos Java) la información en formato JSON se carga en estructuras en memoria. Según las estructuras en memoria utilizadas se pueden identificar dos estrategias:
  - **`Tree model`** (equivalente a `DOM` para `XML`): también conocido como **`Object model.`** Se crea una estructura de árbol en memoria que representa la estructura de los datos JSON. Modelo flexible al ser independiente de la estructura del documento.
  - **`Data binding`**: se utilizan clases simples Java (`POJO`) para modelar la información JSON a serializar o deserializar, utilizando métodos `get/set` o anotaciones (`annotations`). Modelo dependiente de la estructura del documento a serializar/deserializar.
- **Modelo de procesamiento en tiempo real** (**`Streaming model`**): también conocido como modelo de procesamiento de flujo de datos. La información no se mapea en memoria, la serialización/deserialización se realiza secuencialmente generando o consumiendo `tokens` identificados en un flujo en formato JSON. Es el modelo que menos recursos de memoria requiere.

Entre las API Java de código abierto existentes, con soporte para los tres tipos de procesamiento (`tree model`, `data binding` y `streaming`), las de uso más extendido son las siguientes,:

- **GSON** de Google (https://github.com/google/gson).
- **FasterXML** del proyecto Jackson (https://github.com/FasterXML/jackson).
- **JSON-P** de Oracle (https://jsonp.java.net/)

En este documento se presentan ejemplos didáctico orientados a la comprensión del problema de extracción de información de documentos JSON, utilizando la API de código abierto **GSON**. Se va a trabajar con la versión **`2.8.6`** disponible en el repositorio Maven: https://search.maven.org/artifact/com.google.code.gson/gson/2.8.6/jar.

> En **Moodle** se encuentra disponible los **fichero JSON** utilizados, el **código fuente** de los ejemplos, y el fichero **`piatGSON-ejemplo.jar`** para poder probar las clases desde la línea de mandatos.

# 2 API GSON

Esta biblioteca se encuentra formada por un conjunto de clases estructuradas en cuatro paquetes:

- com.google.gson: principales clases implicadas en la conversión de objetos Java a JSON y viceversa.
- com.google.gson.annotations: anotaciones que se pueden utilizar en las clases simples (POJO) implicadas en una solución basada en el modelo Data Binding.
- com.google.gson.reflect: utilidades para manipulación de tipos genéricos.
- com.google.gson.stream: clases para el procesamiento de documentos en tiempo real.

Este apartado tiene por objeto presentar unos sencillos ejemplos que proporcionen conocimientos mínimos sobre los tres tipos de tipos de analizadores JSON que se pueden implementar utilizando la API GSON.

En concreto, se va a codificar un analizador (`parser`) para cada uno de los tipos de procesamiento (`tree model`, `data binding` y `streaming`), que muestre en el dispositivo de salida estándar el contenido del documento **`robot.json`**. El documento a procesar, se pasará como argumento en la línea de mandatos.

```
{
    "id": "ROB001",
    "locomocion": "ruedas",
    "dimensiones" : {
        "largo": 30,
        "alto": 40,
        "ancho": 20
    },
    "sensores": [
        {
            "id": "SEN1a",
            "medida": "temperatura del aire",
            "unidades": "http://qudt.org/vocab/unit/DEG_C"
        },
        {
            "id": "SEN1b",
            "medida": "humedad del aire",
            "unidades": "http://qudt.org/vocab/unit/PERCENT"
        }
    ]
}
```

Código 1 – robot.json

Ejemplo del mandato de ejecución de cada una de las clases:

```
$ java -cp piatGSON-ejemplo.jar piat.examples.robot.ObjectModelParser robot.json

$ java -cp piatGSON-ejemplo.jar piat.examples.robot.DataBindingParser robot.json

$ java -cp piatGSON-ejemplo.jar piat.examples.robot.StreamingParser robot.json
```

La ejecución de cualquiera de estos ejemplos proporcionará el siguiente resultado:

```
- id: ROB001
- locomocion: ruedas
- dimensiones
        . largo: 30
        . alto: 40
        . ancho: 20
- sensores
       sensor 1
                . id: SEN1a
                . medida: temperatura del aire
                . unidades: http://qudt.org/vocab/unit/DEG_C
       sensor 2
                . id: SEN1b
                . medida: humedad del aire
                . unidades: http://qudt.org/vocab/unit/PERCENT
```

## 2.1 GSON - Tree model

El modelo de objetos se basa en la representación de un documento JSON como una estructura jerárquica de elementos de tipo **`JsonElement`** (com.google.gson.JsonElement).

La clase **`JsonElement`** representa a cualquier nodo de la jerarquía, pudiendo ser de alguno de los siguientes tipos:

- **`JsonObject`**: representa un objeto JSON, es decir una secuencia de propiedades representadas mediante pares **`clave-valor`** en los que **`clave`** es una cadena de caracteres con el nombre la propiedad y **`valor`** cualquier tipo de objeto `JsonElement`.
- **`JsonArray`**: representa un array JSON, es decir, una lista de objetos `JsonElement`, pudiendo ser cada uno de ellos de un tipo diferente.
- **`JsonPrimitive`**: representa un tipo de datos primitivo: `simple`, `number`, `boolean`, ...
- **`JsonNull`**: representa a un objeto `null`.

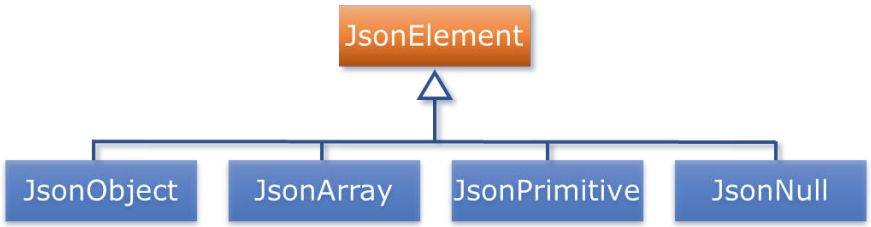


Figura 1.- Clase JsonElement

El siguiente fragmento de código muestra la estructura de un analizador que presenta, en el dispositivo de salida estándar, el contenido de los nodos `JsonElement` del árbol del documento `robot.json`, recorriendo para ello toda la estructura. Con objeto de simplificar el ejemplo, se ha omitido la validación de argumentos de la línea de mandatos y la gestión de excepciones. El fragmento de código se ha extraído de la clase **`piat.examples.robot.ObjectModelParser`**.

```
// elemento root del documento
JsonElement oElemRobot = JsonParser.parseReader(
                    new InputStreamReader(new FileInputStream(args[0]), "UTF-8") );

// como conocemos la estructura de robot, suponemos que el elemento root es de tipo JsonObject,
JsonObject oRobot = oElemRobot.getAsJsonObject();

//propiedades del robot a través de sus nombres,
System.out.format("\n - id: %s\n - locomocion: %s",
                    oRobot.get("id").getAsString(),
                    oRobot.get("locomocion").getAsString());

//propiedad 'dimensiones' de tipo object
System.out.print("\n - dimensiones");
for( Entry<String, JsonElement> oProperty : oRobot.get("dimensiones").getAsJsonObject().entrySet()) {
    System.out.format("\n\t . %s: %s",
                    oProperty.getKey(),
                    oProperty.getValue().getAsString()
                    );
}

//propiedad 'sensores' de tipo array
System.out.print("\n - sensores");
int nSensor = 0;
for(JsonElement oElemSensor : oRobot.get("sensores").getAsJsonArray()) {
    nSensor++;
    System.out.print("\n\tsensor " + nSensor);
    for( Entry<String, JsonElement> oProperty : oElemSensor.getAsJsonObject().entrySet()) {
        System.out.format("\n\t\t . %s: %s",
                        oProperty.getKey(),
                        oProperty.getValue().getAsString()
                        );
    }

}
```

Código 2 - Clase piat.examples.robot.ObjectModelParser

Analizando en detalle el código, lo primero que se hace es invocar al método estático **`parseReader`** de la clase **`JsonParser`**, pasándole un objeto `Reader` que recibe como flujo de entrada el contenido del fichero pasado como argumento de la línea de mandatos. Se utiliza la combinación de `InputStreamReader` y `FileInputStream` para asegurarse de que el fichero es serializado utilizando la codificación **`UTF-8`**. Si no se produce ningún error, el método devuelve un objeto `JsonElement`, que se corresponderá en con el elemento raíz del documento (Figura 2).

```
    // elemento root del documento
    JsonElement oElemRobot = JsonParser.parseReader(
                        new InputStreamReader(new FileInputStream(args[0]), "UTF-8") );
```

Código 3 – Conversión a objetos Java mediante JsonParser

La Figura 2 muestra el árbol de nodos de tipo `JsonElement` que representan la estructura del documento **`robot.json`**.

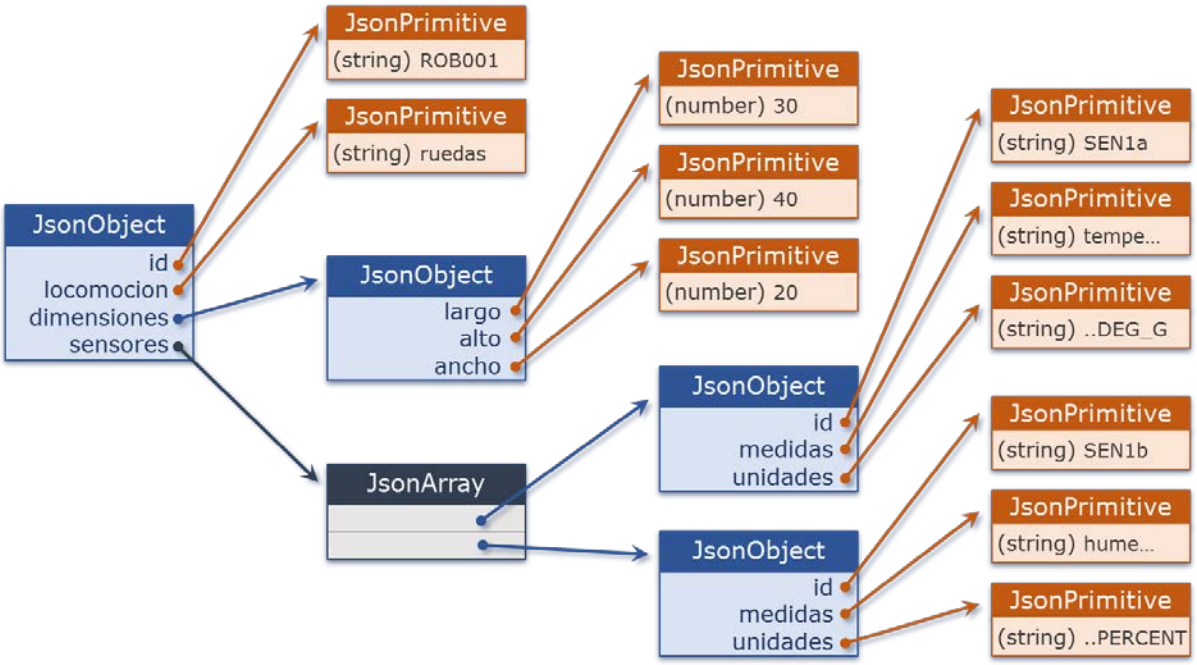


Figura 2.- Modelo de objetos de **robot.json**

Por definición del modelo, todos los nodos del árbol son objetos inmutables de tipo `JsonElement`, sin embargo, para poder utilizar los métodos específicos del tipo de elemento es necesario convertir a la clase adecuada, utilizando alguno de los siguientes métodos del objeto `JsonElement`: **`getAsJsonObject()`**, **`getAsJsonArray()`**, **`getAsJsonPrimitive`** o **`getAsJsonNull()`**.

Como conocemos la estructura del documento podemos asegurar que el elemento raíz del documento es de tipo **`JsonObject`** y lo podemos convertir mediante el método **`getAsJsonObject`** de la clase `JsonElement`.

```
JsonObject oRobot = oElemRobot.getAsJsonObject();
```

Código 4 – JsonElement: ejemplo de utilización del método getAsObject()

A continuación se extraen las propiedades del objeto `robot`. Como la información se encuentra en memoria, se puede consultar atendiendo a las necesidades del algoritmo.

Para la obtención de los valores asociados a las propiedades id y locomoción, se utiliza el método **`get`**, que permite obtener el valor de una propiedad conocido su nombre, dicho método devuelve un objeto de tipo `JsonElement`. Para obtener el contenido de un elemento se pueden utilizar métodos como: **`getAsBoolean()`**, **`getAsDouble()`**, **`getAsFloat()`**, **`getAsInt()`**, **`getAsLong()`**, **`getAsNumber()`**, **`getAsShort()`**, **`getAsString()`**, **`….`** Si el objeto no es de tipo `JsonPrimitive` o no se puede realizar la conversión al tipo asociado, se generará una excepción.

```
        //propiedades del robot a través de sus nombres,
        System.out.format("\n - id: %s\n - locomocion: %s",
                          oRobot.get("id").getAsString(),
                          oRobot.get("locomocion").getAsString());
```

Código 5 – JsonObject: acceso al valor de una propiedad a través de su nombre

El valor asociado a la propiedad dimensiones es de tipo objeto por lo que deberá ser tratado como tal. En el código de ejemplo, en lugar de acceder a cada una de las propiedades del objeto con el método **get**, se ha optado por un algoritmo que permite explorar todas las propiedades de un objeto **JsonObject**, sin necesidad de conocer cómo se llaman ni cuántas hay. Para ello se utiliza el método **entrySet()** del objeto **JsonObject** y se procesan cada una de los elementos de la colección devuelta por el método.

Se ha supuesto que el valor asociado a todas las propiedades es de tipo JsonPrimitive().

```
//propiedad 'dimensiones' de tipo object
System.out.print("\n - dimensiones");
for( Entry<String, JsonElement> oProperty : oRobot.get("dimensiones").getAsJsonObject().entrySet()) {
    System.out.format("\n\t . %s: %s",
                      oProperty.getKey(),
                      oProperty.getValue().getAsString()
                      );
}
```

Código 6 – JsonObject: procesamiento genérico de todas las propiedades

La propiedad sensores es un array, por lo que su procesamiento completo implica recorrer uno a uno los elemento del array y procesarlos en función del tipo de elemento que sean. Si el array fuera heterogéneo, se podría utilizar uno de los siguientes métodos para determinar el tipo de elemento y aplicar el procesamiento correspondiente: **isJsonObject()**, **isJsonArray()**, **isJsonPrimitive()** o **isJsonNull()**.

En el código de ejemplo se ha supuesto que todos los elementos del array son de tipo **JsonObject** y se han extraído las propiedades de cada uno de ellos de forma genérica suponiendo que todas son de tipo **JsonPrimitive**.

```
//propiedad 'sensores' de tipo array
System.out.print("\n - sensores");
int nSensor = 0;
for(JsonElement oElemSensor : oRobot.get("sensores").getAsJsonArray()) {
    nSensor++;
    System.out.print("\n\tsensor " + nSensor);
    for( Entry<String, JsonElement> oProperty : oElemSensor.getAsJsonObject().entrySet()) {
        System.out.format("\n\t\t . %s: %s",
                          oProperty.getKey(),
                          oProperty.getValue().getAsString()
                          );
    }

}
```

Código 7 – Algoritmo de procesamiento o de un objeto de tipo JsonArray

## 2.2 GSON – Data binding

La API **Data Binding** se utiliza para la conversión de objetos Java a JSON y viceversa. Proporciona soporte para la conversión tanto de tipos de datos básicos definidos en Java Core y colecciones incorporadas, como para objetos POJO (Plain Old Java Object).

La clase **`Gson`** (com.google.gson.Gson) es la que permite realizar las operaciones de conversión mediante los métodos **`fromJson()`** y **`toJson()`**. Existen dos formas de crear una instancia de esta clase:

- Mediante el constructor **`Gson()`**: se crea una instancia *thread-safe* con la configuración de conversión por defecto.
- Mediante la clase **`GsonBuilder`** (com.google.gson.GsonBuilder): permite establecer opciones de configuración de conversión.

La Figura 3 muestra el diagrama de clases que modela la información existente en **`robot.json`**.

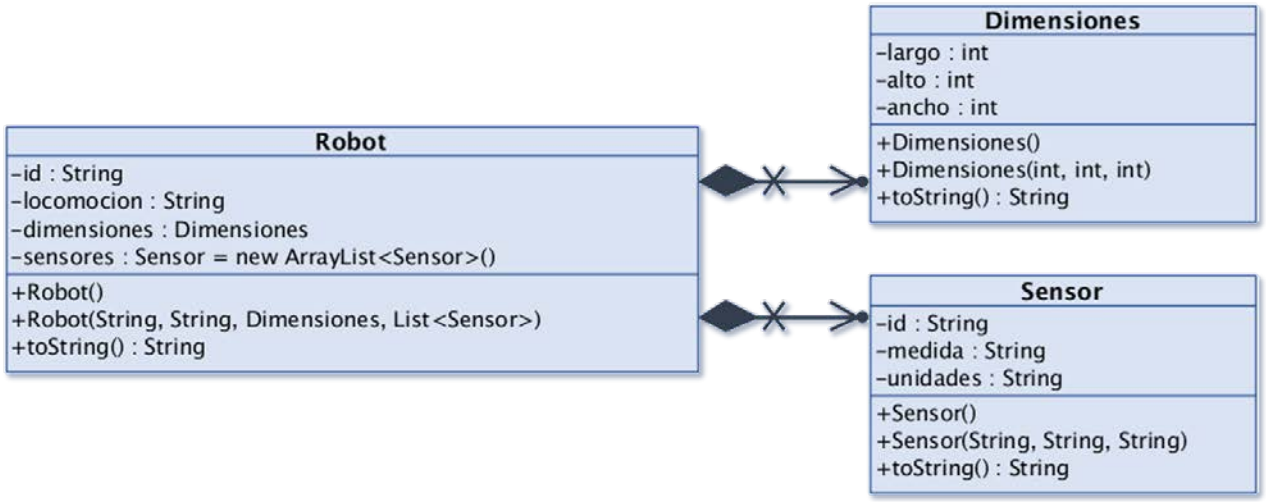


Figura 3 – Diagrama de clases asociada a robot.json

Existen herramientas que generan automáticamente la estructura de clases simples a partir de JSON. Un ejemplo es la herramienta en línea `jsonschema2pojo` (http://www.jsonschema2pojo.org/), soporta varios formatos de entrada (`JSON`, `JSONSchema`, `YAML` y `YAMLSchema`), permite generar anotaciones de salida para diferentes API (`GSON`, `Jackson`,…) y ofrece varias opciones de configuración.

El siguiente fragmento de código muestra la estructura de un analizador que presenta, en el dispositivo de salida estándar, el resultado de la conversión del documento `robot.json` a un objeto de la clase **`Robot`**. El fragmento de código se ha extraído de la clase **`piat.examples.robot.DataBindingParser`**.

```
Gson oGson = new Gson();
Robot oRobot;

oRobot = oGson.fromJson(new InputStreamReader(new FileInputStream(args[0]), "UTF-8"), Robot.class);

//propiedades básicas del robot
System.out.format("\n - id: %s\n - locomocion: %s",
                  oRobot.getId(),
                  oRobot.getLocomocion());

//propiedad 'dimensiones'
System.out.print("\n - dimensiones");
System.out.format("\n\t. largo: %s\n\t. alto: %s\n\t. ancho: %s",
        oRobot.getDimensiones().getLargo(),
        oRobot.getDimensiones().getAlto(),
        oRobot.getDimensiones().getAncho()
        );

//propiedad 'sensores' de tipo array
System.out.print("\n - sensores");
int nSensor = 0;
for( Sensor oSensor : oRobot.getSensores()) {
    nSensor++;
    System.out.print("\n\tsensor " + nSensor);
    System.out.format("\n\t\t. id: %s\n\t\t. medida: %s\n\t\t. unidades: %s",
            oSensor.getId(),
            oSensor.getMedida(),
            oSensor.getUnidades()
            );
}
```

Código 8 -Clase piat.examples.robot.DataBindingParser

Como se observa, el algoritmo del analizador no presenta ninguna diferencia con lo que podría se una aplicación java con las que se ha trabajado en otras asignaturas. Una vez convertido el documento JSON a un objeto **Robot**, el procesamiento sólo depende de la lógica de negocio a implementar.

La conversión del contenido del documento JSON a una instancia de la clase **Robot** se hace mediante el método **fromJson** de un objeto **Gson**, pasándole la clase Java que representa al elemento raíz del documento.

```
oRobot = oGson.fromJson(new InputStreamReader(new FileInputStream(args[0]), "UTF-8"), Robot.class);
```

Código 9 – Conversión, a objetos Java, de flujo de datos en formato JSON mediante Gson.fromJson()

El siguiente fragmento de código se corresponde con el de la clase **Robot** (**piat.examples.robot.pojo.Robot**). El código de esta clase, y del resto de clases del diagrama presentado en la Figura 3 (**piat.examples.robot.pojo.Dimensiones** y **piat.examples.robot.pojo.Sensor,** ha sido generado mediante la herramienta **jsonchema2pojo** a partir del fichero `robot.json`.

```java
public class Robot {
    @SerializedName("id")
    private String sid;

    @SerializedName("locomocion")
    private String sLocomocion;

    @SerializedName("dimensiones")
    private Dimensiones oDimensiones;

    @SerializedName("sensores")
    private List<Sensor> lSensores = new ArrayList<Sensor>();

    //constructor sin argumentos
    public Robot() {}
    //constructor con argumento
    public Robot(String id, String locomocion, Dimensiones dimensiones, List<Sensor> sensores) {
        super();
        this.sid = id;
        this.sLocomocion = locomocion;
        this.oDimensiones = dimensiones;
        this.lSensores = sensores;
    }

    public String getId() {
        return sid;
    }
    public void setId(String id) {
        this.sid = id;
    }

    // resto de getter y setter
     ----------------
}
```

Código 10 - Clase piat.examples.robot.pojo.Robot

El código está formado por la definición de las propiedades, los constructores y los *getter*/*setter* de cada propiedad. Junto a la definición de cada propiedad se encuentra una anotación (**`@SerializedName`**) que permite indicar al *serializador*/*deserializador* JSON a qué propiedad del objeto JSON debe asociarla, si no se indica esta anotación, se asumirá que el nombre de la propiedad JSON es equivalente al del objeto JAVA correspondiente. Las clases **`Dimensiones`** y **`Sensor`** son análogas.

La utilización de anotaciones tienen mayor funcionalidad que la presentada en este ejemplo, se recomienda consultar el paquete com.google.gson.annotations.

El ejemplo presentado es este apartado una primera aproximación a los mecanismos de deserialización a objetos JAVA de GSON, en el que se ha utilizado el comportamiento predeterminado mediante el método **`fromJson()`**. Para escenarios reales en los que la estructura de los objetos de una aplicación no se corresponden directamente con la estructura del documento JSON, GSON ofrece la interfaz **`JsonDeserializer`** que permite definir conversores de JSON a objetos JAVA personalizados.

## 2.3 GSON - Streaming

Esta API proporciona acceso directo de solo lectura a la información de un flujo de datos (`stream`) en formato JSON, mediante un objeto de tipo **`JsonReader`** (com.google.gson.stream.JsonReader).

En este modelo el código de la aplicación controla el proceso de lectura consumiendo secuencialmente los diferentes token que el objeto **`JsonReader`** identifica en un documento JSON, implementando un procesamiento orientado a eventos tipo **`pull`**.

Los diferentes token que se pueden consumir son:

- **Valores literales** (cadenas de caracteres, números, valores lógicos o valor `null`): se corresponden con nombres, contenidos de propiedades o contenidos de componentes simples de arrays.
- **Delimitadores**: caracteres que indican inicio o fin de objetos (**`{`**, **`}`**) o de arrays (**`[`**, **`]`**).

En el momento de abordar el diseño de una aplicación, que adopte este modelo de procesamiento, se tiene que asegurar que éste sea independiente del orden de aparición de las propiedades y de los objetos del documento JSON a analizar ya que, aun cuando el flujo de datos a procesar se valide sobre un JSONSchema, esta validación sólo asegura la presencia de las propiedades y objectos obligatorios y la adecuación de los tipos de datos de los contenidos, no siendo significativo el orden de aparición de los mismos.

```
//apertura del documento
JsonReader oReader = new JsonReader(new InputStreamReader(new FileInputStream(args[0]), "UTF-8"));

// como conocemos la estructura de robot, suponemos que el elemento root es de tipo JsonObject,
oReader.beginObject();              //equivalente a leer el inicio del objeto root '{'
String sNombre;

// Procesamiento de cada uno de los pares clave-calor del objeto
while(oReader.hasNext()) {
    sNombre = oReader.nextName();

    switch (sNombre) {
    //propiedades 'id' y 'locomocion'
        case "id":
        case "locomocion":
            System.out.format("\n - %s: %s", sNombre, oReader.nextString() );
            break;
        case "dimensiones":
            System.out.print("\n - dimensiones");
            // Aquí vendrá el código que permita consumir secuencialmente el conjunto de tokens asociados a la propiedad

            break;

        case "sensores": //propiedad 'sensores' de tipo array
                System.out.print("\n - sensores");
                // Aquí vendrá el código que permita consumir secuencialmente el conjunto de tokens asociados a la propiedad
                break;

            default:
                // cuando se encuentra una propiedad no esperada o que no se desea procesar, se debe ignorar.
                // el método skipValue consume recursivamente todos los token asociados.
                oReader.skipValue();
            }
}
oReader.endObject();                //equivalente a leer el fin del objeto root '}'

oReader.close();                    //equivalente a cerrar el fichero
```

Código 11 - Esqueleto de la clase piat.examples.robot.StreamingParser

Este fragmento de código muestra la estructura de un analizador que presenta, en el dispositivo de salida estándar, el contenido de cada uno de los elementos del documento `robot.json`, mediante el procesamiento secuencial de los token identificados en dicho documento por un objeto **`JsonParser`**. El fragmento de código se ha extraído de la clase **`piat.examples.robot.StreamingParser`**.

Aunque en una aplicación real se debería aplicar un diseño estructurado definiendo un método privado para el procesamiento de cada uno de los valores no primitivos del documento, en este código se ha omitido dicho diseño para facilitar su seguimiento.

El primer paso es crear el objeto **`JsonParser`** que permite procesar el flujo de datos de entrada determinado por el contenido del fichero **`robot.json`**.

```
//apertura del flujo de datos
JsonReader oReader = new JsonReader(new InputStreamReader(new FileInputStream(args[0]), "UTF-8"));
```

Código 12 - Vinculación de un objeto JsonReader con un flujo de datos en formato JSON

Como sabemos que el elemento raíz del documento es un objeto deberemos aplicar el algoritmo genérico de procesamiento:

- Invocar al método **`beginObject()`** para consumir el delimitador **`{`**.
- Mediante una estructura iterativa gestionada por la condición **`hasNext()`** procesar secuencialmente cada uno de los pares clave-valor asociados al objeto.
- Invocar al método **`endObject()`** para consumir el delimitador **`}`**.

El fragmento iterativo del algoritmo deberá recordar toda la información que sea necesaria para la lógica de negocio de la aplicación y deberá estar diseñado de tal forma que en cada iteración se consuma un único par `clave-valor`. Si el diseño es correcto podremos asegurar:

- que el siguiente `token` a consumir tras una evaluación afirmativa de la condición **`hasNext()`** es un `token` de tipo **`JsonToken.NAME`** que podrá ser consumido mediante el método **`nextName()`**.
- que una vez consumido el `token` de tipo **`JsonToken.NAME`** el siguiente token a consumir deberá ser un `token` de tipo contenido primitivo (`JsonToken.BOOLEAN`, `JsonToken.NUMBER`, `JsonToken.STRING` o `JsonToken.NULL`), inicio de objeto (`JsonToken.BEGIN_OBJECT`) o inicio de array (`JsonToken.BEGIN_ARRAY`).
- que se han consumido todos los `token` asociados al `valor` del último par procesado.

El algoritmo del siguiente código de ejemplo identifica y extrae las propiedades del elemento raíz siguiendo el diseño descrito.

```
// Procesamiento de cada uno de los pares clave-calor del objeto
while(oReader.hasNext()) {
    sNombre = oReader.nextName();

    switch (sNombre) {
    //propiedades 'id' y 'locomocion'
        case "id":
        case "locomocion":
            System.out.format("\n - %s: %s", sNombre, oReader.nextString() );
            break;
        case "dimensiones":
            System.out.print("\n - dimensiones");
```

Código 13 – Procesamiento de los pares clave-valor de un objeto

El valor asociado a la propiedad dimensiones es de tipo objeto por lo que deberá ser procesado atendiendo al algoritmo general. Por analogía con el diseño presentado en el apartado anterior para la `ObjectModelParse`, se ha optado por hacer un procesamiento de cada par clave-valor sin hacer suposiciones sobre el nombre de la propiedad, procesando únicamente aquellos pares en los que el valor sea de tipo primitivo diferente a null y no procesando, pero sí consumiendo, el resto de valores utilizando el método **`skipValue()`**.

```
case "dimensiones": //propiedad 'sensores' de tipo object
    System.out.print("\n - dimensiones");
    // Aquí vendrá el código que permita consumir secuencialmente el conjunto de tokens asociados a la propiedad
    String sNombreDim;
    oReader.beginObject();      //equivalente a leer el inicio del objeto dimensiones '{'
    while(oReader.hasNext()) {
        sNombreDim = oReader.nextName();
        //ejemplo de extracción de todas las propiedades con valores de tipo primitivo y no nulo
        if (oReader.peek()==JsonToken.STRING ||
            oReader.peek()==JsonToken.NUMBER||
            oReader.peek()==JsonToken.BOOLEAN) {
            System.out.format("\n\t . %s: %s", sNombreDim, oReader.nextString() );
        }else {
            oReader.skipValue();
        }
    }
    oReader.endObject();        //equivalente a leer  el fin del objeto dimensiones '}'
    break;
```

Código 14 - Procesamiento genérico de los pares clave-valor con valores de tipo primitivo

Cabe destacar que al tratarse de un procesamiento secuencial, el almacenamiento en la variable `sNombreDim` del nombre de la propiedad debe hacerse en el momento en el que se encuentra disponible el `token` **`JsonToken.NAME`**, aun cuando no proceda el procesamiento del valor asociado (por no tratarse de un tipo primitivo distinto de null), condición que no puede ser evaluada hasta que se encuentre disponible el siguiente `token`.

El valor de la propiedad sensores es de tipo array y deberá ser procesado aplicando el algoritmo genérico de procesamiento de arrays:

- Invocar al método **`beginArray()`** para consumir el delimitador **`[`**.
- Mediante una estructura iterativa gestionada por la condición **`hasNext()`** procesar secuencialmente cada uno de los componentes del array.
- Invocar al método **`endArray()`** para consumir el delimitador **`]`**.

El fragmento iterativo del algoritmo deberá recordar toda la información que sea necesaria para la lógica de negocio de la aplicación y deberá estar diseñado de tal forma que en cada iteración se consuma un único elemento del array. Si el diseño es correcto podremos asegurar:

- que el siguiente `token` a consumir tras una evaluación afirmativa de la condición **`hasNext()`** es un componente.
- que el componente a consumir deberá ser un `token` de tipo contenido primitivo (`JsonToken.BOOLEAN`, `JsonToken.NUMBER`, `JsonToken.STRING` o `JsonToken.NULL`), inicio de objeto (`JsonToken.BEGIN_OBJECT`) o inicio de array (`JsonToken.BEGIN_ARRAY`).
- que se han consumido todos los `token` asociados al último elemento procesado.

Como se sabe que el array es heterogéneo y se encuentra formado por objetos, se aplica el algoritmo de procesamiento utilizado para `dimensiones`.

```
case "sensores": //propiedad 'sensores' de tipo array
        System.out.print("\n - sensores");
        // Aquí vendrá el código que permita consumir secuencialmente el conjunto de tokens asociados a la propiedad
        oReader.beginArray();       //equivalente a leer el inicio del array sensores '['
        int nSensor = 0;
        while(oReader.hasNext()) {  //recorrido de cada los elementos de array, se espera que sean Objec
            oReader.beginObject();
            String sNombreSensor;
            nSensor++;
            System.out.print("\n\tsensor " + nSensor);
            while(oReader.hasNext()) {
                sNombreSensor = oReader.nextName();

                if (oReader.peek()==JsonToken.STRING ||
                    oReader.peek()==JsonToken.NUMBER||
                    oReader.peek()==JsonToken.BOOLEAN) {
                    System.out.format("\n\t . %s: %s", sNombreSensor, oReader.nextString() );
                }else {
                    oReader.skipValue();
                }
            }
            oReader.endObject();
        }
        oReader.endArray();     //equivalente a leer el fin del array sensores ']'
        break;
```

Código 15 - Procesamiento de los elemento de un array

Una vez finalizado el procesamiento se deberá cerrar el objeto **`JsonReader`** mediante el método **`close()`**, este método se encargará de cerrar el flujo de datos asociados.

# 3 Ejemplo: aplicación de extracción de información

En este apartado se aborda el diseño e implementación de una aplicación que permite extraer información sobre regiones administrativas españolas (comunidades autónomas o provincias) a partir de la información almacenada en un documento JSON.

El documento contiene información sobre el número de recursos naturales que, de un determinado tema o sector, pueden ser visitados en cada provincia. La aplicación a desarrollar deberá proporcionar información sobre las comunidades autónomas que tienen recursos asociados a un tema.

Restricciones de realización:

- La aplicación recibirá por línea de mandatos la ruta (`path`) al fichero JSON y una secuencia con los temas (`Cultura-ocio`, `Medio-ambiente`, `Sector-publico`, `Transporte` y/o `Turismo`) de los que se desea información. Al menos deberá especificarse un `tema`.
- La validación de argumento y la presentación de resultados deberá hacerse en el método estático **`main()`**.
- La clase que implemente la aplicación, deberá tener un **`constructor`** al que se le pase la ruta al fichero JSON y deberá implementar el método público **`buscarCCAAPorTema()`** que devolverá una lista con los nombres de las comunidades autónomas que tienen recursos del **`tema`** pasado como argumento.

```
public List<String> buscarCCAAPorTema(String sTema) throws Exception {}
```

- El código estará diseñado suponiendo que el documento de entrada está bien formado, es válido y contiene todos los elementos obligatorios.
- La aplicación mostrará en el dispositivo de salida estándar, el nombre de las comunidades autónomas asociadas a cada uno de los temas.

En concreto, se van a diseñar e implementar tres aplicaciones, una por cada una de las API GSON presentadas en el apartado anterior. Cada una de ellas proporcionará la funcionalidad especificada respetando las restricciones de realización. La clase principal de cada una de las aplicaciones será la siguiente:

- Clase **`piat.examples.regadmin.ObjectModelParser`**: analizador implementado mediante la API `tree model`.
- Clase **`piat.examples.regadmin.DataBindingParser`**: analizador implementado mediante la API `data binding`.
- Clase **`piat.examples.regadmin.StreamingParser`**: analizador implementado mediante la API `streaming`.

Ejemplo del mandato de ejecución de cada una de las clases, solicitando información sobre los temas `Medio-ambiente` y `turismo`:

```
java -cp piatGSON-ejemplo.jar piat.examples.regadmin.ObjectModelParser regAdmin.json
Medio-ambiente turismo

java -cp piatGSON-ejemplo.jar piat.examples.regadmin.DataBindingParser regAdmin.json
Medio-ambiente turismo

java -cp piatGSON-ejemplo.jar piat.examples.regadmin.StreamingParser regAdmin.json
Medio-ambiente turismo
```

El resultado para los tres ejemplo será:

```
Tema: Medio-ambiente
   . Aragón
   . Castilla y León
   . Castilla-La Mancha
   . Comunidad de Madrid
   . País Vasco/Euskadi

Tema: turismo
   . Castilla y León
   . Comunidad Foral de Navarra
   . País Vasco/Euskadi
```

El método **`main()`** será similar en las tres clases y equivalente al siguiente:

```
public static void main(String[] args) {

    if (args.length<2){
        uso("ERROR#NUM: Argumentos incorrectos - " + args.length );
        System.exit(0);
    }

    try {
        ObjectModelParser oParser = new ObjectModelParser(args[0]);
        List<String> lCCAA = null;
        for(int i=1; i<args.length; i++) {
            System.out.format("\nTema: %s\n", args[i]);

            lCCAA = oParser.buscarCCAAPorTema(args[i]);
            for(String sCCAA : lCCAA)
                System.out.format("\t. %s\n", sCCAA);
            }
    } catch (Exception  e) {
        System.err.format("\nERROR#FILE: el fichero '%s' no existe, no es un documento JSON o está mal formado\n",
                          args[0]);
        e.printStackTrace();
    }

    System.exit(0);
}
```

Código 16 - Método main()

El esquema del documento JSON (**`regAdmin.json`**) del que extraer la información es el siguiente:

- El elemento raíz es un objeto que tiene, al menos, las propiedades primitivas **`id`** y **`numCCAA`**, y la propiedad de tipo array **`ccaa`**.
- El array **`ccaa`** se encuentra formado por una secuencia de objetos, cada uno de ellos representa a la información de una Comunidad Autónoma (**`CA`**).

- Cada objeto **CA** tiene, al menos, las propiedades primitivas **codigo**, **nombre**, **numProvincia**, **numRecursos**, y la propiedad de tipo array **provincias**.
- El array **provincias** se encuentra formado por una secuencia de objetos, cada uno de ellos representa a la información de un Provincia (**PR**).
- Cada objeto **PR** tiene, al menos, las propiedades primitivas **codigo**, **nombre** y **numRecursos**, y opcionalmente la propiedad de tipo array **temas** .
- El array **temas** se encuentra formado por una secuencia de objetos, cada uno de ellos representa a la información de un **tema**.
- Cada objeto **tema** tiene, al menos, las propiedades primitivas **codigo**, **nombre** y **numRecursos**.

El siguiente listado muestra el contenido del documento con la información asociada a la comunidad autónoma de Madrid, habiéndose omitido la información del resto de comunidades autónomas.

```
{
    "id": "Comunidades Autónomas",
    "numCCAA": 19,
    "ccaa": [
        {
            "codigo": "13",
            "nombre": "Comunidad de Madrid",
            "numProvincias": 1,
            "numRecursos": 4685,
            "provincias": [
                {
                    "codigo": "28",
                    "nombre": "Madrid",
                    "numRecursos": 4685,
                    "temas": [
                        {
                            "codigo": "013",
                            "nombre": "Medio-ambiente",
                            "numRec": 2281
                        },
                        {
                            "codigo": "016",
                            "nombre": "Sector-publico",
                            "numRec": 514
                        },
                        {
                            "codigo": "019",
                            "nombre": "Transporte",
                            "numRec": 1890
                        }
                    ]
                }
            ]
        }
    ]
}
```

Código 17 - Ejemplo de regAdmin.json

## 3.1 Deserializando a objetos Java: object model y data binding

Existe una gran similitud en el diseño y código de las dos clases que implementan la aplicación deserializando el contenido del documento **regAdmin.json** en objetos Java, ya sea utilizado el modelo de objetos (**piat.examples.regadmin.ObjectModelParser**) o el modelo data binding (**piat.examples.regadmin.DataBindingParser)**. Es por ello que el análisis de ambas soluciones se aborda conjuntamente.

```
public class ObjectModelParser {
    private JsonElement oRoot = null;
    private ArrayList<String> lCCAA = null;

    //constructor
    public ObjectModelParser(String sFile) throws Exception {
        oRoot = JsonParser.parseReader(new InputStreamReader(new FileInputStream(sFile), "UTF-8"));
    }

    //método público a implementar
    public List<String> buscarCCAAPorTema(String sTema) {
        lCCAA = new ArrayList<String>();
        // Comunidades Autónomas
        parseCCAA(oRoot.getAsJsonObject().get("ccaa").getAsJsonArray(), sTema);
        return lCCAA;
    }

    //método privado para el procesamiento de una CA
    private void parseCCAA(JsonArray oArrayCCAA, String sTema) {
        for(JsonElement oCCAA : oArrayCCAA) {
            // Provincias
            if (parsePR(oCCAA.getAsJsonObject().get("provincias").getAsJsonArray(), sTema)) {
                //no es necesario seguir procesando el resto de provincias
                lCCAA.add(oCCAA.getAsJsonObject().get("nombre").getAsString() );
            }
        }
    }

    //método privado para el procesamiento de una PR
    private boolean parsePR(JsonArray oArrayPR, String sTema) {
        for(JsonElement oPR : oArrayPR) {
            // Temas
            if (oPR.getAsJsonObject().get("temas")!=null) {
                for(JsonElement oTema : oPR.getAsJsonObject().get("temas").getAsJsonArray()) {
                    if(oTema.getAsJsonObject().get("nombre").getAsString().equalsIgnoreCase(sTema)) {
                        //no es necesario seguir procesando el resto de temas
                        return true;
                    }
                }
            }
        }
        return false;
    }

    public static void main(String[] args) {
    ----------------------------
    }
}
```

Código 18 - Clase piat.examples.regadmin.ObjectModelParser

Principales aspectos del diseño de los dos analizadores:

- La información en memoria debe cargarse una única vez, lo correcto es que esta operación se desencadene en el momento de generar la instancia del analizador (al invocar al constructor).

- Los métodos de búsqueda de información implican la codificación de algoritmos a través de las estructuras en memoria. El algoritmo es muy parecido en ambos analizadores y las diferencias se encuentran determinadas por el modelo de objetos utilizado. En ambos casos, y con objeto de optimizar el procesamiento:
  - cuando se encuentra una evidencia de que una comunidad es pertinente, se deja de analizar el resto provincias de la comunidad.
  - cuando se encuentra una evidencia de que una provincia es pertinente, se deja de analizar el resto de temas de la provincia.

```
public class DataBindingParser {
    private RegAdmin oRegAdmin = null;
    private ArrayList<String> lCCAA = null;

    //constructor
    public DataBindingParser(String sFile) throws Exception {
        Gson oGson = new Gson();
        oRegAdmin = oGson.fromJson(new InputStreamReader(new FileInputStream(sFile), "UTF-8"), RegAdmin.class);
    }

    //método público a implementar
    public List<String> buscarCCAAPorTema(String sTema) {
        lCCAA = new ArrayList<String>();

        // Comunidades Autónomas
        parseCCAA(oRegAdmin.getCCAA(), sTema);
        return lCCAA;
    }

    //método privado para el procesamiento de una CA
    private void parseCCAA(List<ComunidadAutonoma> oListCCAA, String sTema) {
        for(ComunidadAutonoma oCCAA : oListCCAA) {
            // Provincias
            if (parsePR(oCCAA.getProvincias(), sTema)) {
                //no es necesario seguir procesando el resto de provincias
                lCCAA.add(oCCAA.getNombre() );
            }
        }
    }

    //método privado para el procesamiento de una PR
    private boolean parsePR(List<Provincia> oListPR, String sTema) {
        for(Provincia oPR : oListPR) {
            // Temas
            if (oPR.getTemas()!=null) {
                for(Tema oTema : oPR.getTemas()) {
                    if(oTema.getNombre().equalsIgnoreCase(sTema)) {
                        //no es necesario seguir procesando el resto de temas
                        return true;
                    }
                }
            }
        }
        return false;
    }

    public static void main(String[] args) {
        ----------------------------
    }
}
```

Código 19 - Clase piat.examples.regadmin.DataBindingParser

## 3.2 Procesando el flujo de datos de entrada: streaming model

Con objeto de que las tres soluciones sean comparables, el diseño de este tercer analizador se ha abordado manteniendo como requisito que la invocación al método **`buscarCCAAPorTema()`** desencadene el proceso de búsqueda por la información asociada al documento JSON y, cree y devuelva una lista con el nombre de la comunidades autónomas pertinentes.

```java
public class StreamingParser {

    private String sJSONFile;
    private JsonReader oReader;
    private ArrayList<String> lCCAA;

    public StreamingParser(String sFile) throws Exception {
        sJSONFile = sFile;
    }

    public List<String> buscarCCAAPorTema(String sTema) throws Exception {
        lCCAA = new ArrayList<String>();
        String sNombrePropiedad;
        //procesamiento del contenido del documento como una secuencia de token. Cada token es identificado, consumido y liberado.
        // es necesario iniciar el procesamiento por cada tema sometido a búsqueda

        oReader = new JsonReader(new InputStreamReader(new FileInputStream(sJSONFile), "UTF-8"));

        //procesamiento del objeto root
        oReader.beginObject();
        while (oReader.hasNext()) {
            sNombrePropiedad = oReader.nextName();
            if (sNombrePropiedad.equals("ccaa")) {
                // Comunidades Autónomas
                parseCCAA(sTema);
            } else {
                //se consume cada token que no require un procesamiento
                oReader.skipValue();
            }
        }

        oReader.endObject();
        oReader.close();

        return lCCAA;
    }

    //----------- métodos privados

    public static void main(String[] args) {
        // ---------------------
    }
```

Código 20 – Clase piat.examples.regadmin StreamingParser

En los dos analizadores basados en objetos Java, el constructor se encargaba de almacenar la información del documento en objetos Java y el método de búsqueda recorría esa información para conformar la lista de resultados. Sin embargo, en el modelo de `streaming`, en cada invocación a este método se debe `iniciar` el procesamiento del flujo de entrada, `consumir secuencialmente` cada token y `cerrar` el flujo una vez finalizado el procesamiento.

```java
public List<String> buscarCCAAPorTema(String sTema) throws Exception {
    lCCAA = new ArrayList<String>();
    String sNombrePropiedad;

    oReader = new JsonReader(new InputStreamReader(new FileInputStream(sJSONFile), "UTF-8"));
```

Código 21 – Inicialización del procesamiento del flujo de entrada en cada iteración

En esos analizadores, cuando una CA era pertinente se utilizaban los métodos adecuados de los objetos correspondientes para obtener la información necesaria de la comunidad autónoma (en nuestro caso el `nombre`) y añadirlo a la lista de resultados:

```
lCCAA.add(oCCAA.getAsJsonObject().get("nombre").getAsString());
```

Código 22 - Obtención del nombre de una CA en el modelo de objetos

```
lCCAA.add(oCCAA.getNombre());
```

Código 23 - Obtención del nombre de una CA en el modelo Data binding

En este modelo debemos tener en cuenta:

- No se pueden hacer suposiciones sobre el orden de las propiedades de un objeto.
- El objeto **`JsonReader`** realiza un procesamiento no orientado a estados (**`stateless`**), es decir, el procesamiento de un estado es independiente de estados pasados y de estados futuros. Cada `token` es identificado y consumido de forma independiente.
- Es responsabilidad de la aplicación que hace uso del **`JsonReader`**, el mantener la información que pueda ser necesaria, almacenándola en objetos propios en el momento en el que aparezca el `token` que la contiene, aun cuando con posterioridad sean descartados.
- Los elementos que no sean relevantes para la aplicación, deberán ser consumidos, aunque no sea necesario su procesamiento.

Trasladando estas consideraciones a nuestro analizador, el diseño del algoritmo de procesamiento de una CA deberá contemplar lo siguiente:

- No se puede suponer que el `nombre` de una CA se consume antes que el array de `provincias` asociadas.
- Si una vez procesadas las `provincias` se tiene la certeza de que la CA es pertinente, no se puede solicitar al **`JsonParser`** que nos proporcione el nombre de la CA (como se hacía en el modelo de objetos). Se podrá haber extraído en un estado anterior o se podrá extraer en un estado futuro.
- Sólo cuando se terminen de consumir todos los pares clave-valor asociados a un objeto tendremos la seguridad de que la información necesaria ha sido extraída y/o calculada.

```
//método privado para el procesamiento de una CA
private void parseCCAA(String sTema) throws Exception{
    String sNombreCA = "";
    Boolean bEsPertinente = false;
    String sNombrePropiedad;

    oReader.beginArray();
    while(oReader.hasNext()) {
        oReader.beginObject();
            sNombreCA="";
            while(oReader.hasNext()) {
                sNombrePropiedad = oReader.nextName();
                if(sNombrePropiedad.equals("nombre")){
                    sNombreCA = oReader.nextString().trim();
                }else if (sNombrePropiedad.equals("provincias")) {
                    // solo se pueden tomar decisiones una vez que todos el objeto hayan sido procesado
                    bEsPertinente = parsePR(sTema);
                }else {
                    //se consume cada token que no require un procesamiento
                    oReader.skipValue();
                }
            }
            //una vez extraída toda la información, se pueden tomar decisiones
            if (bEsPertinente)
                lCCAA.add(sNombreCA);
        oReader.endObject();
    }
    oReader.endArray();
}
```

Código 24 - Procesamiento de las CCAA mediante Streaming

Para el diseño del algoritmo de procesamiento de las provincias de una CA se han aplicado consideraciones equivalentes y se ha mantenido la funcionalidad de no seguir evaluando la condición de búsqueda en el resto de provincias cuando una de ellas satisface el criterio de búsqueda.

```
    //método privado para el procesamiento de una PR
    private boolean parsePR( String sTema) throws Exception{
        Boolean bEsPertinente = false;
        String sNombrePropiedad;

        oReader.beginArray();
        while(oReader.hasNext()) {
            if (bEsPertinente)
                oReader.skipValue();
            else {
                oReader.beginObject();
                while(oReader.hasNext()) {
                    sNombrePropiedad = oReader.nextName();
                    if (sNombrePropiedad.equals("temas")) {
                        bEsPertinente = parseTemas(sTema);
                    }else {
                        //se consume cada token que no require un procesamiento
                        oReader.skipValue();
                    }
                }
                oReader.endObject();
            }
        }
        oReader.endArray();

        return bEsPertinente;
    }
```

Código 25 - Procesamiento de la provincia de una CA mediante Streaming

Es importante destacar que, en los dos analizadores basados en objetos, una vez identificada la evidencia se suspendía el procesamiento del resto de provincias. En este modelo, no se puede 'abortar' el análisis del resto de provincias, ya que es necesario consumir secuencialmente todos los tokens. El algoritmo propuesto, una vez que sabe que una CA es pertinente, consume los tokens, pero no procesa el resto de provincias.

El diseño del algoritmo del método que analiza los temas de una provincia, es equivalente al realizado para el método **`parsePR()`**. Manteniendo la optimización ante la identificación de un tema pertinente.

```
//método privado para el procesamiento de un tema
private boolean parseTemas( String sTema) throws Exception{
    Boolean bEsPertinente = false;
    String sNombrePropiedad;

    oReader.beginArray();
    while(oReader.hasNext()) {
        if (bEsPertinente)
            oReader.skipValue();
        else {
            oReader.beginObject();
            while(oReader.hasNext()) {
                sNombrePropiedad = oReader.nextName();
                if (sNombrePropiedad.equals("nombre")) {
                    bEsPertinente = oReader.nextString().equalsIgnoreCase(sTema);
                }else {
                    //se consume cada token que no require un procesamiento
                    oReader.skipValue();
                }
            }
            oReader.endObject();
        }
    }
    oReader.endArray();

    return bEsPertinente;
}
```

Código 26 - Procesamiento de la provincia de un Tema mediante Streaming

Si bien este modelo de procesamiento es el que menor memoria consume y el que mejor tiempo de respuesta tiene, puede requerir de diseños de algoritmos optimizados para evitar consumir varias veces el flujo de datos de entrada, ya que en ocasiones puede ocurrir que un flujo una vez producido y consumido desaparezca (y por tanto no puede volver a solicitarse) o que por optimización de recursos (ancho de banda, rendimiento del servidor que proporciona la información, …) no sea adecuado solicitar el recurso varias veces.

En nuestro ejemplo una posible forma de optimización, conocida la funcionalidad esperada, consistiría en que el constructor lance un proceso de análisis que tenga como resultado la creación de un mapa en memoria en el que a cada tema se le asocian las comunidades autónomas. El método de búsqueda sólo debería devolver la información asociada a la clave del mapa, sin necesidad de volver a abrir el flujo de datos y de procesarlo.