voiceBrowser:
un navegador Web para Invidentes

Tecnologías Multimedia en Java: Síntesis y Reconocimiento de Voz

Vicente Miguel Ferrer Gilabert
Sistemas Multimedia

Ingeniería Técnica de Telecomunicaciones, Telemática
Escola Tècnica Superior D´Enyinyeria
Universitat de València

ÍNDICE

NAVEGADOR WEB PARA INVIDENTES

1.Motivación: Multimedia como método de ayuda a discapacitados

Paralelamente, en los últimos años se han producido una serie de importantes avances tecnológicos que han cambiado y todavía siguen cambiando radicalmente la forma de comunicación. Uno de estos avances ha sido Internet que ha hecho reescribir la evolución de las Tecnologías de la Información y la Comunicación. Otro de los avances, cuya expansión es coetánea a la explosión de Internet ha sido la Multimedia. De hecho, casi desde el principio, ambas tecnologías han estado íntimamente relacionadas.

Internet es la mayor base de datos que la humanidad ha tenido a su disposición en la historia. Internet nació y se mantiene hoy como un medio de presentación y acceso a la información pensada para todos; de una manera sencilla se puede acceder a cualquier tipo de información, localizada en cualquier parte del planeta y en medios heterogéneos, mediante un conjunto de herramientas de sencillo uso.

Sin embargo y como ocurre muchas veces, realmente estas tecnologías no están al alcance de **todos**. Las barreras arquitectónicas que un discapacitado se puede encontrar en su desplazamiento por el mundo real, también existen, incluso de mayor tamaño y más difícilmente salvables, en el mundo de Internet. No podemos permitir que las personas que padecen algún tipo de discapacidad física, no tengan la posibilidad de acceder al mayor compendio de información que la humanidad ha visto nunca. La tecnología debe ponerse al alcance de todos y, para salvar las dificultades que esto pueda acarrear, se debe utilizar la propia tecnología.

Hasta la llegada de *2001: Una odisea del Espacio* premiada por su computadora parlante, HAL, el público ha estado esperando que los sistemas guiados por voz se convirtieran en realidad. Quién puede olvidar el ordenador de *Juegos de Guerra* diciendo "¿Quieres jugar a un juego?" Casi cuarenta años de investigación y grandes cantidades de dinero invertidas nos permiten, hoy en día, poder disponer de sistemas de reconocimiento y síntesis de voz en ordenadores domésticos, así como implementaciones para los lenguajes de programación más comunes, que acercan las tecnologías de guiado por voz a sus potenciales usuarios.

En la práctica, no todo el mundo puede navegar por Internet. De hecho, no todo el mundo puede utilizar un ordenador fácilmente. Las personas que padecen algunos tipos de discapacidad física tienen verdaderos problemas para el uso de un ordenador en general y de Internet en concreto. Sin embargo, aquí es donde entran en juego las tecnologías multimedia. Como ejemplo, un discapacitado motriz podría tener verdaderos problemas para utilizar un periférico tan simple y a la vez tan necesario en las actuales interfaces gráficas de usuario, como es el ratón. O en el caso que nos ocupa, el uso de un ordenador por personas con discapacidades visuales puede ser realmente difícil. Y en concreto, el acceso a Internet a través del Web, por ejemplo, puede ser tarea casi imposible para un ciego o discapacitado visual; Sin embargo, aquí es donde la multimedia, entendida en esta ocasión como **integración de medios** puede ayudar a paliar esta situación. Este discapacitado visual puede utilizar tecnologías de síntesis y reconocimiento de voz que le ayuden a "conocer" qué es lo que hay en cada página web y a decidir qué nueva página quiere visitar en cada momento, de manera similar a como lo hacemos las personas que no tenemos mermadas nuestras capacidades visuales.

La idea inicial de mi trabajo es precisamente esa, la de la utilización de la *Multimedia no como una manera de mostrar contenido de forma enriquecida, sino como una forma de utilizar la combinación de diferentes medios digitales para facilitar a personas con discapacidad visual la navegación por Internet*. Internet ofrece nuevas posibilidades para el acceso a la información, pero a veces el diseño de las páginas web obstruye los contenidos y las hacen inaccesibles, especialmente para las personas con discapacidades visuales.

El problema tiene varias vertientes. Por un lado, el propio diseño de las páginas web, que las hacen inaccesibles o simplemente reducen su usabilidad al absurdo para personas invidentes. Por el otro, los navegadores usados para navegar por la red están pensados para usuarios sin discapacidades visuales.

Desde su creación, Internet ha ofrecido una amplia gama de posibilidades para el acceso a la información. A través de este medio, los datos publicados pueden estar disponibles inmediatamente y a un mismo tiempo desde cualquier parte del mundo. Pero algunas veces el diseño de las páginas y aplicaciones web no son accesibles para todo el mundo. Las personas con discapacidad a menudo son incapaces o encuentran verdaderas dificultades cuando intentan recuperar información de la red. El diseño de las páginas Web ha ido sustituyendo e incluso eliminando la funcionalidad y capacidad de comunicación de información de las páginas. Se ha sustituido la información por estética, dificultando el acceso a los contenidos, especialmente para aquellas personas cuyas discapacidades físicas les impide disfrutar con el diseño: los invidentes.

voiceBrowser es un proyecto que pretende facilitar la navegación por Internet de personas con discapacidades visuales. Está programado en Java utilizando el reconocimiento y síntesis de voz que ofrece la librería JSAPI (Java Speech API) utilizando la implementación proporcionada por IBM en su producto Via Voice. Así mismo, para la reproducción de determinados clips de audio utiliza la librería de Sun JMF (Java Media Framework) y para el parsing HTML utiliza la librería javax.swing.text.html.

2.Tecnologías Reconocimiento y Síntesis de Voz: JSAPI y MSAPI

Hace unos años y con el aumento de la capacidad de computación de los ordenadores, surgieron los primeros programas de reconocimiento de voz. Inicialmente la capacidad de reconocimiento de voz de estos programas era bastante reducida, la fiabilidad del sistema no era demasiado alta; sin embargo, hoy en día la fiabilidad es más que suficiente para muchas aplicaciones. Aún así y lejos de lo que pudieran pensar los tecnógrafos hace unos años, todavía hoy, las aplicaciones y sistemas de reconocimiento de voz no son habituales y mucho menos, estándares, en los programas informáticos.

Otro de los aspectos que dificultan el despegue de las tecnologías de reconocimiento del habla es la ausencia de una interfaz de programación (API) estándar, reconocida realmente por toda la industria. El uso de estándares permitiría que los motores de voz de distintos fabricantes puedan trabajar con aplicaciones de otras empresas, de modo que los desarrolladores pueden centrarse en lo que debe hacer su aplicación.

En la actualidad, los modelos de interfaz de reconocimiento de voz más extendidos son JSAPI *(Java Speech API),* de Sun Microsytems; SRAPI *(Speech Recognition API),* de Novell y SAPI *(Speech API),* de la propia Microsoft.

2.1. **JSAPI:** Sun Microsystems definió una librería de programación que proporciona funcionalidad de síntesis y reconocimiento de voz a su lenguaje de programación Java. Sin embargo, Sun Microsystems no es especialista en aplicaciones de reconocimiento de voz, y tampoco tenía la intención de invertir los recursos necesarios en desarrollar un motor de reconocimiento. Así que proporcionó tan solo una interfaz, una estructura de clases, interfaces y métodos y unas reglas de funcionamiento para su API de reconocimiento de voz, dejando las implementaciones reales en manos de terceras empresas. Esta es, precisamente, la idea del JSAPI (página web de JSAPI) Podemos destacar dos consecuencias positivas de esta "descentralización" de la especificación del API y la implementación de sus métodos:

- Por un lado, cualquier motor de voz que provea los servicios que requiere el *Java Speech API* podrá ser utilizado en cualquier tipo de aplicación desarrollada en Java. Esto no sólo permite desarrollar aplicaciones de escritorio que implementen métodos de reconocimiento y síntesis de voz, sino que permite que otro tipo de aplicaciones, distribuídas o ejecutadas a través de Internet, por ejemplo, utilicen herramientas de reconocimiento de voz de la misma forma que se haría si la aplicación se estuviera ejecutando *stand alone*.
- Por otro lado, cualquier desarrollador, empresa o comunidad interesada en el desarrollo de un motor de reconocimiento de voz tiene a su disposición la posibilidad de dotarlo de un funcionamiento estándar, de un interfaz que permitirá que su motor de voz se integre con cualquier aplicación que ya exista o se diseñe después que implemente el *Java Speech API*.

Entendido de esta manera, el JSAPI puede verse como un intento de Sun Microsystems, no sólo de dotar a su lenguaje de funcionalidades de reconocimiento de voz sin emplear grandes cantidades de recursos en el desarrollo de un motor de voz, sino que el verdadero éxito del JSAPI es su voluntad de "estándar", un sólo interfaz de programación que pueda *atacar* a cualquier motor de voz. Esta es precisamente la intención subyacente en cualquier desarrollo de software moderno y especialmente, en cualquier desarrollo Java: la reutilización de código y esfuerzos.

El *Java Speech API* es un conjunto de clases abstractas e interfaces que representan un motor de voz para un programador Java, pero que no asume la implementación de las tareas de reconocimiento y síntesis. De esta manera, el motor puede estar implementado como una solución hardware o software (o incluso, una combinación de ambos) Incluso se puede dar la situación de que diferentes implementaciones de motores pueden tener diferentes características; de hecho, algunos pueden tener la capacidad de "aprender" acerca de los patrones del habla de un usuario y otros, pueden elegir no implementar esta funcionalidad.

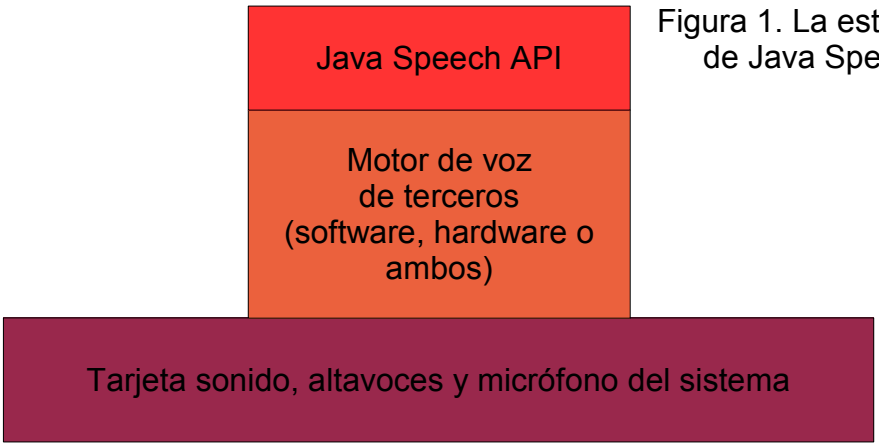


Figura 1. La estructura de Java Speech

Sun Microsystems facilita las implementaciones válidas de terceras partes de su *Java Speech API* en la página web del proyecto, en http://java.sun.com/products/java-media/speech

- FreeTTS: Este es un sintetizador de voz de código abierto escrito completamente en Java.
- IBM Speech for Java: Esta es una implementación de IBM basada en su motor de reconocimiento del habla Via Voice. Se debe adquirir una copia del software Via Voice para poder utilizarlo.
- The Cloud Garden: Esta implementación puede funcionar con cualquier motor de voz basado en el Microsoft Speech API.
- Lernout and Hauspie´s TTS for Java Speech API: Este paquete funciona solamente en máquinas Sun y proporcina un gran número de características avanzadas.
- Festival: Producto escocés basado en Unix. Soporta un gran número de interfaces además de la interfaz basada en Java.

Mediante estas herramientas, la programación de aplicaciones de reconocimiento de voz, se facilita enormemente. Las tareas más complejas del reconocimiento y la síntesis de voz desde el punto de vista de la ingeniería, son las transformaciones de la señal analógica recogida por un micrófono en palabras del lenguaje; o bien al revés, la transformación de texto escrito en voz sintetizada. Con JSAPI desaparece toda la dificultad subyacente al motor de reconocimiento, pues la especificación de la Interfaz de programación y del motor de reconocimiento en sí, son cosas diferentes.

Mediante esta implementación, el API de JavaSpeech es tan solo una capa que descansa sobre el procesado de voz que provee una tercera empresa, en este caso, IBM gracias a su producto ViaVoice. Mediante una capa de adaptación, Via Voice ofrece sus servicios de voz a JavaSpeech API. Por último, será IBM ViaVoice quien se encargará de comunicarse con la tarjeta de sonido a través de los controladores instalados en el sistema. Será el propio IBM ViaVoice quien recogerá la entrada de audio del micrófono y enviará la salida a los altavoces a través de la tarjeta de sonido.

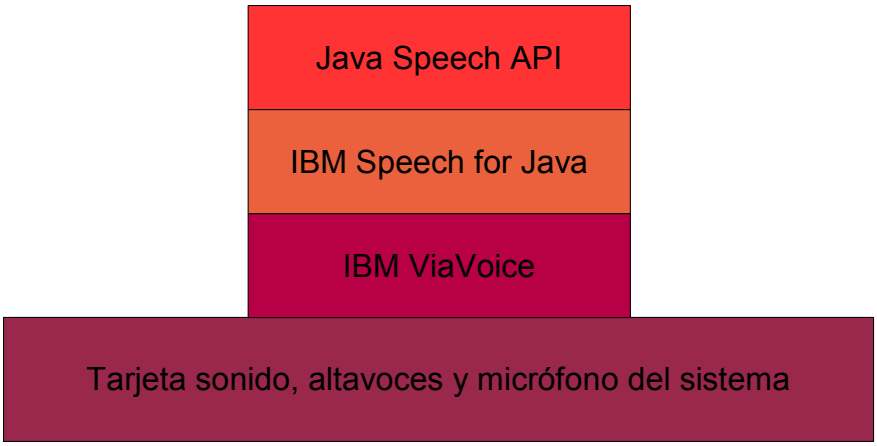


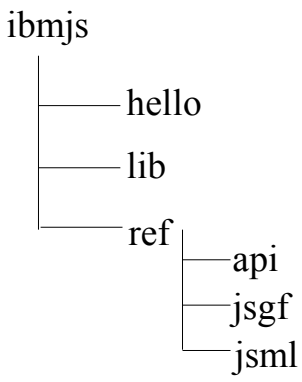
Figura 2. La estructura de IBM Speech for Java

3.El paquete IBM Speech for Java.

Para comenzar, necesitamos un Motor de Voz que provea funciones de Síntesis y de Reconocimiento. Existen productos basados en Software Libre que proveen implementaciones de las funciones de síntesis de voz del JavaSpeech API, pero ninguno de los examinados en el momento de escribir este documento, provee de funciones de reconocimiento de voz. Así pues, no nos queda más remedio que utilizar el *IBM ViaVoice for Java* como motor de voz. Nótese que la estructura de bloques del JSAPI de Sun, permite sustituir fácilmente el motor de voz por otro diferente, con lo cual, si en algún momento surgiera alguna implementación libre de funcionalidad completa, podría sustituirse sin dificultad.

Así pues, los pasos a seguir para dotar de acceso al motor de voz de IBM por parte del JSAPI, son los siguientes:

1. Instalar ViaVoice de acuerdo con las instrucciones que incluye este producto.
2. Descargar e instalar el paquete *IBM Speech for Java*. Se puede descargar de manera gratuíta desde la web http://alphaworks.ibm.com/tech/speech.Este paquete incluye básicamente, tres directorios:



El directorio *hello* contiene un ejemplo de uso de IBM Speech for Java. El directorio *ref* como su nombre indica es un directorio de referencia y contiene la documentación del paquete (incluído el API en formato JavaDoc.

Por último, el directorio que realmente nos interesa es *lib*. Este directorio contiene las librerías y el jar de la implementación de IBM Speech for Java. Deberemos añadir este directorio a la variable PATH del sistema y el fichero *ibmjs.jar* a la variable CLASSPATH.

Echemos un vistazo ahora al propio API de JavaSpeech para familiarizarnos con su estructura. El JSAPI está compuesto, fundamentalmente, de tres paquetes diferentes:

- **javax.speech:** Interfaces y clases abstractas utilizadas fundamentalmente para declarar y crear un motor de voz, así como para controlar eventos asociados con la entrada y salida de audio, control del volumen de captura o reproducción, etc.
- **javax.speech.recognition:** Como su nombre indica, provee las clases e interfaces para declarar un motor de reconocimiento de voz y trabajar con él.
- **javax.speech.synthesis:** También en este caso, provee de clases e interfaces para declarar un motor de síntesis de voz y operar con él.

4.Declaración de un motor de voz.

La declaración de un motor de voz dependerá de la acción que queramos realizar: síntesis o reconocimiento de voz. En ambos casos se debe utilizar la clase *Central* del paquete *javax.speech* y que provee, mediante el patrón *Factoría*, sendos métodos que devuelven bien un objeto de tipo *Recognizer*, bien un objeto de tipo *Synthesizer*:

`static Recognizer createRecognizer(EngineModeDesc require)`	
`static Synthesizer createSynthesizer(EngineModeDesc require)`	

En ambos casos, una invocación del tipo:

```
RecognizerModeDesc require = new RecognizerModeDesc(new Locale("es"));
Recognizer rec = Central.createRecognizer(EngineModeDesc require);
rec.allocate();
rec.resume();

SynthesizerModeDesc require = new SynthesizerModeDesc(new Locale("es"));
Synthesizer synth = Central.createSynthesizer(EngineModeDesc require);
synth.allocate();
synth.resume();
```

Devuelve un objeto de reconocimiento y de síntesis de voz, respectivamente. Tanto Recognizer como Synthesizer son interfaces, que se asocian con el objeto que devuelve Central.createRecongnizer y `Central.createSynthesizer`. Ambos métodos `static` reciben un parámetro de tipo `EngineModeDesc`, una interface que representa las características generales del motor de voz. Si queremos modificar alguna característica concreta del funcionamiento de un motor de reconocimiento o de un motor de síntesis de voz, existen sendas clases que extiende a `EngineModeDesc` y que implementan características concretas de síntesis/reconocimiento y que son `SynthesizerModeDesc` y `RecognizerModeDesc`. De hecho, podemos modificar algunas características interesantes del reconocimiento o síntesis de voz mediante estos objetos `XXXModeDesc`. Por ejemplo:

`public void addVoice(Voice v);`
`Voice(String name, int gender, int age, String style);`

Se puede utilizar para declarar un motor de Síntesis de Voz que imite a una persona de determinado sexo y edad, por ejemplo:

```
require.addVoice(new Voice(null, Voice.GENDER_MALE,
                 Voice.AGE_YOUNGER_ADULT, null));
```

El IBM Speech for Java proporciona implementaciones diferentes para sexo masculino y femeino (valores `Voice.GENDER_MALE` y `Voice.GENDER_FEMALE`) así como para un niño (`Voice.AGE_CHILD`), un adulto (`Voice.AGE_YOUNGER_ADULT`) o un anciano (`Voice.AGE_OLDER_ADULT`)

Estas clases también determinan la lengua en que trabajará el sintetizador o reconocedor de voz. Esto se hace, por ejemplo, mediante un constructor de `SynthesizerModeDesc` o `RecognizerModeDesc` que recibe un objeto de tipo `Locale`. Si en el momento de la obtención de un objeto reconocedor o sintetizador de voz, no se especifica la lengua de trabajo, Java tratará de generar un `Recongnizer` o Synthesizer en la lengua por defecto del sistema, que es el objeto de tipo Locale que devuelve la llamada a la función java.util.Locale.getDefault().

Por cuestiones de diseño del JavaSpeech API los objetos de tipo `Recongnizer` o Synthesizer no están inicialmente listos para funcionar, sino que se deben reservar recursos y posteriormente activar para poder funcionar. Los motores no son activados por defecto, por dos motivos: el primer motivo es que la creación de un motor es una tarea de alto coste computacional; para mejorar el rendimiento, el API de JavaSpeech permite activar el motor en un hilo diferente mientras el programa principal realiza otras tareas. El segundo motivo es que el motor necesita acceso exclusivo a algunos dispositivos, como el micrófono (para tareas de reconocimiento); este diseño del API permite declarar el motor y activarlo o desactivarlo en tiempo de ejecucion cuando, por ejemplo, se requiriera el acceso a algún dispositivo que utiliza el motor. Las operaciones de activación y desactivación pueden realizarse, sobre un objeto de tipo `Engine`, mediante los métodos:

`public void` **`allocate`**`()`
`throws EngineException,`
`EngineStateError`
	Reserva los recursos necesarios para el motor de reconocimiento o síntesis de voz.

`public void` **`deallocate`**`()`
`throws EngineException,`
`EngineStateError`
	Desreserva los recursos utilizados por el motor de reconocimiento o síntesis de voz.

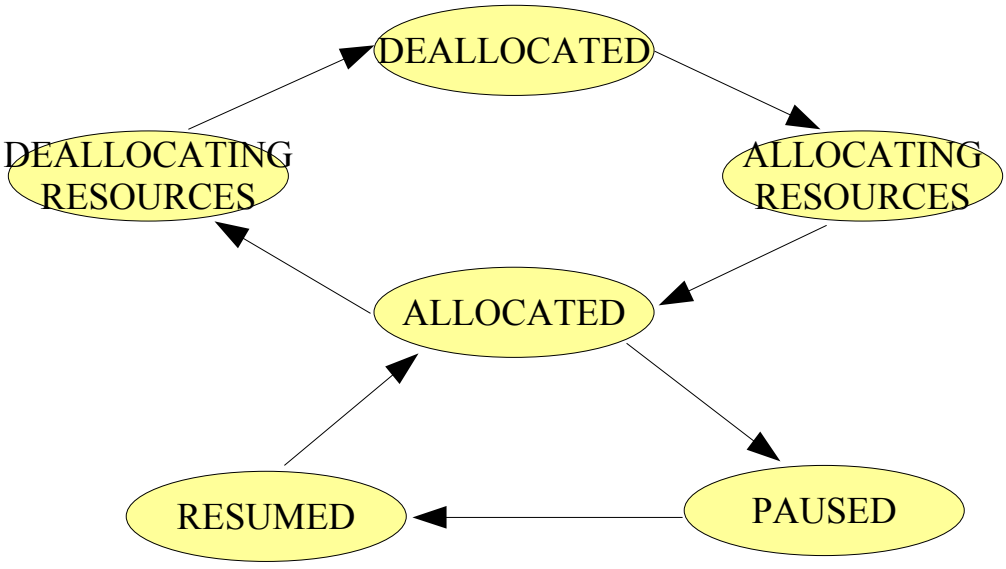


Figura 3. Posibles estados de un motor de voz

Para terminar con este apartado, otras funciones interesantes que se han de conocer para trabajar con motores de voz están relacionadas con el concepto de *estado del motor.* Durante el ciclo de ejecución de un motor de voz, éste puede encontrarse en diferentes estados: activado, desactivado, pausado, funcionando, activando recursos o desactivando recursos.

Estos diferentes estados están declarados en la interface `javax.speech.Engine`. Los estados descritos hasta el momento son comunes a los motores de reconocimiento y de síntesis. Además de estos estados, existen determinados sub-estados propios de cada motor de reconocimiento; por ejemplo, un **motor de síntesis** que está en estado `ALLOCATED`, puede estar a su vez en uno de los siguientes estados: `QUEUE_EMPTY` o `QUEUE_NOT_EMPTY`, indicando si todavía le queda audio en la cola de reproducción o no. Un **motor de reconocimiento** de voz que se encuentra en estado `ALLOCATED` puede a su vez estar en uno de los tres sub-estados: `LISTENING`, `PROCESSING` o `SUSPENDED`, indicando que el motor está esperando, procesando o suspendido.

La interface `javax.speech.Engine`, permite precisamente reservar recursos y desreservar recursos, pausar o activar un motor, así como acceder y modificar las características del motor mediante una referencia a `EngineProperties` o del dispositivo de audio mediante la obtención de una referencia a la clase `AudioManager`. Tanto los objetos Recongnizer como Sinthesizer extienden la interface Engine y, por tanto, proporcionan una implementación para todos los métodos que permiten allocalte y deallocate o pausar y resume un motor de voz, además de implementar los métodos correspondientes a los estados específicos para un motor de reconocimiento o para un motor de síntesis de voz.

El correcto manejo del estado del motor en nuestra aplicación es muy importante. Así, por ejemplo, si tenemos un motor de síntesis de voz que está sintetizando un determinado texto en un instante de tiempo determinado, deberíamos esperar a que esta operación terminara para pedirle que sintetizara un nuevo texto. Para ello, tendríamos que manejar el estado del motor:

```
synth.waitEngineState(Engine.ALLOCATED && Synthesizer.QUEUE_EMPTY);
```

5.Síntesis de voz.

Ahora que ya estamos familiarizados con el motor de voz, podemos pasar a abordar la parte de reconocimiento de voz. Veremos cómo implementar funciones básicas de síntesis de voz, y también alguna característica más avanzada como los mecanismos para modificar la pronunciación de determinadas frases mediante el JSML (Java Speech Markup Language) que también proporciona el API de JavaSpeech.

Veamos un ejemplo extraído directamente del código fuente del programa voiceBrowser. El método public static void sintesis(String texto) sintetiza en castellano el texto pasado como parámetro.

```
public static void sintesis(String texto) {
            try {
                  //Crear un objeto SynthesizerModeDesc en español con
                        //determinadas características de voz
                  SynthesizerModeDesc modeDesc = new SynthesizerModeDesc();
                  modeDesc.addVoice(new Voice(null, Voice.GENDER_MALE,
                                    Voice.AGE_YOUNGER_ADULT, null));
                  modeDesc.setLocale(new Locale("es"));

                  //Obtener mediante patrón Factoría, una referencia a un
                        //motor de síntesis de voz
                  Synthesizer synth = Central.createSynthesizer(modeDesc);

                  //Reservar los recursos necesarios para el motor y activarlo
                  synth.allocate();
                  synth.resume();

                  //Sintetizar el texto en modo Texto Plano (sin considerar
                        //tags de JSML)
                  synth.speakPlainText(texto, null);

                  //Esperar hasta que la síntesis ha terminado
                  synth.waitEngineState(Synthesizer.QUEUE_EMPTY);

                  //Desreservar los recursos empleados por el motor
                  synth.deallocate();

            } catch (IllegalArgumentException illegalArgument) {
                  illegalArgument.printStackTrace();
            } catch (EngineException engineException) {
                  engineException.printStackTrace();
            } catch (EngineStateError stateError) {
                  stateError.printStackTrace();
            } catch (AudioException audioException) {
                  audioException.printStackTrace();
            }
      }
```

El código presenta dos partes: una primera parte en la que se declara el motor de síntesis de voz, en español, con voz masculina, de adulto joven; en la segunda parte se produce la síntesis en sí del texto, mediante el comando:

```
synth.speakPlainText(texto, null);
```

Cuya declaración sería:

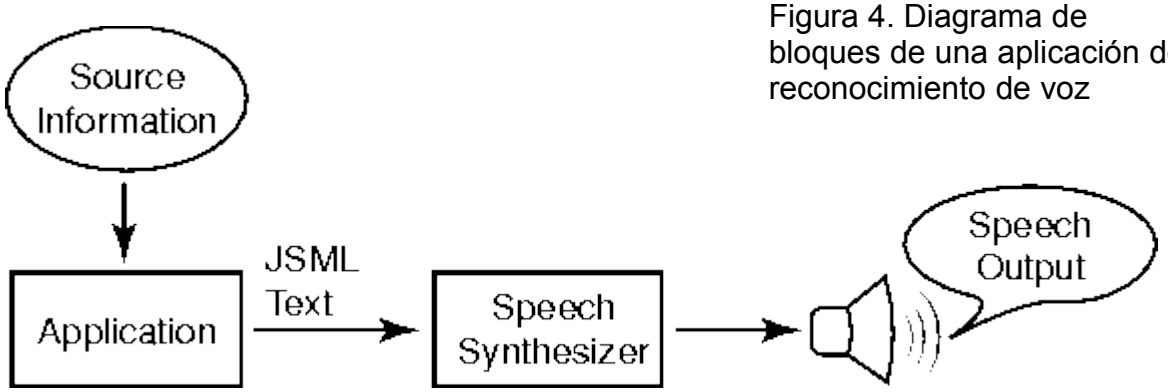
```
public void speakPlainText(String text,
                           SpeakableListener listener)
                    throws EngineStateError
```

De una manera sencilla se cubren gran parte de las necesidades de síntesis que se pueden tener en una aplicación. Sin embargo, cuando empezamos a hacer uso de las funciones de síntesis de voz del JSAPI, descubrimos que se requieren ciertas funcionalidades extras que aumenten el control sobre la forma en que se sintetiza la voz. Utilizando los métodos anteriormente descritos, parece que cualquier texto es sintetizado de misma forma; sin embargo, en el habla real no es así. Cuando hablamos, utilizamos diferentes entonaciones para dar mayor énfasis o valor a una palabra o frase concretas; cuando escribimos, utilizamos abreviaturas o siglas, que no siempre deben ser

deletreadas cuando sean leídas; utilizamos fechas, cantidades numéricas, ... Para cubrir estas necesidades, el API de JavaSpeech presenta un sub-API que define un lenguaje de marcas para manipular el modo en que los textos serán sintetizados por el motor de voz.

5.1.**JSML (Java Speech Markup Language)**: El JSML permite definir la forma en que el texto será sintetizado. Su sintaxis se basa en el estándar SGML que es el estándar general para los lenguajes de marcas. De hecho, JSML es un subconjunto de XML. Por tanto, su sintaxis se parece mucho a la del propio HTML y es fácil de aprender y utilizar.

Figura 4. Diagrama de bloques de una aplicación de reconocimiento de voz



Los lenguajes de especificación del habla no son una idea nueva. Existieron otros intentos similares antes de JSML y de hecho, éste está en parte basado en lenguajes anteriores. Por ejemplo, los laboratorios Bell junto con la Universidad de Edimburgo desarrollaron juntos un lenguaje de especificación de síntesis llamado STML, que sirvió de base para gran parte de la especificación del JSML. Sin embargo, JSML no ha conseguido ser aceptado como estándar en el sector. Por ello, Sun ha anunciado el desarrollo de un nuevo estándar de especificación de síntesis llamado, en clave, SABLE, cuyos datos técnicos no han sido publicados todavía. SABLE combinará características del JSML de Sun y del STML de los laboratorios Bell y pretende ser el estándar en el futuro.

Cualquier documento JSML debe seguir una sintaxis predeterminada. La cabecera que debe utilizar cualquier documento JSML está directamente heredada de XML. Así mismo, el documento está enmarcado entre dos tags `<JSML>` y `</JSML>`

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

y terminaría con:

```
</JSML>
```

Un documento JSML debe ser un documento XML bien formado, lo cual significa que, además de poder editarlo con cualquier programa de escritura XML, podemos también comprobar su integridad con las mismas herramientas utilizadas en XML. Los elementos JSML están divididos en tres grupos:

- Elementos estructurales: Sirven para especificar párrafos y sentencias.
- Elementos de Producción: Especifican propiedades prosódicas, pausas, énfasis y otra información concerniente a la pronunciación de palabras o porciones de palabras.
- Elementos variados: Incluyen marcadores especiales que pueden ser usados para proporcionar instrucciones nativas al propio motor de voz.

Tipo de elemento	Nombre del elemento	Descripción
Estructural	PARA	Especifica que el texto contenido es un párrafo.
	SENT	Especifica que el texto contenido es una sentencia.
	<PARA> <SENT>Ana fue a escuela y aprobó los exámenes.</SENT> <SENT>Cuando volvió a casa, lucía el Sol.</SENT> </PARA>	
Producción	SAYAS	Especifica cómo se debe pronunciar determinado texto. <SAYAS SUB="I E cubo">IEEE</SAYAS> <SAYAS CLASS="date">Enero 1952</SAYAS> <!--pronunciado Enero de mil novecientos cincuenta y dos --> <SAYAS CLASS="literal">JSML</SAYAS> <!-pronunciado como J S M L -->
	EMP	Especifica que hay que aplicar cierto tipo de énfasis al texto contenido. Llévate <EMP LEVEL="moderate">tu</EMP> coche.
	BREAK	Especifica cierto tipo de pausa en la síntesis. <BREAK SIZE="small" MARK="145"/> <BREAK MSECS="300"/>
	PROS	Especifica una propiedad de pronunciación, como la velocidad del habla, el tono o el volumen del texto <PROS RATE="-30%"><SAYAS SUB="sun punto com">sun.com</SAYAS></PROS> <!-pronunciado como sun punto com un 30% más despacio --> <PROS RATE="150">Texto a 150 palabras por minuto</PROS>
Varios	MARKER	Proporciona una notificación a la línea de ejecución de la aplicación Java cuando el motor de síntesis alcanza la posición en que se encuentra el tag. Pregunta <MARKER MARK="yes_no_prompt"/> sí o no
	ENGINE	Para proporcionar instrucciones nativas al motor de síntesis utilizado. Me convierto <ENGINE ENGID="Acme Voice"> en otro </ENGINE> y vuelvo a ser yo <!- Texto pronunciado por otra voz -->

6.Reconocimiento de voz.

La otra parte importante del JavaSpeech API es el reconocimiento de voz. Sin embargo, el estado en relación al reconocimiento de voz no está siquiera, cercano al avanzado estado de la síntesis de voz. El motivo de esto es muy simple, el reconocimiento de voz es una tarea mucho más compleja.

La tarea de reconocimiento de voz, a grandes rasgos, consiste en que el ordenador captura la voz a través del micrófono convirtiéndola después en información digital. El motor de reconocimiento debe reconocer las sílabas de entre el conjunto de fonemas que ha recibido. Estas sílabas deben ser entonces combinadas para formar las palabras que inicialmente se había pronunciado.

Estas operaciones a priori bastante complejas en sí mismas, todavía aumentan su complejidad cuando consideramos características más concretas del habla humana, como las diferentes voces, la pronunciación, las diferencias dialectales dentro de una misma lengua: Diferentes personas tienen diferentes tonos de voz, hablan a diferentes velocidades, utilizan diferentes entonaciones, ...

De hecho, podríamos diferenciar de una manera genérica, dos tipos de dificultades técnicas en relación al reconocimiento de voz:

- Problemas relacionados con las diferentes características de la voz de cada persona: Como ya hemos dicho, diferentes personas tienen diferentes tonos de voz, diferente pronunciación, existen diferencias dialectales dentro de una misma lengua, ...
- Problemas relacionados con la complejidad de las estructuras del lenguaje: El reconocimiento de voz supone la identificación de palabras. Pero el lenguaje es mucho más complejo que la simple utilización de palabras una después de otra; el lenguaje tiene estructuras cuya identificación en ocasiones, es incluso esencial para entender el sentido completo de la frase. Así pues, los motores de reconocimiento actuales deben incorporar métodos para reconocer gramáticas y, por supuesto, el JavaSpeech API no sólo las soporta, sino que son un elemento imprescindible.

6.1.**JSGF (JavaSpeech Grammar Format):**Un sistema de declaración de gramáticas simplifica la tarea del motor de reconocimiento limitando el número de posibles palabras y frases que se deben considerar cuando se intenta determinar qué ha dicho el usuario. Las gramáticas también facilitan la tarea del programador de aplicaciones de voz, reduciendo y agrupando los posibles casos que se deben tener en cuenta. Existen dos tipos de gramáticas:

- Gramáticas basadas en reglas: Están compuestas de tokens y reglas. Cuando un usuario habla, la entrada es comparada con las reglas y los tokens de la gramática con el fin de identificar los patrones esperados. La aplicación de reconocimiento de voz debe proporcionar una gramática basada en reglas al motor, normalmente durante la inicialización de este.
- Gramáticas de dictado: Estas gramáticas definen miles de palabras que el usuario puede decir de una manera libre. Se trataría de gramáticas integradas

en el propio motor de reconocimiento que identifican el habla sin las restricciones que imponen las gramáticas basadas en reglas. Sin embargo, este tipo de métodos es más complejo desde un punto de vista técnico, con lo que son motores más lentos y cometen muchos errores.

La especificación de JSGF soporta reconocimiento de voz mediante gramáticas basadas en reglas, no soportando las gramáticas de dictado por estar en un estado de desarrollo mucho más temprano e inmaduro. Sin embargo, JSGF soporta gramáticas dinámicas. Esto significa que las gramáticas que utiliza una aplicación que hace uso de JSAPI pueden ser modificadas en tiempo de ejecución, según las necesidades de la aplicación. Esto permite aumentar la flexibilidad del sistema, así como optimizar las aplicaciones que requieren reconocimiento de voz, adaptando las capacidades de reconocimiento gramatical a cada contexto.

Vamos a analizar la sintaxis del *Java Speech Grammar Format* más detenidamente:

1. **Cabecera de la gramática:** Si definimos un fichero de gramáticas externo a nuestra aplicación donde declaramos las diferentes construcciones gramaticales aceptadas por el programa, este fichero debe tener un nombre identificativo único, declarado en su cabecera siguiente la sintaxis:

```
#JSGF V1.0 ISO8859-9 es;
grammar packageName.simpleGrammarName;
```

El nombre de los paquetes de gramáticas es equivalente al nombre de los paquetes en Java; por tanto un nombre completo para una gramática será una lista de identificadores de paquete separados por puntos. Utilizar estos convencionalismos puede reducir el número de conflictos entre paquetes.

Dentro de una gramática podemos importar el contenido de otra u otras gramáticas. La declaración de una importación agrega una o todas las reglas públicas de gramática a nuestra gramática local:

```
//Añade todas las reglas públicas
import <fullyQualifiedGrammar.*>;

/Añade la regla nameRule del paquete fullyQualifiedGrammar
import <fullyQualifiedGrammar.nameRule>;
```

2. **Cuerpo de la gramática:** En el cuerpo de la gramática es donde se declaran las reglas del motor de reconocimiento. Una regla gramatical debe presentar la siguiente sintaxis:

```
[public] <ruleName> = ruleExpansion ;
```

- Una regla se puede declarar con o sin el modificador `public` . Una regla que no sea declarada explícitamente como `public` será implícitamente privada. En esencia este modificador afectará a la importación de reglas en otros ficheros de gramáticas. Las reglas declaradas como `public` pueden ser importadas, las reglas privadas no podrán ser importadas, sólo podrán ser accedidas localmente.

- El nombre de una regla gramatical debe ser único en un mismo ámbito. Si una regla importada y una regla local tienen el mismo nombre, se podrían diferenciar (como en Java) utilizando el nombre totalmente cualificado de la regla.

3. **Cuerpo de una regla gramatical:** En el campo `ruleExpansion` del patrón sintáctico anterior es donde se encuentra la verdadera potencia del JSGF. En él se puede declarar la estructura gramatical que debe tener una locución para que JavaSpeech considere que "casa" con esta regla gramatical. Una expresión gramatical puede estar formada por:

   - una palabra

   - varias palabras relacionadas entre sí mediante operadores

   - una referencia a otra regla gramatical.

- **Secuencias:** Una regla puede ser definida como una secuencia de expresiones válidas. Una secuencia de expresiones válidas, separadas entre sí mediante espacios en blanco es, así mismo, una expresión válida.

  Debido a que tanto palabras como referencias a otras reglas son expresiones gramaticales válidas, las siguientes definiciones de reglas son también válidas:

```
<donde> = Vivo en Madrid;
<frase> = este <object> es <OK>;
```

  Para poder reconocer una secuencia, cada elemento debe ser pronunciado en el orden definido y obedeciendo a las condiciones impuestas en cada regla gramatical. Así, para reconocer la regla `<donde>` el usuario debe decir las palabras *"Vivo en Madrid"* en el orden exacto. El segundo ejemplo mezcla palabras con referencias a otras reglas; para reconocer la regla `<frase>` el usuario debe pronunciar *"este"* seguido de una expresión que case con la regla `<object>`, entonces *"es"* y finalmente algo que case con `<OK>`

  Una secuencia vacía no es legal. Sin embargo, existe la Secuencia vacía.

```
<d> = ; // no legal
<e> = <VOID>; // legal
```

- **Alternativas:** Una regla puede ser definida como un conjunto de expresiones alternativas separadas por una barra vertical `'|'` y opcionalmente por espacios en blanco. Por ejemplo:

```
<objeto> = lápiz | avión | monitor | <otherObjects>;
```

  Para que el motor reconozca la regla <objeto> el usuario debe decir una y sólo una de las palabras separadas mediante la barra vertical, o bien, alguna expresión que case con la regla `<otherObjects>`

  Una alternativa vacía no es legal.

```
<name> = Michael | | Mary; // not legal
```

- **Agrupaciones:**

- **Paréntesis:** Cualquier expresión legal puede ser explícitamente agrupada utilizando paréntesis '()'. Una agrupación mediante paréntesis supone considerar de manera atómica aquéllo que está contenido entre los paréntesis de apertura y cierre. Veamos un ejemplo:

```
<accion> = por favor (abre | cierra | borra);
```

Para que el motor reconozca la regla `<accion>` el usuario debe pronunciar las palabras *"por favor"* seguidas de una de entre las alternativas entre paréntesis, es decir, o bien *"abre"* o *"cierra"* o *"borra"*, pero no dos de ellas, por ejemplo.

Un ejemplo más elaborado:

```
<comando> = (abre | cierra) la (ventana |puerta) (ahora | después);
```

Esta regla admitiría frases diferentes como *"abre la ventana ahora"*, *"cierra la puerta después"*.

Un paréntesis sin contenido no está permitido

```
( ) // no legal
```

- **Corchetes:** Cualquier expresión legal puede ser explícitamente agrupada mediante corchetes `'[]'`. Una expresión situada entre corchetes puede, opcionalmente, ser pronunciada o no por el usuario. Por ejemplo:

```
<halago> = amigo mío | oh mi gran amigo | compañero del alma;
public <comando> = [ <halago> ] ven a recogerme;
```

Permite al usuario decir `"ven a recogerme"` y opcionalmente frases como `"compañero del alma ven a recogerme"` o bien `"amigo mío ven a recogerme"`

Unos corchetes sin contenido no está permitido:

```
[ ] // no legal
```

- **Asterisco *:** Una expresión legal seguida de un símbolo asterisco, significa que la expresión puede ser pronunciada cero o más veces. Veamos un ejemplo:

```
<numtelf> = cinco* dos tres ocho seis;
```

El motor admitiría como válido tanto "dos tres ocho seis" como "cinco cinco cinco dos tres ocho seis".

```
<halago> = amigo mío | oh mi gran amigo | compañero del alma;
public <comando> = <halago>* ven a recogerme;
```

En este caso, la regla `<comando>` casaría con frases diferentes como: `"ven a recogerme"`, `"amigo mío compañero del alma ven a recogerme"` o `"compañero del alma oh mi gran amigo ven a recogerme"`.

- **Operador Suma +:** Una expresión legal seguida de un símbolo asterisco significa que la expresión puede ser pronunciada una o más veces (pero al menos una vez). Veamos un ejemplo:

```
<comando> = (por favor)+ no me digas eso;
```

El motor admitiría frases como "por favor no me digas eso" o "por favor por favor no me digas eso".

- **Tags:** Los tags son un mecanismo para añadir información específica de cara a la aplicación a partes concretas de la definición de la regla. Las aplicaciones típicamente utilizan los tags para simplificar o mejorar el proceso de reconocimiento. Los tag son texto encerrado entre llaves de conjunto '{}'

  Los tags no afectan al reconocimiento de una gramática. Los tags solamente son adjuntados al objeto devuelto por el reconocedor de voz a la aplicación. De esta manera, el programador puede saber que el texto pronunciado por el usuario, casa con una determinada parte de la regla gramatical o bien con otra. Veamos un ejemplo:

```
<comando>= por favor (abre {OPEN} | cierra {CLOSE}) la ventana;
```

Mediante el uso de tags podremos saber, a nivel de código Java y en tiempo de ejecución, si el usuario ha dicho "por favor abre la ventana" o bien "por favor cierra la ventana", ambas expresiones válidas de la regla gramatical anterior pero que no serían diferenciables de otro modo.

6.2.Reconocimiento de voz en Java

Ahora que ya conocemos el JSGF podemos ver ejemplos concretos de código Java que permite el reconocimiento de voz basado en reglas. Para ello, vamos a utilizar código que forma parte del Proyecto desarrollado como ejemplo para la asignatura, el voiceBrowser. Empecemos definiendo previamente una gramática válida, utilizando algunos de los elementos vistos anteriormente:

```
#JSGF V1.0 ISO8859-9 es;
grammar voiceBrowser.linksGrammar;

public <empezar> =  empezar {empezar};
public <detener> =  detener {detener};
public <continuar> = continuar {continuar};
public <releer> = releer {releer};
public <leer_opciones> = leer texto {leer_texto} | leer
enlaces {leer_enlaces} | leer todo {leer_todo};
```

Ahora vamos a programar la parte de la aplicación Java que permitirá reconocer las reglas gramaticales definidas en voiceBrowser.linksGrammar. Para ello debemos realizar dos operaciones claramente diferenciadas en el API de JavaSpeech:

1. Declaración e inicialización del motor de reconocimiento de voz. Lo haremos en el método main de la aplicación, dado que nuestro ejemplo es muy simple (para aplicaciones más complejas se suele definir un método que realice esta tarea). En la parte de inicialización del motor de reconocimiento también:
   - Cargaremos de forma estática la gramática que vamos a utilizar.
   - Agregaremos un ResultListener, que deberá ser un objeto que extienda la clase ResultAdapter.

```
public class RecognizerMultimedia extends ResultAdapter {
       private static Recognizer recognizer;

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

                     recognizer.allocate();

                     FileReader grammar = new FileReader("grammar.txt");
                     RuleGrammar ruleGrammar = recognizer.loadJSGF(grammar);
                     ruleGrammar.setEnabled(true);

                     recognizer.addResultListener(new RecognizerMultimedia());

                     System.out.println("[Motor de Reconocimiento] Listo");
                     recognizer.commitChanges();
                     recognizer.requestFocus();
                     recognizer.resume();

              } catch (EngineException engineException) {
                     engineException.printStackTrace();
              } catch (EngineStateError stateError) {
                     stateError.printStackTrace();
              } catch (AudioException audioException) {
                     audioException.printStackTrace();
              } catch (GrammarException grammarException) {
                     grammarException.printStackTrace();
              } catch (FileNotFoundException fileNotFoundException) {
                     fileNotFoundException.printStackTrace();
              } catch (IOException ioException) {
                     ioException.printStackTrace();
              }
       }
}
```

1

2. La segunda tarea será la definición de un método que sea llamado cada vez que el motor reconozca una regla de nuestra gramática. Para ello, nuestra clase debe extender la clase `ResultAdapter`, de manera que cada vez que el motor reconozca una regla gramatical, llamará al método `public void` **`resultAccepted`**`(ResultEvent e)`, que forma parte de la clase `ResultAdapter` y en cuya reimplementación deberemos interpretar cuál ha sido la regla gramatical que ha pronunciado el usuario.

La creación del objeto `Recognizer` es similar a la creación de un objeto `Synthesizer`. Utilizamos la clase `Central` para obtener un objeto del tipo esperado, pasándole al método `createRecognizer` un objeto de tipo `Local` que especifica la lengua del objeto de reconocimiento de voz:

```
recognizer = Central.createRecognizer(new EngineModeDesc(new Locale("es")));
```

Una vez hemos hecho esto, cargamos la gramática que vamos a utilizar. La gramática debe estar definida en el fichero que pasamos como parámetro al método `loadJSGF(Reader r)` del objeto `Recognizer`. Esta gramática debe ser activada después

```
FileReader grammar = new FileReader("grammar.txt");
RuleGrammar ruleGrammar = recognizer.loadJSGF(grammar);
ruleGrammar.setEnabled(true);
```

Después, completamos la inicialización del objeto de reconocimiento de voz activando los cambios, proporcionando el foco al objeto `Recognizer` y poniéndolo en estado `RESUMED`:

```
recognizer.commitChanges();
recognizer.requestFocus();
recognizer.resume();
```

Además, para poder tratar los eventos de reconocimiento de voz del objeto `Recognizer`, le debemos añadir un `ResultListener`, y en la implementación del método `resultAccepted(ResultEvent re)` interpretaremos los textos reconocidos por el motor.

```
recognizer.addResultListener(new RecognizerMultimedia());
```

2

```
public class RecognizerMultimedia extends ResultAdapter {
	private static Recognizer recognizer;

	public void resultAccepted(ResultEvent re) {
		try {
			StringBuffer strBufferTokens = new StringBuffer();
			StringBuffer strBufferTags = new StringBuffer();

			Result result = (Result)(re.getSource());
			ResultToken tokens[] = result.getBestTokens();

			if (tokens != null) {
				for (int i = 0; i <= tokens.length; i++)
					strBufferTokens.append(
						tokens[i].getSpokenText() + " ");
			}
			System.out.println("Tokens: " + strBufferTokens.toString());


			String tags[] = ((FinalRuleResult)(re.getSource())).getTags();

			if (tags != null) {
				for (int i = 0; i <= tags.length; i++)
					strBufferTags.append(tags[i] + " ");
			}
			System.out.println("Tags: " + strBufferTokens.toString());

		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Cuando un patrón de habla es reconocido como parte de la gramática, se lanza un evento de tipo `result` y el método `public void resultAccepted(ResultEvent re)` del objeto que se ha añadido como oyente de este tipo de eventos es llamado. El objeto `ResultEvent` contiene toda la información que se necesita para saber qué frase de la gramática ha sido pronunciada por el usuario. Podemos obtener, tanto el nombre de las reglas gramaticales, como los `Tags` (si los hubiera) de esas reglas.

Nombre de las reglas:

```
Result result = (Result)(re.getSource());
ResultToken tokens[] = result.getBestTokens();
```

Tags:

```
String tags[] = ((FinalRuleResult)(re.getSource())).getTags();
```

7.Reproducción de audio: AU, WAV y MP3

Java es un lenguaje tremendamente completo, cuya mayor virtud de cara al programador es la facilidad con que se pueden añadir nuevas funcionalidades al API básica que trae el JDK, con tan solo añadir al CLASSPATH el fichero jar que incorpora la implementación del API en cuestión.

Con la creación de Java, Sun Microsystems creó también un formato de audio propio que utilizarían las aplicaciones programadas en su lenguaje. El origen de Java estuvo totalmente influenciado por su cometido principal: servir como lenguaje para Internet. También bajo esta premisa se desarrolló su formato de audio propio, el formato AU; es un formato, pues, de baja calidad pero muy alta tasa de compresión, idóneo para Internet, pero muchas veces inadecuado para otro tipo de aplicaciones.

Dada esta limitación del formato AU, cuando más tarde se descubrió la potencia de Java, no sólo en aplicaciones orientadas a Web, sino también en aplicaciones de escritorio, se desarrolló un nuevo proyecto que ampliara las características Multimedia del lenguaje. El resultado es el JMF (Java Media Frame), un paquete que provee características multimedia avanzadas tanto en audio como en vídeo, con soporte para reproducción de audio y vídeo en diferentes formatos, soporte para el control de dispositivos de adquisición de datos multimedia como cámaras de vídeo o tarjetas de sonido, soporte para formatos multimedia en tiempo real como RTP y RTCP, etc. Un paquete realmente completo, de hecho es uno de los más potentes de toda la familia de paquetes del J2SE.

En un trabajo como este, donde se profundiza en una de las tecnologías multimedia de Java relacionadas con el audio que NO está incluida en el JMF, he querido acercarme de una manera muy somera a las tecnologías de Audio incluidas en el paquete JMF. Ni siquiera vamos a explicar la filosofía de funcionamiento del Java Media Frame, bastante compleja en sí misma y que nos llevaría varias páginas de explicación, tan sólo vamos a ver con qué facilidad podemos reproducir audio en formato AU, WAV o MP3 en una aplicación Java. Para un conocimiento más extenso de las tecnologías del Java Media Frame, me remito a su propia documentación en la página web del paquete http://java.sun.com/products/java-media/jmf/2.1.1/download.html

```
//Para URL
import java.net.*;

//Import de la JMF
import javax.media.*;

public class Audio {

	public static void main(String args[]) {
		playerJMF("file:///C:/sound.wav");
		playerJMF("file:///C:/sound.au");
		playerJMF("file:///C:/sound.mp3");

	}

	private static void playerJMF(String pathJMF) {
		Player player = null;
		try {
			System.out.println("Comienza reproducción");

			URL url = new URL(pathJMF);
			player = Manager.createPlayer(url);
			player.start();

			System.out.println("Reproducción finalizada");

		} catch (MalformedURLException malformedURL) {
			malformedURL.printStackTrace();
		} catch (IOException ioException) {
			ioException.printStackTrace();
		} catch (NoPlayerException noPlayerException) {
			noPlayerException.printStackTrace();
		}
	}
}
```

8.voiceBrowser: La idea de un Navegador Web para invidentes

La Multimedia utilizada como una manera de acercar la tecnología a aquellas personas que sufren alguna deficiencia sensorial es un tema apasionante. Acercar el acceso a Internet a personas con discapacidades visuales, no sólo es algo posible, sino totalmente necesario; como dijimos al principio de este documento, uno de los objetivos más importantes de la tecnología es mejorar la calidad de vida de las personas y este objetivo toma su sentido por completo cuando hablamos de discapacitados.

voiceBrowser no es más que una buena idea. En el tiempo dedicado a la realización de este proyecto, no ha sido posible diseñar un navegador con las capacidades suficientes de potencia y usabilidad para considerarse una aplicación *"de producción". S*in embargo, las ideas sobre las que se basa y su propia estructura pueden servir como base para seguir trabajando en este interesantísimo proyecto. Ni qué decir tiene que tanto el código de la aplicación como este mismo documento están a disposición de todo el que quiera utilizarlos, modificarlos o ampliarlos, bajo las licencias GPL [página web licencia GPL] para el código fuente de la aplicación y Creative Commons [página web de Creative Commons] para el presente documento.

voiceBrowser combina varias tecnologías proveidas por Java para cumplir su

cometido:

1. **Reconocimiento y síntesis de voz:** Proveído por la JavaSpeech API y cuyas definiciones se encuentran en los paquetes del J2SDK

```
javax.speech;
javax.speech.recognition;
javax.speech.synthesis;
```

Las funciones de voz ya han sido sobradamente explicadas a lo largo de los puntos anteriores de este documento, por lo que no se hará hincapié aquí.

2. **Procesado HTML:** El propio J2SDK de Sun provee de manera nativa, herramientas tanto para realizar conexiones a servidores Web y obtener las páginas deseadas (paquete `java.net`) como para procesar estas páginas Web (paquetes `javax.swing.text.html` y `javax.swing.text.html.parser`) Estas herramientas de *parsing* que trae por defecto el API de Java 2, permiten trabajar con documentos HTML, identificar los tags que los componen, los atributos de los diferentes tags y otras tareas similares, aunque no proporcionan demasiada potencia

   Existe una implementación de una librería de parsing HTML mucho más potente que las clases que incorpora el J2SDK. Esta librería se llama Xerces e incorpora muchas clases de parsing de elementos web concretos.

3. **Presentación de la página HTML en pantalla:** De esto se encarga un control Swing de Java que permite mostrar documentos de texto, de texto enriquecido (RTF) y documentos HTML (con ciertas limitaciones). Se trata de la clase `JEditorPane`.

8.1.Procesado HTML en Java. Uso del paquete `javax.swing.text.html`

El paquete `javax.swing.text.html` provee las clases e interfaces necesarias para trabajar con documentos HTML de una manera no demasiado potente, pero suficiente para muchos cometidos. La aplicación voiceBrowser utiliza estas funciones de parsing HTML para *"conocer"* la estructura de los documentos HTML que accede el usuario para poder proporcionar una versión para invidentes.

voiceBrowser funciona de la siguiente manera:

1. Obtiene la página web solicitada del servidor Web en Internet. Para ello, utiliza las clases de manejo de Streams de Java, que permiten acceder a una dirección web de una forma tan sencilla como esta:

```
URL direction = new URL("http://www.google.com");
InputStream is = direction.openStream();
InputStreamReader parseIn = new InputStreamReader(is);
```

2. Crear una clase que extienda a la clase `HTMLEditorKit.ParserCallback`

del paquete `javax.swing.text.html`. Esta clase permite definir métodos que son llamados cada vez que en un documento HTML que está siendo parseado se encuentra un tag. Provee métodos para tratar el tag de apertura, el tag de cierre y el texto contenido entre los tags de apertura y cierre:

```
public void handleStartTag(HTML.Tag tag, MutableAttributeSet
                                attributes, int position);

public void handleText(char[] text, int position);

public void handleEndTag(HTML.Tag tag, int position);
```

El orden de llamada de los métodos de HTMLEditorKit.ParserCallback es el siguiente:

```
<a href="http://www.google.com"> Google </a>
```

****	Java llamará al método handleStartTag con tag = HTML.Tag.A attributes = "href="http://www.google.com" position = offset en la página web
Google	Java llamará al método handleText con text = "Google" position = offset en la página web
****	Java llamará al método handleEndTag con tag = HTML.Tag.A position = offset en la página web

3. Por último, se debe pedir a Java que realice el parseado de la página web. Asumiendo que:

   - Hemos cargado la página web original en un objeto de tipo `Reader` llamado `parseIn`.
   - Hemos creado una clase que extiende a `HTMLEditorKit.ParserCallback`, que implementa la lógica de procesado del documento HTML que llamaremos `processHTML`.
   - El documento después de ser procesado queremos almacenarlo en un objeto de tipo `Writer` llamado `parseOut`.

   Podemos ejecutar el siguiente código para realizar el parsing:

```
ParserGetter kit = new ParserGetter();
HTMLEditorKit.Parser parser = kit.getParser();
```

```
BufferedWriter parseOut =
            new BufferedWriter(new StringWriter());

HTMLEditorKit.ParserCallback callback =
            new processHTML(parseOut, direction.toString());

parser.parse(parseIn, callback, true);

parseOut.flush();
parseIn.close();
parseOut.close();
```

4. Una vez realizado esto, el documento HTML original habrá sido procesada según la implementación de la clase `processHTML` y el resultado estará almacenado en el objeto de tipo `Writer parseOut`.

9.Bibliografía

- Potts, Stephen - Pestikov, Alex - Kopack, Mike. “Chapter 12: Procesing Speech with Java”. En: *Java 2 Unleashed*, Sixth Edition (ISBN:0-672-32394-X). Ed. Pearson Education. Versión electrónica: http://www.developer.com/java/article.php/1471001

- *Java Speech API Programmer´s Guide*, Versión 1.0 [en línea]. 26 de Octubre de 1998. Disponible en Web: http://java.sun.com/products/java-media/speech/forDevelopers/jsapi-guide/index.html

- *Java Speech API Specification Document*, Versión 1.0 [en línea]. Disponible en Web: http://java.sun.com/products/java-media/speech/forDevelopers/jsapi-doc/index.html

- *Java Speech Grammar Format Specification*, Versión 1.0 [en línea]. Disponible en Web: http://java.sun.com/products/java-media/speech/forDevelopers/JSGV/index.html

- *Java Speech Markup Language Specification*, Versión 0.5 [en línea]. Disponible en Web: http://java.sun.com/products/java-media/speech/forDevelopers/JSML/index.html

- Java Speech API, Jarkko Enden [en línea]. Disponible en Web formato PDF: http://www.cs.helsinki.fi/u/campa/teaching/jarkko-final.pdf