Java Speech Application Program Interface and Java Speech Recognition Package

3.1 Introduction
3.2 Design Goals for the Java Speech API
3.3 Speech Engine (Java Speech Package)
3.4 Speech Recognizer as a Speech engine
3.5 Speech Recognizer State System
3.6 Speech Recognition States
3.7 Speech Recognition Grammars
3.8 Speech Recognizer Results
3.9 Speech Recognizer Properties
3.10 Conclusion
CHAPTER – 3

JAVA SPEECH APPLICATION PROGRAM INTERFACE AND JAVA SPEECH RECOGNITION PACKAGE

3.1 INTRODUCTION

The description of a software package in a research thesis is remarkable. The description here has been prepared to provide a better understanding of what has been done in this thesis and of the process involved in the recognition task by utilizing the Java Speech API and speech engine. Speech technology was once restricted to the area of science fiction, but now it is easily accessible for use in real life applications. We have use the Java Speech API which defines a standard, easy and cross platform software interface to state of the art speech technology. Two basic speech technologies supported by the Java Speech API are speech recognition and speech synthesis. We have used Java Speech API for recognition of Gujarati and Hindi speech.

The Java Speech Application Program Interface was developed by the Sun Microsystems in cooperation with speech technology companies [38-46, 52, 53, 55, 56]. It defines a software interface to allow developers to take benefits of speech technology for individual and enterprise computing. By the power of the inbuilt strengths of the Java platform, the Java Speech Application Program Interface enables developers of speech applications to incorporate more complicated and ordinary user interfaces into Java applications and applets that can be installed on a broad range of platforms.

Speech technology is becoming progressively more significant in both individual and enterprise computing as it is used to improve existing
user interfaces and to provide new way of human communication with computers. Speech technology provides hands free use of computers and access to many computing facilities not available from the desktop computer and over the telephone. It improves convenience for the disabled users.

Large companies can benefit from a wide range of applications of speech recognition using the Java Speech API. For example, Interactive Voice Response (IVR) systems are an attractive option to touch tone interfaces over the telephone. Speech dictation systems can be significantly faster than typing for users. Speech recognition improves accessibility to computers for many people with physical limitations.

Speech enabled interfaces offer Java application developers the chance to implement distinct and engaging personalities for their applications and to differentiate their products. Java application developers can have access to state of the art speech technology from top speech companies. Using standard API for speech, users can decide the speech products which best meet their requirements and their budget.

The existing capabilities of the Java platform make it attractive for the development of a wide range of applications. Using the Java Speech API, Java developers can expand existing user interfaces with speech input and speech output. The Java platform offers an attractive option with:

- **Portability:** Java, Java Speech API and Java virtual machine are available for a wide variety of hardware and operating systems and web browsers.
• **Powerful and compact:** Java platform provides a powerful, object oriented and garbage collected language which allows rapid development and improved reliability.

• **Network support and security:** The Java platform supports network and robust security.

The Java Speech API is part of the Java Media and Communication APIs which is a set of software interfaces that provides access to multimedia content such as audio, video, 2D and 3D graphics, animation, telephony, advanced imaging etc.

The Internationalization features of the Java programming language and the use of the Unicode character set make development of multilingual applications simpler. Various classes and interfaces of Java Speech API use the patterns of JavaBeans. Java Speech API events and the event mechanisms of AWT are integrated with JavaBeans and the Java Foundation Classes (JFC).

Sun Microsystems doesn’t provide a reference implementation for Java Speech API but it provides a list of third party vendors who have products that provide a Java Speech API interface. The following companies are providers of Java compatible speech products:

• **FreeTTS** is an interface for speech synthesis facility and is written entirely in Java [32]

• **IBM’s Speech for Java** is based on the IBM ViaVoice product [41][55]

• **The Cloud Garden** will work with any speech engine that is based on Microsoft Speech API (SAPI) version 5 [43]

• **Lernout and Hauspie’s TTS for JSAPI** runs on Sun and provides a number of sophisticated features [56]

• **Conversa Web 3.0** is a speech-enabled Web browser [38]
• **Festival** is UNIX based and supports a number of programming interfaces in addition to Java [44]

### 3.1.1 Java Speech API Implementation

The Java Speech API can enable access to the most important and useful state of the art speech technologies. Sun is actively working with speech technology companies to encourage the implementations of the API that can extend Java virtual machines.

The following are the primary implementation mechanisms:

i. **Native Implementations:** most existing speech technology is implemented in C and C++ and accessed through platform-specific APIs such as the Apple Speech Managers [59] and Microsoft’s Speech API (SAPI) [36], or other APIs such as the Speech Recognition API (SRAPI) and vendor-specific APIs. Using the Java Native Interface (JNI) and Java software wrappers, speech vendors can implement the Java Speech API on top of their existing speech software. This is expected to be the means of support for JSAPI in the first round of deployment.

ii. **Java software Implementations:** Speech synthesizers and speech recognizers can be written in Java software. It benefits from the portability of the Java platform and from the continuing improvements in the execution speed of Java virtual machines we have used Cloud Garden implementation of Java Speech API [43].

iii. **Telephony Implementations:** Telephony applications are typically implemented with dedicated hardware to support a large number of simultaneous connections. Speech recognition and speech synthesis capabilities on this dedicated hardware can be wrapped with Java software to support the Java Speech API as a special kind of native implementation.
3.1.2 Java Speech API Requirements

A user must have certain minimum software and hardware available in order to use the Java Speech API. The following is a general requirements. The individual requirements of speech application can vary greatly.

i. **Speech software:** A Java Speech API compliant speech recognizer or synthesizer is required.

ii. **System requirements:** Most speech recognizers and some speech synthesizers require powerful desktop computers to run effectively.

iii. **Audio Hardware:** Speech synthesizers require audio output while speech recognizers require audio input. The dictation system performs better with good quality sound cards.

iv. **Microphone:** Speech recognition systems get audio input through a microphone. Speech recognizers with dictation systems are sensitive to the microphone. Headset microphones generally provide best performance in noisy environments. Desktop microphones can be used in some environments and are typically designed for best performance speaking from about 40 to 60 centimeters from the microphone.

3.2 Design Goals for the Java Speech API

The Java Speech API allows developers to include advanced user interfaces into Java applications.

Design goals for the Java Speech API are as the following:

i. It supports for speech synthesizers and for both command-and control and dictation speech recognizers.
ii. Write once, Run anywhere access to speech synthesis and speech recognition for consistent access to speech technology from different companies and across multiple platforms.

iii. Access to existing speech technology.

iv. Integration with other features of the Java platform, including other Java Media and Communication APIs.

v. Simple and compact.

### 3.3 SPEECH ENGINE (JAVA SPEECH PACKAGE)

Speech engine is a general term for a system designed to handle either speech input or speech output. The Java speech package of the Java Speech API defines conceptual software representation of a speech engine. The speech synthesizer (javax.speech.synthesizer) and speech recognizer (javax.speech.recognition) are instance of speech engine. Speaker verification systems and speaker identification systems are also speech engines but are not currently supported through the Java Speech API.

The java speech package (javax.speech) defines classes and interfaces that define the basic functions of an engine. The Java speech synthesis and Java speech recognition package extend java speech package and enhance the basic functionality to define the specific capabilities of speech synthesizers and speech recognizers.

The Java Speech API makes only one assumption about the implementation of a speech engine that it provides a true implementation of the Java classes and interfaces defined by the API. A speech engine may be completely software based or it may consist of both the software and the hardware. The engine may be local or operating on a server. The engine may be written either entirely in Java or may be a combination of java software and native code. The
basic processes for using a speech engine in an application are as follows:

i. Identify the application’s functional requirements for an engine (e.g., language or dictation capability).

ii. Locate and create an engine that meets those functional requirements.

iii. Allocate the resources for the engine.

iv. Set up the engine.

v. Begin operation of the engine technically it means resume engine.

vi. Use the engine.

vii. De-allocate the resources of the engine.

3.3.1 Basic Classes and Interfaces of Speech Package

The table 3.1 lists the basic classes and interfaces defined by the java speech package.

<table>
<thead>
<tr>
<th>Name of class</th>
<th>Description</th>
<th>Sub class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central</td>
<td>Object through which speech engines are located and selected.</td>
<td></td>
</tr>
<tr>
<td>Engine</td>
<td>Basic interface implemented by all speech engines and extended by recognizers and synthesizers.</td>
<td>Synthesizer, Recognizer</td>
</tr>
<tr>
<td>EngineModeDesc</td>
<td>Description of a mode of operation of a speech engine.</td>
<td>SynthesizerModeDesc, RecognizerModeDesc</td>
</tr>
<tr>
<td>EngineList</td>
<td>Container for EngineModeDesc objects.</td>
<td></td>
</tr>
<tr>
<td>SpeechEvent</td>
<td>Basic speech event object from which all speech events are derived.</td>
<td>EngineEvent, AudioEvent, SpeakableEvent, GrammarEvent, Result Event, RecognizerEvent,</td>
</tr>
<tr>
<td>Name of class</td>
<td>Description</td>
<td>Sub class</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>EngineAttributes</td>
<td>Basic run-time adjustable properties of a speech engine.</td>
<td>SynthesizerAttributes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RecognizerAttributes</td>
</tr>
<tr>
<td>EngineEvent</td>
<td>Basic event provided by all speech engines.</td>
<td>RecognizerEvent</td>
</tr>
<tr>
<td>EngineListener</td>
<td>Interface implemented by applications to receive engine events.</td>
<td>SynthesizerListener</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RecognizerListener</td>
</tr>
<tr>
<td>EngineAdapter</td>
<td>Implementation of the EngineListener interface that can be easily extended by</td>
<td>SynthesizerAdapter</td>
</tr>
<tr>
<td></td>
<td>applications.</td>
<td>RecognizerAdapter</td>
</tr>
<tr>
<td>AudioManager</td>
<td>Each speech engine provides an audio manager through which applications</td>
<td></td>
</tr>
<tr>
<td></td>
<td>control the audio input or output of the engine.</td>
<td></td>
</tr>
<tr>
<td>AudioEvent</td>
<td>Event associated with audio input/output.</td>
<td>AudioLevelEvent</td>
</tr>
<tr>
<td>AudioListener</td>
<td>Listener that receives audio events.</td>
<td>RecognizerAudioListener</td>
</tr>
<tr>
<td>AudioAdapter</td>
<td>Convenience implementation of AudioListener.</td>
<td>RecognizerAudioAdapter</td>
</tr>
<tr>
<td>VocabManager</td>
<td>Each speech engine implements a vocabulary manager to allow applications to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>provide engines with word lists and pronunciations.</td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>A basic description of a word for use in vocabulary management.</td>
<td></td>
</tr>
<tr>
<td>EngineCentral</td>
<td>Not used by applications: used by speech engines to identify themselves to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Central.</td>
<td></td>
</tr>
</tbody>
</table>

*Table-3.1: Classes and Interfaces of the Java Speech Package*
3.3.2 Properties of a Speech Engine

Applications are responsible for determining their functional requirements for a speech synthesizer and/or speech recognizer. In our project we use rule based grammar and English language as language for recognition. Applications are also responsible for determining behavior when there is no speech engine available with the required features. After determining the functional requirements, speech engine can be started. Functional requirements are handled in applications as engine selection properties. Every speech synthesizer and speech recognizer is defined by a set of properties. An engine may have one or many modes of operation each of which is defined by a unique set of properties and represented by a mode descriptor object.

The basic engine properties are defined in the EngineModeDesc class while other specific properties for speech recognizers and synthesizers are defined by the RecognizerModeDesc and SynthesizerModeDesc classes that are enclosed in the Java speech recognition and Java speech synthesis packages respectively.

Default mode descriptor object is provided by speech engines to describe their capabilities but an application can create its own mode descriptor objects to specify its functional requirements. An engine provided mode descriptor describes an actual mode of operation whereas an application defined mode descriptor defines a preferred mode of operation. The basic properties defined for all speech engines are listed in Table 3.2.

<table>
<thead>
<tr>
<th>Name of Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EngineName</td>
<td>A String that defines the name of the speech engine. For example My Dictation System.</td>
</tr>
<tr>
<td>ModeName</td>
<td>A String that defines a specific mode of operation of the</td>
</tr>
</tbody>
</table>
Table-3.2: Basic engine selection properties: EngineModeDesc

The one additional property defined by the SynthesizerModeDesc class for speech synthesizers is shown in Table 3.3.

Table-3.3: Synthesizer selection properties: SynthesizerModeDesc

The two additional properties defined by the RecognizerModeDesc class for speech recognizers are shown in Table 3.4.

Table-3.4: Recognizer Selection Properties: RecognizerModeDesc
All three mode descriptor classes use the get and set property patterns for JavaBeans. For example, the Locale property has get and set methods of the form:

```java
Locale getLocale();
void setLocale(Locale l);
```

Moreover, all the properties are defined by class objects, never by primitives. A null value always represents don’t care condition and is used by applications to indicate that a particular property is unimportant to its functionality. For example, a null value for the dictation supported property indicates that dictation is not relevant to engine selection. A value of TRUE indicates that dictation is required and FALSE indicates explicitly that dictation should not be provided.

### 3.3.3 Finding and Selecting Engines

Applications are responsible for determining the properties they require of a synthesizer or recognizer. For example application must determine the language of operation, whether dictation is required in the application, what types of voices might be required in synthesis etc.

Applications are also responsible for determining behavior when there is no speech engine available that has the required properties. For an application that requires speech to operate, it could be necessary to indicate an error to the user and to exit.

### 3.3.4 Speech Engine Creation

The easiest way to create an engine is to use the methods of the Central class:

```java
Synthesizer Central.createSynthesizer(EngineModeDesc props);
Recognizer Central.createRecognizer(EngineModeDesc props);
```
The above methods return a synthesizer or recognizer that has the properties required by the application or null if no matching engine is available. The methods are suitable for most applications. However, if more than one engine is available that matches the required properties then the application has no control for selection of available engine.

If the application needs more control, then it should use the availableSynthesizers or availableRecognizers methods and iterate the EngineList until it has a single EngineModeDesc object. Both the SynthesizerModeDesc and RecognizerModeDesc classes provide createEngine method which returns a Synthesizer or Recognizer respectively.

### 3.3.5 Engine Allocation

Engine allocation is the process in which the resources required by the recognizer or synthesizer are obtained. Engines allocation requires considerable resources (CPU, memory and disk space) and since they made need exclusive access to an audio resource (e.g. microphone input or speaker output). Allocation can be a slow procedure for some.

The allocate method of the Engine interface requests the engine to perform allocation. Allocation is an asynchronous call: the call may return before the allocation is complete. To decide whether allocation is completed or not, an application must attach an Engine Listener and wait for an ENGINE_ALLOCATED event.

### 3.3.6 Engine De-Allocation

Applications should de-allocate an engine once it is no longer required since a speech engine may use substantial resources to operate. De-allocation frees system resources for other uses. Like allocate, de-
allocate is an asynchronous method and the application can receive an ENGINE_DEALLOCATED event through the Engine Listener once de-allocation is completed.

A de-allocated engine can be allocated again. The engine may preserve limited information to reduce the re-allocation time. Figure 3.1 shows the state diagram for allocation and de-allocation of an engine.

![Figure-3.1: Engine Allocation State Diagrams](image)

### 3.3.7 Engine States: Pause and Resume

A speech engine provides two primary states (paused and resumed) of operation once it has been allocated. Pausing and resuming affect the audio input or output of an engine. If an engine is allocated in the resumed state then applications can immediately start use of the engine. Any engine may be paused or resumed for an indefinite period using the pause and resume methods of the Engine interface.

The pause and resume methods of the Engine interface are asynchronous hence, they may return before the call has fully taken effect. ENGINE_PAUSED and ENGINE_RESUMED events are sent to Engine Listeners attached to the synthesizer or recognizer when the pause and resume take effect.

The isPaused() method of the Engine interface is available to test the state of the engine. If there is a pending pause between a call to pause
and the ENGINE_PAUSED event then isPaused() returns false. This test method indicates the current state and not pending changes.

Figure 3.2 shows the basic pause and resume diagram for a speech engine. Because an engine is allocated into the resumed state, the ENGINE_ALLOCATED event enters the engine resumed state. The ENGINE_DEALLOCATED event may take the engine from either the paused or resumed states to the de-allocated state.

![Figure-3.2: Paused and Resume Engine State](image)

For a recognizer, pausing and resuming turns audio input off and on and is similar to the switching the microphone off and on. Generally if we pause a recognizer then it will stop the recognizer's internal processes that match audio with grammars. If the user was in the middle of speaking at the instant at which the recognizer was paused then the recognizer is forced to finalize its recognition process because a recognizer can't assume that the audio received just before pausing is in any way linked to the audio data that it will receive after being resumed. In technical term pausing introduces a discontinuity into the audio input stream.

One difficulty for pausing and resuming a recognizer is the role of internal buffering. A recognizer has a buffer for audio input which mediates between the audio device and the internal component of the recognizer which perform that match of the audio with the grammars. If recognizer is performing faster than the buffer is empty or nearly
empty but if the recognizer is temporarily suspended or operates slower than real time then the buffer may contain seconds of audio or more.

When a recognizer is paused, the pause takes effect on the input end of the buffer. It means that the recognizer stops putting data into the buffer. On the other end of the buffer where the actual recognition is performed and the recognizer continues to process audio data remaining in buffer. The recognizer can produce recognition results for a limited period of time even after it has been paused.

### 3.3.8 Speech Events

Speech engine has generally many types of events. Applications need not require handling of all events however, some events are mainly important for implementing speech applications. For example some result events must be handled to obtain recognized text from a speech recognizer. Events are issued to a listener attached to an object involved in generating that event. All the speech events are extended from the SpeechEvent class in the java speech package. The events of the java speech package and recognition package are listed in Table 3.5 and Table 3.6 respectively.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SynthesizerEvent</td>
<td>Defines the specialized events of a Synthesizer.</td>
</tr>
<tr>
<td>SpeakableEvent</td>
<td>Indicates the progress in output of synthesized text.</td>
</tr>
</tbody>
</table>

*Table-3.5: Events of the Java Speech Package*

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecognizerEvent</td>
<td>Extends the EngineEvent for the specialized events of a Recognizer.</td>
</tr>
</tbody>
</table>
### Table-3.6: Events of Java Speech Recognition Package

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrammarEvent</td>
<td>Indicates an update of or a status change of a recognition grammar.</td>
</tr>
<tr>
<td>Result Event</td>
<td>Indicates status and data changes of recognition results.</td>
</tr>
<tr>
<td>RecognizerAudioEvent</td>
<td>Extends AudioEvent with events for start and stop of speech and audio level updates.</td>
</tr>
</tbody>
</table>

### 3.3.9 Other Engine Functions

The other functions of speech engines are:

- Runtime Engine Properties
- Audio management
- Vocabulary management

### 3.4. SPEECH RECOGNIZER AS A SPEECH ENGINE

The basic functions provided by a speech recognizer consist of grammar management and the generation of results when a user says things that match active grammars. The Recognizer interface is derived from the Engine interface to provide these functions.

The following is a list of the functions that the java speech recognition package derived from the java speech package and summarizes some of the ways in which that function is specialized.

- The properties of a speech engine defined by the EngineModeDesc class is also applies to recognizers. The RecognizerModeDesc class added information regarding dictation capabilities of a recognizer and regarding users who have trained the engine.
Recognizers can be searched, selected and created through the Central class in the java speech package.

Recognizers derived the basic state systems of an engine from the Engine interface. The four allocation states, the pause and resume state. It’s monitoring methods and its update events.

Recognizers generate all the standard engine events. The java speech recognition package also extends the EngineListener interface as RecognizerListener to provide events that are specific to recognizers.

Other engine functionality derived as an engine comprises of the runtime properties.

**3.5 SPEECH RECOGNIZER STATE SYSTEM**

As mentioned above, a Recognizer inherits the basic state systems defined in the java speech package through the Engine interface.

- The basic engine state system specifies the current allocation state of the engine. The four allocation states are ALLOCATED, DEALLOCATED, ALLOCATING_RESOURCES and DEALLOCATING_RESOURCES.
- The PAUSED and RESUMED states are sub states of the ALLOCATED state. The paused and resumed state of a recognizer shows that whether audio input is on or off. Pause a recognizer is similar to turning off the input microphone: input audio is lost.
- The getEngineState() method of the Engine returns a long value representing the current engine state. For example an ALLOCATED recognizer in the RESUMED state will have both the ALLOCATED and RESUMED bits set.
- The testEngineState() and waitEngineState() methods are simple methods for monitoring engine state. The test() method tests for existence in a particular state. The wait() method blocks until a particular state is reached.
- An EngineEvent is generated each time an engine changes its state. The event class includes the new and old engine states.

The recognizer adds two sub state systems to the ALLOCATED state. The two new sub state systems specifies the current activities of the recognizer’s processing the LISTENING, PROCESSING and SUSPENDED and the FOCUS_ON and FOCUS_OFF states. These new sub-state systems are analogous states to the PAUSED and RESUMED states and operate independently as shown in Figure 3.3.

![Figure-3.3: Recognition States](image)

### 3.6 SPEECH RECOGNITION STATES

The state system of a recognizer represents the current recognition activity of the recognizer. An ALLOCATED Recognizer is always in any of the following three states:
- **LISTENING state:** The Recognizer is listening to incoming audio for speech that may match an active grammar and it has not detected speech yet. A recognizer remains in this state until listening to silence and when audio input runs out because the engine is paused.

- **PROCESSING state:** The Recognizer is processing incoming speech that may match an active grammar. In this state the recognizer is generating a result.

- **SUSPENDED state:** The Recognizer is temporarily suspended while grammars are updated. In this state audio input is buffered for processing before the recognizer returns to the LISTENING and PROCESSING states.

### 3.7 SPEECH RECOGNITION GRAMMAR

A grammar defines what a recognizer should listen for in incoming speech and set of tokens a user can say. A token is typically a single word. Grammar also defines the patterns in which those words are spoken. The Java Speech API supports two types of grammars: rule grammars and dictation grammars. We have used rule based grammar for Gujarati and Hindi speech recognition. Table 3.7 lists some important differences between rule grammar and dictation grammar are:

<table>
<thead>
<tr>
<th>Rule Grammar</th>
<th>Dictation Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is defined by an application</td>
<td>It is defined by a recognizer.</td>
</tr>
<tr>
<td>It is provided by an application to a recognizer to define a set of rules that indicates what a user may say.</td>
<td>It defines a very large number of words which may be spoken in a relatively unrestricted way.</td>
</tr>
<tr>
<td>Rules are defined by tokens, by references to other rules and by logical combinations of tokens</td>
<td>It is built into the recognizer</td>
</tr>
</tbody>
</table>
Table 3.7: Difference between Rule Grammar and Dictation Grammar

<table>
<thead>
<tr>
<th>Rule Grammar</th>
<th>Dictation Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>It can be defined to capture a wide range of spoken input from users by the progressive combination of simple grammars and rules.</td>
<td>It is closest to the goal of unrestricted natural speech input to computers.</td>
</tr>
<tr>
<td>Less flexible</td>
<td>More flexible</td>
</tr>
<tr>
<td>Faster and accurate</td>
<td>Slow and less accurate</td>
</tr>
<tr>
<td>Support for a rule grammar is compulsory for a recognizer.</td>
<td>Support for a dictation grammar is optional for a recognizer.</td>
</tr>
<tr>
<td>A recognizer may have many rule grammars loaded at any time.</td>
<td>Recognizer interface restricts a recognizer to dictation grammar.</td>
</tr>
</tbody>
</table>

The Java Speech API supports dynamic grammars. It supports the capability to modify grammars at runtime. In the case of rule grammars any characteristic of any grammar can be changed at any time.

A grammar may be active or inactive. When a grammar is active the recognizer is matching incoming audio with that grammar to decide whether the anything that user is saying is matches that grammar. Inactive grammar is not used in the recognition. Application to do not directly activate and deactivate grammar, instead they provided methods for:

1. To enable and disable a grammar,
2. To Set the activation mode for grammar, and
3. To request and release the speech focus of a speech recognizer.
3.7.1 Rule Grammar

A rule grammar is defined by a set of rules which are defined by logical combinations of tokens to be spoken and references to other rules. The references may be to other rules defined in the same rule grammar or to rules imported from other grammars. Rule grammars follow the style and conventions of grammars in the Java Speech Grammar Format defined in the Java Speech Grammar Format (JSGF) Specification. Grammar defined in the JSGF can be converted to a RuleGrammar object can be printed out in JSGF.

The simplest way to load a rule grammar is from a Java Speech Grammar Format file or URL using loadJSGF() methods of the Recognizer. If many grammars are to be loaded where a grammar references one or more imported grammars than importing by URL is most easy. The application must specify the base URL and the name of the root grammar to be loaded.

Once a rule grammar has been loaded or created with the newRuleGrammar() method, the following methods of a rule grammar are used to create, edit and manage the rules of the grammar. Any of the below methods take effect only after they are committed.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setRule()</td>
<td>It assigns a Rule object to a rule name.</td>
</tr>
<tr>
<td>getRule()</td>
<td>It returns the Rule object for a rule name.</td>
</tr>
<tr>
<td>getRuleInternal()</td>
<td>It returns a reference to the recognizer’s internal Rule object for a rule name.</td>
</tr>
<tr>
<td>listRuleNames()</td>
<td>It List known rule names.</td>
</tr>
<tr>
<td>isRulePublic()</td>
<td>This method test whether a rule name is public</td>
</tr>
<tr>
<td>deleteRule()</td>
<td>It is used to delete a rule.</td>
</tr>
<tr>
<td>setEnabled()</td>
<td>It is used to enable and disable this Rule Grammar</td>
</tr>
</tbody>
</table>
### Method Name | Description
--- | ---
| isEnabled() | It tests whether a Rule Grammar or a specified rule is enabled.

**Table-3.8: Methods for Rule Grammar**

The rule definition of a rule grammar is a collection of named Rule objects. Each Rule object is referenced by its rule name which is a String. The following is an example of a simple rule grammar. The public rule `<order>` may be spoken in many ways. For example, “push”, “move that door” or “pull that door please”, “Kindly push the window”.

```java
public <order> = <action> [<object>] [<polite>];
<action> = push {OP} | pull {CL} | move {MV};
<object> = [<this_that_etc>] window | door;
<this_that_etc> = a | the | this | that | the current;
<polite> = please | kindly;
```

### 3.7.2 Dictation Grammars

Dictation grammars come nearer to the ultimate objective of a speech recognition system that takes natural spoken input and converted it into text. Dictation grammars are used for text typing in applications such as word processing and email.

A Recognizer which supports dictation provides a single dictation grammar which is obtained from the recognizer’s `getDictationGrammar()` method. A speech recognizer that supports the JSAPI can optionally provide a dictation grammar. Applications that require a recognizer with dictation capability can explicitly request dictation when creating a recognizer by setting the
DictationGrammarSupported property of the RecognizerModeDesc object to true.

A dictation grammar is more complex than a rule grammar but a dictation grammar is generally easier to use than a rule grammar because the dictation grammar is built into the recognizer hence most of the complexity is handled by the recognizer and is hidden from the application. Recognition of a dictation grammar is typically more computationally expensive and less accurate than that of simple rule grammars.

The dictation grammar inherits its basic functionality from the Grammar interface. Like all grammars changes to a dictation grammar require to be committed before its effect take place. A dictation grammar is typically adaptive. A recognizer with dictation improves its performance i.e. accuracy and speed by adapting to the style of language used by a speaker. The recognizer may adapt to the particular sounds of a speaker. A recognizer can adapt to a user's normal vocabulary and to the patterns of those words. Such adaptation is technically known as language model adaptation. The adaptation data for a dictation grammar is maintained as part of a speaker profile. The dictation grammar is derived from Grammar and specialize it by adding the following functionality:

- Suggestion of the current textual context,
- Control of word lists.

Table 3.9 shows methods provided by the dictation grammar interface allow an application to manage word lists and text context.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setContext()</td>
<td>Provide the recognition engine with the preceding and following textual context.</td>
</tr>
</tbody>
</table>
### Table-3.9: Methods for Dictation Grammar

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addWord()</td>
<td>Add a word to the DictationGrammar.</td>
</tr>
<tr>
<td>removeWord()</td>
<td>Remove a word from the DictationGrammar.</td>
</tr>
<tr>
<td>listAddedWords()</td>
<td>List the words that have been added to the DictationGrammar.</td>
</tr>
<tr>
<td>listRemovedWords()</td>
<td>List the words that have been removed from the DictationGrammar.</td>
</tr>
</tbody>
</table>

### 3.8 SPEECH RECOGNITION RESULTS

A recognizer gives result to an application when it listen incoming speech that matches with an active grammar. The result informs the application what words the user said and gives a range of other useful information such as different guesses and audio data.

There are actually two methods to finalize a result which are signaled by the RESULT_ACCEPTED and RESULT_REJECTED events. A result is considered as an accepted when a recognizer is confident that it has correctly heard the words spoken by a user. It means that the tokens in the result exactly represent what a user said.

Rejection takes place when a recognizer is not confident that it has correctly recognized a result which means that the tokens and other information in the result do not necessarily match what a user said. Most applications ignore the RESULT_REJECTED event and ignore the detail of a result when it is rejected. Some applications a RESULT_REJECTED event is used to provide users with feedback that something was heard but no action were taken for example by displaying some symbol or sounding an error beep.
An accepted result may not be a correct result. Recognizers make errors when recognizing speech for a variety of reasons. It means that even for an accepted result developers should consider the possible impact of a false recognition. Misrecognition could cause an action with serious consequences or could make changes that can’t be undone such as delete all files. The application should perform any action after checking with users.

The speech recognition systems are constantly improving so the number of errors is steadily decreasing but there will be a possibility of a misunderstanding. Result interface is available for any result in any state finalized or un-finalized and matching any grammar. Table 3.10 shows the important methods of ResultSet

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getResultState()</td>
<td>It returns the current state of the result which are values UNFINALIZED, ACCEPTED and REJECTED defined by static values of the Result interface</td>
</tr>
<tr>
<td>getGrammar()</td>
<td>It returns a reference to the matched Grammar. When result is ACCEPTED, this method will return a RuleGrammar or a DictationGrammar. When a result is REJECTED, this method may return a grammar, or null if the recognizer could not identify the grammar for this result.</td>
</tr>
<tr>
<td>numTokens()</td>
<td>It returns the total number of finalized tokens for a result. Return value may be zero or greater for an un-finalized result. For a finalized result this number is always greater than zero for an ACCEPTED result but may be zero or more for a REJECTED</td>
</tr>
</tbody>
</table>

Automatic Text Conversion of Continuous Speech for Indian Languages
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBestToken() and getBestTokens()</td>
<td>The methods return either a specified finalized best-guess token of a result or all the finalized best-guess tokens.</td>
</tr>
<tr>
<td>getUnfinalizedTokens()</td>
<td>It returns a list of un-finalized tokens. An un-finalized token is a recognizer’s current guess of what a user has said. For a finalized result it always returns null.</td>
</tr>
</tbody>
</table>

Table-3.10: Methods for Result

A Result is produced in response to a user’s speech. Because speech is not instantaneous, a speech recognition result is not produced instantaneously. A Result is produced through a sequence of events beginning some time after a user starts speaking and finishing some time after the user stops speaking.

Every result starts in the UNFINALIZED state when a RESULT_CREATED event is issued. If result is un-finalized then the recognizer gives information including finalized and un-finalized tokens and the identity of the grammar matched by the result and the RESULT_UPDATED and GRAMMAR_FINALIZED events are issued.

Once all information associated with a result is finalized, the entire result is finalized. If result is finalized with either the ACCEPTED or REJECTED state then all information associated with the result becomes available including the best guess tokens and the information provided through the three final result interfaces. For finalized result the information available through all the result interfaces is fixed. If audio data is released then an AUDIO_RELEASED event is issued.
If training information is released then TRAINING_INFO_RELEASED event is issued. Applications can track result states in a many ways. Generally applications handle result in ResultListener implementation which receives ResultEvents as recognition proceeds.

![Result States Associated with Result Event](image)

Figure-3.4: Result States Associated with Result Event

A recognizer provides a range of information to an application through the stages of producing a recognition result. However, many applications only care about the last step and event in that process the RESULT_ACCEPTED event. The state of a result is available through the getResultState() method of the Result interface which returns one of the three result states i.e. UNFINALIZED, ACCEPTED or REJECTED.

### 3.9 SPEECH RECOGNIZER PROPERTIES

A speech engine has both read only and run time modifiable properties. A recognizer has seven run time changeable properties. Applications get and set these properties through RecognizerProperties which extends the EngineProperties interface. Table 3.11 lists run-
time properties of a recognizer with its corresponding value and brief description of the property.

<table>
<thead>
<tr>
<th>Name of Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfidenceLevel</td>
<td>Its value is in the range zero to one. Results will be rejected if the engine is not confident about determined spoken text. A value of one requires a recognizer to have maximum confidence in every result, hence more results are possible to be rejected. A value of zero means low confidence and indicating less rejection. 0.5 is the default value for recognizer.</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>Its value is in the range zero and one. A value of 0.5 is the default for the recognizer. The value one gives maximum sensitivity hence making the recognizer sensitive to quiet input however more sensitive to noise also. Zero gives minimum sensitivity hence user has to speak loudly and making the recognizer less sensitive to background noise.</td>
</tr>
<tr>
<td>SpeedVsAccuracy</td>
<td>Its value is in the range zero and one. The value zero provides the best response. One maximizes recognition accuracy. The default value for the recognizer is 0.5 which is considered as the best compromise between speed and accuracy.</td>
</tr>
</tbody>
</table>
| CompleteTimeout        | The value is in seconds and indicates the minimum time between silence starts and finalizing a result. It is applied when the speech before the silence matches with an active grammar. A short time out may
### Name of Property | Description
---|---
| | results in an utterance being broken up inappropriately e.g. when the user takes a breath. Total time out values are generally in the range of 0.3 seconds to one second.
| IncompleteTimeout | Its value is in seconds and indicates the minimum period between silence starts and the recognizer finalizing a result. It is applied when the speech before the silence does not match an active grammar. This is the time the recognizer will wait before rejecting an incomplete utterance. The Incomplete Timeout is typically longer than the Complete Timeout.
| ResultNumAlternatives | It is an integer value representing the desired maximum number of N-best alternatives in FinalDictationResult and FinalRuleResult objects. It requires additional computation. Recognizers don’t always generate the maximum number of alternatives and the number of alternatives may differ between results and between tokens. A value of 0 or 1 needs that no alternatives are provided but only a best guess.
| ResultAudioProvided | It specifies the Boolean value indicating whether the application wants the recognizer to return audio with FinalResult objects or not. If recognizers do not provide result audio then it can ignore this call.
| TrainingProvided | It specifies the Boolean value indicating whether the application wants the
### 3.10 CONCLUSION

The fundamental characteristics of a speech recognizer supporting the Java Speech API are:

- It supports a single specified language.
- It can optionally identify the voice of its users.
- It processes a single input audio stream at a time.
- Its grammars can be dynamically updated.
- It has a small set of application controllable properties.

The major steps of any speech recognizer are designing a grammar, processing speech signal, identifying phoneme and generating appropriate result.

If application is using a rule based speech recognition system than it provides the recognizer with rules that define what the user is expected to say. These rules restrict the recognition process. Careful design of the rules with careful user interface design can produce rules that allow users reasonable freedom of expression though still limiting the range of things that may be said. It makes the recognition process as fast and accurate as possible.

If the application is using dictation grammars then it impose fewer restrictions on what can be said and providing the free form speech input. The disadvantage is that it requires more computing resources;

<table>
<thead>
<tr>
<th>Name of Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recognizer</td>
<td>support training with FinalResult objects.</td>
</tr>
</tbody>
</table>

*Table-3.11: Runtime Properties of a Recognizer*
higher quality audio input and be likely to make more errors. The following is a list of major factors that affect recognition accuracy:

- A quiet environment has generally higher recognition accuracy.
- High quality microphones and audio hardware can improve recognition accuracy.
- Users that speak clearly and naturally generally get better accuracy.
- An application with simpler grammars generally gets better accuracy.
- Users with accents or typical voices may get lower accuracy.
- Similar sounding words are difficult to differentiate by the recognizer.

Most recognition errors can be classified into the following types:

- **Rejection**: the recognizer can’t understand what was said.
- **Misrecognition**: This is the most common type of recognition error in which the recognizer returns a result with words that are different from what the user spoke.
- **Misfire**: in this kind of error, the user doesn’t speak but the recognizer returns a result.