Java™ Speech API Programmer's Guide

Version 1.0 — October 26, 1998

THE NETWORK IS THE COMPUTER™

A Sun Microsystems, Inc. Business
901 San Antonio Road
Palo Alto, CA 94303 USA
415 960-1300 Fax 415 969-9131

Table of Contents

List of Figures

List of Tables

Preface

The *Java™ Speech API Programmer's Guide* is an introduction to speech technology and to the development of effective speech applications using the Java Speech API. An understanding of the Java programming language and the core Java APIs is assumed. An understanding of speech technology is not required.

About this Guide

Part 1

The first three chapters of this guide provide an introduction to speech technology and to the Java Speech API.

Chapter 1, Introduction: This chapter introduces the Java Speech API, reviews the design goals for JSAPI, discusses the types of applications that JSAPI enables and discusses the requirements for using JSAPI.

Chapter 2, Speech Technology: This chapter is a more detailed introduction to speech technology systems with a greater focus on technical issues. It describes both the capabilities and limitations of speech recognition and speech synthesis systems. An understanding of these issues is important to developers who use speech technology in their applications.

Chapter 3, Designing Effective Speech Applications: This chapter is an introduction to the art and the science of effective user interface design with speech technology. As with design of graphical interfaces using AWT or the Java Foundation Classes, good user interface design with speech is important to ensure that applications are usable. Chapter 3 also discusses how some of the limitations of speech technology need to be considered in the design of an effective speech-enabled user interface.

Part 2

The next three chapters of this guide describe in technical detail the three Java software packages that comprise the Java Speech API. These chapters provide both introductory and advanced descriptions of programming with the Java Speech API. Where possible, code examples are included to illustrate the principles of speech application development.

Chapter 4, Speech Engines: javax.speech: introduces the root package of the Java Speech API. The classes and interfaces of the root package define basic speech engine functionality.

Chapter 5, Speech Synthesis: javax.speech.synthesis: introduces the package that supports speech synthesis capabilities. A speech synthesizer is a type of speech engine. Thus, the speech synthesis package inherits the general speech engine behavior from the `javax.speech` package but extends it with the ability to produce speech output.

Chapter 6, Speech Recognition: javax.speech.recognition: introduces the package that supports speech recognition capabilities. A speech recognizer is also a type of speech engine, so the speech recognition package also extends the general speech engine behavior, in this case, with the ability to convert incoming audio to text.

Getting Started

Newcomers to speech technology are encouraged to read Chapter 2 and then to consider the “Hello World!” code examples at the start of both Chapters 5 and 6. These code examples illustrate the basics of speech synthesis and speech recognition programming.

All developers are encouraged to read Chapter 3 on Designing Effective Speech Applications.” Appropriate and effective use of speech input and output makes the development of speech applications easier and faster and improves the experience that application users will have.

Finally, as with all Java APIs, the Javadoc for the three Java Speech API packages is the master description of the functionality of the API and documents every API capability.

Web Resources

To obtain the latest information on the Java Speech API other Java Media APIs visit:

```
http://java.sun.com/products/java-media/speech/index.html
```

Related Reading

This document describes the software interface of the Java Speech API. For information on related topics, refer to the following:

- Java Speech Markup Language Specification
- Java Speech Grammar Format Specification

Both documents are available from the Java Speech API home page:

```
http://java.sun.com/products/java-media/speech/index.html
```

Mailing Lists

Discussion lists have been set up for everyone interested in the Java Speech API, the Java Speech Grammar Format specification, the Java Synthesis Markup Language, and related technologies. The `javaspeech-announce` mailing list carries important announcements about releases and updates. The `javaspeech-interest` mailing list is for open discussion of the Java Speech API and the associated specifications.

To subscribe to the `javaspeech-announce` list or the `javaspeech-interest` list, send email with "`subscribe javaspeech-announce`" or "`subscribe javaspeech-interest`" or both in the message body to:

```
javamedia-request@sun.com
```

The `javaspeech-announce` mailing list is moderated. It is not possible to send email to that list. To send messages to the interest list, send email to:

```
javaspeech-interest@sun.com
```

To unsubscribe from the `javaspeech-announce` list or the `javaspeech-interest` list, send email with "`unsubscribe javaspeech-announce`" or "`unsubscribe javaspeech-interest`" or both in the message body to:

```
javamedia-request@sun.com
```

Comments and proposals for enhancements should be sent to:

```
javaspeech-comments@sun.com
```

Revision History

Version 1.0: October 26, 1998

Version 0.7: May, 1998. Revised public beta release.

Version 0.6: February 98. Initial public beta release

Contributions

Sun Microsystems, Inc. has worked in partnership with leading speech technology companies to define the specification of the Java Speech API. These companies brought decades of research on speech technology to the project as well as experience in the development and use of speech applications. Sun is grateful for the contributions of:

- Apple Computer, Inc.
- AT&T
- Dragon Systems, Inc.
- IBM Corporation
- Novell, Inc.
- Philips Speech Processing
- Texas Instruments Incorporated

CHAPTER 1

Introduction

Speech technology, once limited to the realm of science fiction, is now available for use in real applications. The *Java™ Speech API*, developed by Sun Microsystems in cooperation with speech technology companies, defines a software interface that allows developers to take advantage of speech technology for personal and enterprise computing. By leveraging the inherent strengths of the Java platform, the Java Speech API enables developers of speech-enabled applications to incorporate more sophisticated and natural user interfaces into Java applications and applets that can be deployed on a wide range of platforms.

1.1 What is the Java Speech API?

The Java Speech API defines a standard, easy-to-use, cross-platform software interface to state-of-the-art speech technology. Two core speech technologies are supported through the Java Speech API: *speech recognition* and *speech synthesis*. Speech recognition provides computers with the ability to listen to spoken language and to determine what has been said. In other words, it processes audio input containing speech by converting it to text. Speech synthesis provides the reverse process of producing synthetic speech from text generated by an application, an applet or a user. It is often referred to as *text-to-speech* technology.

Enterprises and individuals can benefit from a wide range of applications of speech technology using the Java Speech API. For instance, interactive voice response systems are an attractive alternative to touch-tone interfaces over the telephone; dictation systems can be considerably faster than typed input for many users; speech technology improves accessibility to computers for many people with physical limitations.

Speech interfaces give Java application developers the opportunity to implement distinct and engaging personalities for their applications and to differentiate their products. Java application developers will have access to state-of-the-art speech technology from leading speech companies. With a standard

API for speech, users can choose the speech products which best meet their needs and their budget.

The Java Speech API was developed through an open development process. With the active involvement of leading speech technology companies, with input from application developers and with months of public review and comment, the specification has achieved a high degree of technical excellence. As a specification for a rapidly evolving technology, Sun will support and enhance the Java Speech API to maintain its leading capabilities.

The Java Speech API is an extension to the Java platform. Extensions are packages of classes written in the Java programming language (and any associated native code) that application developers can use to extend the functionality of the core part of the Java platform.

1.2 Design Goals for the Java Speech API

Along with the other Java Media APIs, the Java Speech API lets developers incorporate advanced user interfaces into Java applications. The design goals for the Java Speech API included:

- Provide support for speech synthesizers and for both command-and-control and dictation speech recognizers.
- Provide a robust cross-platform, cross-vendor interface to speech synthesis and speech recognition.
- Enable access to state-of-the-art speech technology.
- Support integration with other capabilities of the Java platform, including the suite of Java Media APIs.
- Be simple, compact and easy to learn.

1.3 Speech-Enabled Java Applications

The existing capabilities of the Java platform make it attractive for the development of a wide range of applications. With the addition of the Java Speech API, Java application developers can extend and complement existing user interfaces with speech input and output. For existing developers of speech applications, the Java platform now offers an attractive alternative with:

- *Portability:* the Java programming language, APIs and virtual machine are available for a wide variety of hardware platforms and operating systems and are supported by major web browsers.

- *Powerful and compact environment:* the Java platform provides developers with a powerful, object-oriented, garbage collected language which enables rapid development and improved reliability.
- *Network aware and secure:* from its inception, the Java platform has been network aware and has included robust security.

1.3.1 Speech and other Java APIs

The Java Speech API is one of the Java Media APIs, a suite of software interfaces that provide cross-platform access to audio, video and other multimedia playback, 2D and 3D graphics, animation, telephony, advanced imaging, and more. The Java Speech API, in combination with the other Java Media APIs, allows developers to enrich Java applications and applets with rich media and communication capabilities that meet the expectations of today's users, and can enhance person-to-person communication.

The Java Speech API leverages the capabilities of other Java APIs. The Internationalization features of the Java programming language plus the use of the Unicode character set simplify the development of multi-lingual speech applications. The classes and interfaces of the Java Speech API follow the design patterns of JavaBeans™. Finally, Java Speech API events integrate with the event mechanisms of AWT, JavaBeans and the Java Foundation Classes (JFC).

1.4 Applications of Speech Technology

Speech technology is becoming increasingly important in both personal and enterprise computing as it is used to improve existing user interfaces and to support new means of human interaction with computers. Speech technology allows hands-free use of computers and supports access to computing capabilities away from the desk and over the telephone. Speech recognition and speech synthesis can improve computer accessibility for users with disabilities and can reduce the risk of repetitive strain injury and other problems caused by current interfaces.

The following sections describe some current and emerging uses of speech technology. The lists of uses are far from exhaustive. New speech products are being introduced on a weekly basis and speech technology is rapidly entering new technical domains and new markets. The coming years should see speech input and output truly revolutionize the way people interact with computers and present new and unforeseen uses of speech technology.

1.4.1 Desktop

Speech technology can augment traditional graphical user interfaces. At its simplest, it can be used to provide audible prompts with spoken *"Yes/No/OK"* responses that do not distract the user's focus. But increasingly, complex commands are enabling rapid access to features that have traditionally been buried in sub-menus and dialogs. For example, the command *"Use 12-point, bold, Helvetica font"* replaces multiple menu selections and mouse clicks.

Drawing, CAD and other hands-busy applications can be enhanced by using speech commands in combination with mouse and keyboard actions to improve the speed at which users can manipulate objects. For example, while dragging an object, a speech command could be used to change its color and line type all without moving the pointer to the menu-bar or a tool palette.

Natural language commands can provide improvements in efficiency but are increasingly being used in desktop environments to enhance usability. For many users it's easier and more natural to produce spoken commands than to remember the location of functions in menus and dialog boxes. Speech technology is unlikely to make existing user interfaces redundant any time soon, but spoken commands provide an elegant complement to existing interfaces.

Speech dictation systems are now affordable and widely available. Dictation systems can provide typing rates exceeding 100 words per minute and word accuracy over 95%. These rates substantially exceed the typing ability of most people.

Speech synthesis can enhance applications in many ways. Speech synthesis of text in a word processor is a reliable aid to proof-reading, as many users find it easier to detect grammatical and stylistic problems when listening rather than reading. Speech synthesis can provide background notification of events and status changes, such as printer activity, without requiring a user to lose current context. Applications which currently include speech output using pre-recorded messages can be enhanced by using speech synthesis to reduce the storage space by a factor of up to 1000, and by removing the restriction that the output sentences be defined in advance.

In many situations where keyboard input is impractical and visual displays are restricted, speech may provide the only way to interact with a computer. For example, surgeons and other medical staff can use speech dictation to enter reports when their hands are busy and when touching a keyboard represents a hygiene risk. In vehicle and airline maintenance, warehousing and many other hands-busy tasks, speech interfaces can provide practical data input and output and can enable computer-based training.

1.4.2 Telephony Systems

Speech technology is being used by many enterprises to handle customer calls and internal requests for access to information, resources and services. Speech recognition over the telephone provides a more natural and substantially more efficient interface than touch-tone systems. For example, speech recognition can "flatten out" the deep menu structures used in touch tone systems.

Systems are already available for telephone access to email, calendars and other computing facilities that have previously been available only on the desktop or with special equipment. Such systems allow convenient computer access by telephones in hotels, airports and airplanes.

Universal messaging systems can provide a single point of access to multiple media such as voice-mail, email, fax and pager messages. Such systems rely upon speech synthesis to read out messages over the telephone. For example: *"Do I have any email?"* "*Yes, you have 7 messages including 2 high priority messages from the production manager." "Please read me the mail from the production manager." "Email arrived at 12:30pm..."*.

1.4.3 Personal and Embedded Devices

Speech technology is being integrated into a range of small-scale and embedded computing devices to enhance their usability and reduce their size. Such devices include Personal Digital Assistants (PDAs), telephone handsets, toys and consumer product controllers.

Speech technology is particularly compelling for such devices and is being used increasingly as the computer power of these devices increases. Speech recognition through a microphone can replace input through a much larger keyboard. A speaker for speech synthesis output is also smaller than most graphical displays.

PersonalJava™ and EmbeddedJava™ are the Java application environments targeted at these same devices. PersonalJava and EmbeddedJava are designed to operate on constrained devices with limited computing power and memory, and with more constrained input and output mechanisms for the user interface.

As an extension to the Java platform, the Java Speech API can be provided as an extension to PersonalJava and EmbeddedJava devices, allowing the devices to communicate with users without the need for keyboards or other large peripherals.

1.4.4 Speech and the Internet

The Java Speech API allows applets transmitted over the Internet or intranets to access speech capabilities on the user's machine. This provides the ability to enhance World Wide Web sites with speech and support new ways of browsing. Speech recognition can be used to control browsers, fill out forms, control applets and enhance the WWW/Internet experience in many other ways. Speech synthesis can be used to bring web pages alive, inform users of the progress of applets, and dramatically improve browsing time by reducing the amount of audio sent across the Internet.

The Java Speech API utilizes the security features of the Java platform to ensure that applets cannot maliciously use system resources on a client. For example, explicit permission is required for an applet to access a dictation recognizer since otherwise a recognizer could be used to bug a user's workspace.

1.5 Implementations

The Java Speech API can enable access to the most important and useful state-of-the-art speech technologies. Sun is working with speech technology companies on implementations of the API. Already speech recognition and speech synthesis are available through the Java Speech API on multiple computing platforms.

The following are the primary mechanisms for implementing the API.

- *Native implementations:* most existing speech technology is implemented in C and C++ and accessed through platform-specific APIs such as the Apple Speech Managers and Microsoft's Speech API (SAPI), or via proprietary vendor APIs. Using the Java Native Interface (JNI) and Java software wrappers, speech vendors can (and have) implemented the Java Speech API on top of their existing speech software.
- *Java software implementations:* Speech synthesizers and speech recognizers can be written in Java software. These implementations will benefit from the portability of the Java platform and from the continuing improvements in the execution speed of Java virtual machines.
- *Telephony implementations:* Enterprise telephony applications are typically implemented with dedicated hardware to support a large number of simultaneous connections, for example, using DSP cards. 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 type of native implementation.

1.6 Requirements

To use the Java Speech API, a user must have certain minimum software and hardware available. The following is a broad sample of requirements. The individual requirements of speech synthesizers and speech recognizers can vary greatly and users should check product requirements closely.

- *Speech software:* A JSAPI-compliant speech recognizer or synthesizer is required.
- *System requirements:* most desktop speech recognizers and some speech synthesizers require relatively powerful computers to run effectively. Check the minimum and recommended requirements for CPU, memory and disk space when purchasing a speech product.
- *Audio Hardware:* Speech synthesizers require audio output. Speech recognizers require audio input. Most desktop and laptop computers now sold have satisfactory audio support. Most dictation systems perform better with good quality sound cards.
- *Microphone:* Desktop speech recognition systems get audio input through a microphone. Some recognizers, especially dictation systems, are sensitive to the microphone and most recognition products recommend particular microphones. Headset microphones usually provide best performance, especially in noisy environments. Table-top microphones can be used in some environments for some applications.

CHAPTER 2

Speech Technology

As an emerging technology, not all developers are familiar with speech technology. While the basic functions of both speech synthesis and speech recognition take only minutes to understand (after all, most people learn to speak and listen by age two), there are subtle and powerful capabilities provided by computerized speech that developers will want to understand and utilize.

Despite very substantial investment in speech technology research over the last 40 years, speech synthesis and speech recognition technologies still have significant limitations. Most importantly, speech technology does not always meet the high expectations of users familiar with natural human-to-human speech communication. Understanding the limitations — as well as the strengths — is important for effective use of speech input and output in a user interface and for understanding some of the advanced features of the Java Speech API.

An understanding of the capabilities and limitations of speech technology is also important for developers in making decisions about whether a particular application will benefit from the use of speech input and output. Chapter 3 expands on this issue by considering when and where speech input and output can enhance human-to-computer communication.

2.1 Speech Synthesis

A speech synthesizer converts written text into spoken language. Speech synthesis is also referred to as *text-to-speech* (TTS) conversion.

The major steps in producing speech from text are as follows:

- *Structure analysis*: process the input text to determine where paragraphs, sentences and other structures start and end. For most languages, punctuation and formatting data are used in this stage.
- *Text pre-processing*: analyze the input text for special constructs of the

language. In English, special treatment is required for abbreviations, acronyms, dates, times, numbers, currency amounts, email addresses and many other forms. Other languages need special processing for these forms and most languages have other specialized requirements.

The result of these first two steps is a spoken form of the written text. The following are examples of the difference between written and spoken text.

```
St. Mathews hospital is on Main St.
-> "Saint Mathews hospital is on Main street"

Add $20 to account 55374.
-> "Add twenty dollars to account five five, three seven four."

Leave at 5:30 on 5/15/99.
-> "Leave at five thirty on May fifteenth nineteen ninety nine."
```

The remaining steps convert the spoken text to speech.

- *Text-to-phoneme conversion*: convert each word to *phonemes*. A phoneme is a basic unit of sound in a language. US English has around 45 phonemes including the consonant and vowel sounds. For example, “times” is spoken as four phonemes “t ay m s”. Different languages have different sets of sounds (different phonemes). For example, Japanese has fewer phonemes including sounds not found in English, such as “ts” in “tsunami”.
- *Prosody analysis*: process the sentence structure, words and phonemes to determine appropriate *prosody* for the sentence. Prosody includes many of the features of speech other than the sounds of the words being spoken. This includes the pitch (or melody), the timing (or rhythm), the pausing, the speaking rate, the emphasis on words and many other features. Correct prosody is important for making speech sound right and for correctly conveying the meaning of a sentence.
- *Waveform production*: finally, the phonemes and prosody information are used to produce the audio waveform for each sentence. There are many ways in which the speech can be produced from the phoneme and prosody information. Most current systems do it in one of two ways: *concatenation* of chunks of recorded human speech, or *formant synthesis* using signal processing techniques based on knowledge of how phonemes sound and how prosody affects those phonemes. The details of waveform generation are not typically important to application developers.

2.1.1 Speech Synthesis Limitations

Speech synthesizers can make errors in any of the processing steps described above. Human ears are well-tuned to detecting these errors, so careful work by developers can minimize errors and improve the speech output quality.

The Java Speech API and the *Java Speech Markup Language* (JSML) provide many ways for an application developer to improve the output quality of a speech synthesizer. Chapter 5 describes programming techniques for controlling a synthesis through the Java Speech API.

The Java Synthesis Markup Language defines how to markup text input to a speech synthesizer with information that enables the synthesizer to enhance the speech output quality. It is described in detail in the Java Synthesis Markup Language Specification. In brief, some of its features which enhance quality include:

- Ability to mark the start and end of paragraphs and sentences.
- Ability to specify pronunciations for any word, acronym, abbreviation or other special text representation.
- Explicit control of pauses, boundaries, emphasis, pitch, speaking rate and loudness to improve the output prosody.

These features allow a developer or user to override the behavior of a speech synthesizer to correct most of the potential errors described above. The following is a description of some of the sources of errors and how to minimize problems.

- *Structure analysis*: punctuation and formatting do not consistently indicate where paragraphs, sentences and other structures start and end. For example, the final period in "U.S.A." might be misinterpreted as the end of a sentence.
  Try: Explicitly marking paragraphs and sentences in JSML reduces the number of structural analysis errors.
- *Text pre-processing*: it is not possible for a synthesizer to know all the abbreviations and acronyms of a language. It is not always possible for a synthesizer to determine how to process dates and times, for example, is "8/5" the "eighth of May" of the "fifth of August"? Should "1998" be read as "nineteen ninety eight" (as a year), as "one thousand and ninety eight" (a regular number) or as "one nine nine eight" (part of a telephone number). Special constructs such as email addresses are particularly difficult to interpret, for example, should a synthesizer say "tedwards@cat.com" as "Ted Wards", as "T. Edwards", as "Cat dot com" or as "C. A. T. dot com"?

Try: The SAYAS element of JSML supports substitutions of text for abbreviations, acronyms and other idiosyncratic textual forms.

- *Text-to-phoneme conversion*: most synthesizers can pronounce tens of thousands or even hundreds of thousands of words correctly. However, there are always new words which it must guess for (especially proper names for people, companies, products, etc.), and words for which the pronunciation is ambiguous (for example, "object" as "OBject" or "obJECT", or "row" as a line or as a fight).
  Try: The SAYAS element of JSML is used to provide phonetic pronunciations for unusual and ambiguous words.
- *Prosody analysis*: to correctly phrase a sentence, to produce the correct melody for a sentence, and to correctly emphasize words ideally requires an understanding of the meaning of languages that computers do not possess. Instead, speech synthesizers must try to guess what a human might produce and at times, the guess is artificial and unnatural.
  Try: The EMP, BREAK and PROS elements of JSML can be used to indicate preferred emphasis, pausing, and prosodic rendering respectively for text.
- *Waveform production*: without lips, mouths, lungs and the other apparatus of human speech, a speech synthesizer will often produce speech which sounds artificial, mechanical or otherwise different from human speech. In some circumstances a robotic sound is desirable, but for most applications speech that sounds as close to human as possible is easier to understand and easier to listen to for long periods of time.
  Try: The Java Speech API and JSML do not directly address this issue.

2.1.2 Speech Synthesis Assessment

The major feature of a speech synthesizer that affects its understandability, its acceptance by users and its usefulness to application developers is its output quality. Knowing how to evaluate speech synthesis quality and knowing the factors that influence the output quality are important in the deployment of speech synthesis.

Humans are conditioned by a lifetime of listening and speaking. The human ear (and brain) are very sensitive to small changes in speech quality. A listener can detect changes that might indicate a user's emotional state, an accent, a speech problem or many other factors. The quality of current speech synthesis remains below that of human speech, so listeners must make more effort than normal to understand synthesized speech and must ignore errors. For new users, listening to a speech synthesizer for extended periods can be tiring and unsatisfactory.

The two key factors a developer must consider when assessing the quality of a speech synthesizer are its *understandability* and its *naturalness*. Understandability is an indication of how reliably a listener will understand the words and sentences spoken by the synthesizer. Naturalness is an indication of the extent to which the synthesizer sounds like a human - a characteristic that is desirable for most, but not all, applications.

Understandability is affected by the ability of a speech synthesizer to perform all the processing steps described above because any error by the synthesizer has the potential to mislead a listener. Naturalness is affected more by the later stages of processing, particularly the processing of prosody and the generation of the speech waveform.

Though it might seem counter-intuitive, it is possible to have an artificial-sounding voice that is highly understandable. Similarly, it is possible to have a voice that sounds natural but is not always easy to understand (though this is less common).

2.2 Speech Recognition

Speech recognition is the process of converting spoken language to written text or some similar form. The basic characteristics of a speech recognizer supporting the Java Speech API are:

- It is mono-lingual: it supports a single specified language.
- It processes a single input audio stream.
- It can optionally adapt to the voice of its users.
- Its grammars can be dynamically updated.
- It has a small, defined set of application-controllable properties.

The major steps of a typical speech recognizer are:

- *Grammar design*: recognition grammars define the words that may be spoken by a user and the patterns in which they may be spoken. A grammar must be created and activated for a recognizer to know what it should listen for in incoming audio. Grammars are described below in more detail.
- *Signal processing*: analyze the spectrum (frequency) characteristics of the incoming audio.
- *Phoneme recognition*: compare the spectrum patterns to the patterns of the phonemes of the language being recognized. (A brief description of

phonemes is provided in the “Speech Synthesis” section in the discussion of text-to-phoneme conversion.)

- *Word recognition*: compare the sequence of likely phonemes against the words and patterns of words specified by the active grammars.
- *Result generation*: provide the application with information about the words the recognizer has detected in the incoming audio. The result information is always provided once recognition of a single utterance (often a sentence) is complete, but may also be provided during the recognition process. The result always indicates the recognizer’s best guess of what a user said, but may also indicate alternative guesses.

Most of the processes of a speech recognizer are automatic and are not controlled by the application developer. For instance, microphone placement, background noise, sound card quality, system training, CPU power and speaker accent all affect recognition performance but are beyond an application’s control.

The primary way in which an application controls the activity of a recognizer is through control of its *grammars*.

A grammar is an object in the Java Speech API which indicates what words a user is expected to say and in what patterns those words may occur. Grammars are important to speech recognizers because they constrain the recognition process. These constraints makes recognition faster and more accurate because the recognizer does not have to check for bizarre sentences, for example, “pink is recognizer speech my”.

The Java Speech API supports two basic grammar types: *rule grammars* and *dictation grammars*. These grammar types differ in the way in which applications set up the grammars, the types of sentences they allow, the way in which results are provided, the amount of computational resources required, and the way in which they are effectively used in application design. The grammar types are describe in more detail below. The programmatic control of grammars is detailed in Chapter 6.

Other speech recognizer controls available to a Java application include pausing and resuming the recognition process, direction of result events and other events relating to the recognition processes, and control of the recognizer’s vocabulary.

2.2.1 Rule Grammars

In a rule-based speech recognition system, an application provides the recognizer with rules that define what the user is expected to say. These rules constrain the recognition process. Careful design of the rules, combined with careful user interface design, will produce rules that allow users reasonable freedom of expression while still limiting the range of things that may be said so that the recognition process is as fast and accurate as possible.

Any speech recognizer that supports the Java Speech API must support rule grammars.

The following is an example of a simple rule grammar. It is represented in the Java Speech Grammar Format (JSGF) which is defined in detail in the Java Speech Grammar Format Specification.

```
#JSGF V1.0;
// Define the grammar name
grammar SimpleCommands;
// Define the rules
public <Command> = [<Polite>] <Action> <Object> (and <Object>)*;
<Action> = open | close | delete;
<Object> = the window | the file;
<Polite> = please;
```

Rule names are surrounded by angle brackets. Words that may be spoken are written as plain text. This grammar defines one *public* rule, `<Command>`, that may be spoken by users. This rule is a combination of three sub-rules, `<Action>`, `<Object>` and `<Polite>`. The square brackets around the reference to `<Polite>` mean that it is optional. The parentheses around "`and <Object>`" group the word and the rule reference together. The asterisk following the group indicates that it may occur zero or more times.

The grammar allows a user to say commands such as "Open the window" and "Please close the window and the file".

The Java Speech Grammar Format Specification defines the full behavior of rule grammars and discusses how complex grammars can be constructed by combining smaller grammars. With JSGF application developers can reuse grammars, can provide Javadoc-style documentation and can use the other facilities that enable deployment of advanced speech systems.

2.2.2 Dictation Grammars

Dictation grammars impose fewer restrictions on what can be said, making them closer to providing the ideal of free-form speech input. The cost of this greater freedom is that they require more substantial computing resources, require higher quality audio input and tend to make more errors.

A dictation grammar is typically larger and more complex than rule-based grammars. Dictation grammars are typically developed by statistical training on large collections of written text. Fortunately, developers don't need to know any of this because a speech recognizer that supports a dictation grammar through the Java Speech API has a built-in dictation grammar. An application that needs to use that dictation grammar simply requests a reference to it and enables it when the user might say something matching the dictation grammar.

Dictation grammars may be optimized for particular kinds of text. Often a dictation recognizer may be available with dictation grammars for general purpose text, for legal text, or for various types of medical reporting. In these different domains, different words are used, and the patterns of words also differ.

A dictation recognizer in the Java Speech API supports a single dictation grammar for a specific domain. The application and/or user selects an appropriate dictation grammar when the dictation recognizer is selected and created.

2.2.3 Limitations of Speech Recognition

The two primary limitations of current speech recognition technology are that it does not yet transcribe free-form speech input, and that it makes mistakes. The previous sections discussed how speech recognizers are constrained by grammars. This section considers the issue of recognition errors.

Speech recognizers make mistakes. So do people. But recognizers usually make more. Understanding why recognizers make mistakes, the factors that lead to these mistakes, and how to train users of speech recognition to minimize errors are all important to speech application developers.

The reliability of a speech recognizer is most often defined by its *recognition accuracy*. Accuracy is usually given as a percentage and is most often the percentage of correctly recognized words. Because the percentage can be measured differently and depends greatly upon the task and the testing conditions it is not always possible to compare recognizers simply by their percentage recognition accuracy. A developer must also consider the seriousness of recognition errors: misrecognition of a bank account number or the command “delete all files” may have serious consequences.

The following is a list of major factors that influence recognition accuracy.

- Recognition accuracy is usually higher in a quiet environment.
- Higher-quality microphones and audio hardware can improve accuracy.
- Users that speak clearly (but naturally) usually achieve better accuracy.
- Users with accents or atypical voices may get lower accuracy.
- Applications with simpler grammars typically get better accuracy.
- Applications with less *confusable* grammars typically get better accuracy. Similar sounding words are harder to distinguish.

While these factors can all be significant, their impact can vary between recognizers because each speech recognizer optimizes its performance by trading off various criteria. For example, some recognizers are designed to work reliably in high-noise environments (e.g. factories and mines) but are restricted to very simple grammars. Dictation systems have complex grammars but require good microphones, quieter environments, clearer speech from users and more powerful computers. Some recognizers adapt their process to the voice of a particular user to improve accuracy, but may require training by the user. Thus, users and application developers often benefit by selecting an appropriate recognizer for a specific task and environment.

Only some of these factors can be controlled programmatically. The primary application-controlled factor that influences recognition accuracy is grammar complexity. Recognizer performance can degrade as grammars become more complex, and can degrade as more grammars are active simultaneously. However, making a user interface more natural and usable sometimes requires the use of more complex and flexible grammars. Thus, application developers often need to consider a trade-off between increased usability with more complex grammars and the decreased recognition accuracy this might cause. These issues are discussed in more detail in Chapter 3 which discusses the effective design of user interfaces with speech technology.

Most recognition errors fall into the following categories:

- *Rejection*: the user speaks but the recognizer cannot understand what was said. The outcome is that the recognizer does not produce a successful recognition result. In the Java Speech API, applications receive an event that indicates the rejection of a result.
- *Misrecognition*: the recognizer returns a result with words that are different from what the user spoke. This is the most common type of recognition error.
- *Misfire*: the user does not speak but the recognizer returns a result.

Table 2-1 lists some of the common causes of the three types of recognition errors.

Table 2-1 Speech recognition errors and possible causes

Problem	Cause
Rejection or Misrecognition	User speaks one or more words not in the vocabulary.
	User's sentence does not match any active grammar.
	User speaks before system is ready to listen.
	Words in active vocabulary sound alike and are confused (e.g., "too", "two").
	User pauses too long in the middle of a sentence.
	User speaks with a disfluency (e.g., restarts sentence, stumbles, "umm", "ah").
	User's voice trails off at the end of the sentence.
	User has an accent or cold.
	User's voice is substantially different from stored "voice models" (often a problem with children).
	Computer's audio is not configured properly.
	User's microphone is not properly adjusted.
Misfire	Non-speech sound (e.g., cough, laugh).
	Background speech triggers recognition.
	User is talking with another person.

Chapter 6 describes in detail the use of speech recognition through the Java Speech API. Ways of improving recognition accuracy and reliability are discussed further. Chapter 3 looks at how developers should account for possible recognition errors in application design to make the user interface more robust and predictable.

CHAPTER 3

Designing Effective Speech Applications

Speech applications are like conversations between the user and the computer. Conversations are characterized by turn-taking, shifts in initiative, and verbal and non-verbal feedback to indicate understanding.

A major benefit of incorporating speech in an application is that speech is natural: people find speaking easy, conversation is a skill most master early in life and then practice frequently. At a deeper level, naturalness refers to the many subtle ways people cooperate with one another to ensure successful communication.

An effective speech application is one that simulates some of these core aspects of human-human conversation. Since language use is deeply ingrained in human behavior, successful speech interfaces should be based on an understanding of the different ways that people use language to communicate. Speech applications should adopt language conventions that help people know what they should say next and that avoid conversational patterns that violate standards of polite, cooperative behavior.

This chapter discusses when a speech interface is and is not appropriate, and then provides some concrete design ideas for creating effective speech applications that adhere to conversational conventions.

3.1 When to Use Speech

A crucial factor in determining the success of a speech application is whether or not there is a clear benefit to using speech. Since speech is such a natural medium for communication, users' expectations of a speech application tend to be extremely high. This means speech is best used when the need is clear — for example, when the user's hands and eyes are busy — or when speech enables something that cannot otherwise be done, such as accessing electronic mail or an on-line calendar over the telephone.

Speech applications are most successful when users are motivated to cooperate. For example, telephone companies have successfully used speech recognition to automate collect calls. People making a collect call want their call to go through, so they answer prompts carefully. People accepting collect calls are also motivated to cooperate, since they do not want to pay for unwanted calls or miss important calls from their friends and family. Automated collect calling systems save the company money and benefit users. Telephone companies report that callers prefer talking to the computer because they are sometimes embarrassed by their need to call collect and they feel that the computer makes the transaction more private.

Speech is well suited to some tasks, but not for others. The following tables list characteristics that can help you determine when speech input and output are appropriate choices.

Table 3-1 When is speech input appropriate?

Use When...	**Avoid When...**
• No keyboard is available (e.g., over the telephone, at a kiosk, or on a portable device). • Task requires the user's hands to be occupied so they cannot use a keyboard or mouse (e.g., maintenance and repair, graphics editing). • Commands are embedded in a deep menu structure. • Users are unable to type or are not comfortable with typing. • Users have a physical disability (e.g., limited use of hands).	• Task requires users to talk to other people while using the application. • Users work in a very noisy environment. • Task can be accomplished more easily using a mouse and keyboard.

Table 3-2 When is speech output appropriate?

Use When...	Avoid When...
• Task requires the user's eyes to be looking at something other than the screen (e.g., driving, maintenance and repair). • Situation requires grabbing users' attention. • Users have a physical disability (e.g., visual impairment). • Interface is trying to embody a personality.	• Large quantities of information must be presented. • Task requires user to compare data items. • Information is personal or confidential.

Including speech in an application because it is a novelty means it probably will not get used. Including it because there is some compelling reason increases the likelihood for success.

3.2 Design for Speech

After you determine that speech is an appropriate interface technique, consider how speech will be integrated into the application. Generally, a successful speech application is designed with speech in mind. It is rarely effective to add speech to an existing graphical application or to translate a graphical application directly into a speech-only one. Doing so is akin to translating a command-line-driven program directly into a graphical user interface. The program may work, but the most effective graphical programs are designed with the graphical environment in mind from the outset.

Graphical applications do not translate well into speech for several reasons. First, graphical applications do not always reflect the vocabulary, or even the basic concepts, that people use when *talking* to one another in the domain of the application. Consider a calendar application, for example. Most graphical calendar programs use an explicit visual representation of days, months, and years. There is no concept of relative dates (e.g., "the day after Labor Day" or "a week from tomorrow") built into the interface. When people speak to each other about scheduling, however, they make extensive use of relative dates. A speech interface to a calendar, whether speech-only or multi-modal, is therefore more likely to be effective if it allows users to speak about dates in both relative and absolute terms. By basing the speech interface design exactly on the graphical interface design, relative dates would not be included in the design, and the usability of the calendar application would be compromised.

Information organization is another important consideration. Presentations that work well in the graphical environment can fail completely in the speech environment. Reading exactly what is displayed on the screen is rarely effective. Likewise, users find it awkward to speak exactly what is printed on the display.

Consider the way in which many e-mail applications present message headers. An inbox usually consists of a chronological, sometimes numbered, list of headers containing information such as sender, subject, date, time, and size:

Table 3-3 Email message information

Sender	Subject	Date & Time	Size
`Arlene Rexford`	`Learn about Java`	`Mon Oct 28 11:23`	`2K`
`Shari Jackson`	`Re: Boston rumors`	`Fri Jul 18 09:32`	`3K`
`Hilary Binda`	`Change of address`	`Wed Jul 16 12:59`	`1K`
`Arlene Rexford`	`Class Openings`	`Tue Jul 21 12:35`	`8K`
`George Fitz`	`Re: Boston rumors`	`Tue Jul 21 12:46`	`1K`

You can scan this list and find a subject of interest or identify a message from a particular person. Imagine if someone read this information out loud to you, exactly as printed. It would take a long time! And the day, date, time, and size information, which you can easily ignore in the graphical representation, becomes quite prominent. It doesn't sound very natural, either. By the time you hear the fifth header, you may also have forgotten that there was an earlier message with the same subject.

An effective speech interface for an e-mail application would probably not read the date, time, and size information from the message header unless the user requests it. Better still would be an alternate organization scheme which groups messages into categories, perhaps by subject or sender (e.g., "You have two messages about 'Boston rumors'" or "You have two messages from Arlene Rexford"), so that the header list contains fewer individual items. Reading the items in a more natural spoken form would also be helpful. For example, instead of "Three. Hilary Binda. Change of address." the system might say "Message 3 from Hilary Binda is about Change of address."

On the speech input side, users find speaking menu commands is often awkward and unnatural. In one e-mail program, a menu called "Move" contains a list of mail box names. Translating this interface to speech would force the user to say something like "Move. Weekly Reports." A more natural interface would allow the user to say "File this in my Weekly Reports folder." The natural version is a little longer, but it is probably something the user could remember to say without looking at the screen.

3.3 Challenges

Even if you design an application with speech in mind from the outset, you face substantial challenges before your application is robust and easy to use. Understanding these challenges and assessing the various trade-offs that must be made during the design process will help to produce the most effective interface.

3.3.1 Transience: What did you say?

Speech is *transient*. Once you hear it or say it, it's gone. By contrast, graphics are *persistent*. A graphical interface typically stays on the screen until the user performs some action.

Listening to speech taxes users' short-term memory. Because speech is transient, users can remember only a limited number of items in a list and they may forget important information provided at the beginning of a long sentence. Likewise, while speaking to a dictation system, users often forget the exact words they have just spoken.

Users' limited ability to remember transient information has substantial implications for the speech interface design. In general, transience means that speech is not a good medium for delivering large amounts of information.

The transient nature of speech can also provide benefits. Because people can look and listen at the same time, speech is ideal for grabbing attention or for providing an alternate mechanism for feedback. Imagine receiving a notification about the arrival of an e-mail message while working on a spreadsheet. Speech might give the user the opportunity to ask for the sender or the subject of the message. The information can be delivered without forcing the user to switch contexts.

3.3.2 Invisibility: What can I say?

Speech is *invisible*. The lack of visibility makes it challenging to communicate the functional boundaries of an application to the user. In a graphical application, menus and other screen elements make most or all of the functionality of an application visible to a user. By contrast, in a speech application it is much more difficult to indicate to the user what actions they may perform, and what words and phrases they must say to perform those actions.

3.3.3 Asymmetry

Speech is *asymmetric*. People can produce speech easily and quickly, but they cannot listen nearly as easily and quickly. This asymmetry means people can speak faster than they can type, but listen much more slowly than they can read.

The asymmetry has design implications for what information to speak and how much to speak. A speech interface designer must balance the need to convey lots of instructions to users with users' limited ability to absorb spoken information.

3.3.4 Speech synthesis quality

Given that today's synthesizers still do not sound entirely natural, the choice to use synthesized output, recorded output, or no speech output is often a difficult one. Although recorded speech is much easier and more pleasant for users to listen to, it is difficult to use when the information being presented is dynamic. For example, recorded speech could not be used to read people their e-mail messages over the telephone. Using recorded speech is best for prompts that don't change, with synthesized speech being used for dynamic text.

Mixing recorded and synthesized speech, however, is not generally a good idea. Although users report not liking the sound of synthesized speech, they are, in fact, able to adapt to the synthesizer better when it is not mixed with recorded speech. Listening is considerably easier when the voice is consistent.

As a rule of thumb, use recorded speech when all the text to be spoken is known in advance, or when it is important to convey a particular personality to the user. Use synthesized speech when the text to be spoken is not known in advance, or when storage space is limited. Recorded audio requires substantially more disk space than synthesized speech.

3.3.5 Speech recognition performance

Speech recognizers are not perfect listeners. They make mistakes. A big challenge in designing speech applications, therefore, is working with imperfect speech recognition technology. While this technology improves constantly, it is unlikely that, in the foreseeable future, it will approach the robustness of computers in science fiction movies.

An application designer should understand the types of errors that speech recognizers make and the common causes of these errors. Refer to Table 2-1 in the previous chapter for a list of common errors and their causes.

Unfortunately, recognition errors cause the user to form an incorrect model of how the system works. For example, if the user says “Read the next message,” and the recognizer hears “Repeat the message,” the application will repeat the current message, leading the user to believe that “Read the next message” is not a valid way to ask for the next message. If the user then says “Next,” and the recognizer returns a rejection error, the user now eliminates “Next” as a valid option for moving forward. Unless there is a display that lists all the valid commands, users cannot know if the words they have spoken should work; therefore, if they don’t work, users assume they are invalid.

Some recognition systems adapt to users over time, but good recognition performance still requires cooperative users who are willing and able to adapt their speaking patterns to the needs of the recognition system. This is why providing users with a clear motivation to make speech work for them is essential.

3.3.6 Recognition: flexibility vs. accuracy

A flexible system allows users to speak the same commands in many different ways. The more flexibility an application provides for user input, the more likely errors are to occur. In designing a command-and-control style interface, therefore, the application designer must find a balance between flexibility and recognition accuracy. For example, a calendar application may allow the user to ask about tomorrow’s appointments in ways such as:

- What about tomorrow?
- What do I have tomorrow?
- What’s on my calendar for tomorrow?
- Read me tomorrow’s schedule.
- Tell me about the appointments I have on my calendar tomorrow.

This may be quite natural in theory, but, if recognition performance is poor, users will not accept the application. On the other hand, applications that provide a small, fixed set of commands also may not be accepted, even if the command phrases are designed to sound natural (e.g., Lookup tomorrow). Users tend to forget the exact wording of fixed commands. What seems natural for one user may feel awkward for another. Section 3.6, “Involving Users,” describes a technique for collecting data from users in order to determine the most common ways that people talk about a subject. In this way, applications can offer some flexibility without causing recognition performance to degrade dramatically.

3.4 Design Issues for Speech-Only Applications

A speech-only system is one in which speech input and output are the only options available to the user. Most speech-only systems operate over the telephone.

3.4.1 Feedback & Latency

In conversations, timing is critical. People read meaning into pauses. Unfortunately, processing delays in speech applications often cause pauses in places where they do not naturally belong. For example, users may reply to a prompt and then not hear an immediate response. This leads them to believe that they were not heard, so they speak again. This results in either missing the application's response when it does come (because the user is speaking at the same time) or causing a recognition error.

Giving users adequate feedback is especially important in speech-only interfaces. Processing delays, coupled with the lack of peripheral cues to help the user determine the state of the application, make consistent feedback a key factor in achieving user satisfaction.

When designing feedback, recall that speech is a slow output channel. This speed issue must be balanced with a user's need to know several vital facts:

- ♦ Is the recognizer processing or waiting for input?
- ♦ Has the recognizer heard the user's speech?
- ♦ If heard, was the user's speech correctly interpreted?

Verification should be commensurate with the cost of performing an action. *Implicitly verify* commands that present data and *explicitly verify* commands that destroy data or trigger actions. For example, it would be important to give the user plenty of feedback before authorizing a large payment, while it would not be as vital to ensure that a date is correct before checking a weather forecast. In the case of the payment, the feedback should be explicit (e.g., "Do you want to make a payment of $1,000 to Boston Electric? Say yes or no."), The feedback for the forecast query can be implicit (e.g., "Tomorrow's weather forecast for Boston is...."). In this case, the word "Tomorrow" serves as feedback that the date was correctly (or incorrectly) recognized. If correct, the interaction moves forward with minimal wasted time.

3.4.2 Prompting

Well designed prompts lead users smoothly through a successful interaction with a speech-only application. Many factors must be considered when designing prompts, but the most important is assessing the trade-off between flexibility and performance. The more you constrain what the user can say to an application, the less likely they are to encounter recognition errors. On the other hand, allowing users to enter information flexibly can often speed the interaction (if recognition succeeds), feel more natural, and avoid forcing users to memorize commands. Here are some tips for creating useful prompts.

- Use *explicit prompts* when the user input must be tightly constrained. For example, after recording a message, the prompt might be "Say cancel, send, or review." This sort of prompt directs the user to say just one of those three keywords.
- Use *implicit prompts* when the application is able to accept more flexible input. These prompts rely on conversational conventions to constrain the user input. For example, if the user says "Send mail to Bill," and "Bill" is ambiguous, the system prompt might be "Did you mean Bill Smith or Bill Jones?" Users are likely to respond with input such as "Smith" or "I meant Bill Jones." While possible, conversational convention makes it less likely that they would say "Bill Jones is the one I want."
- When possible, *taper* prompts to make them shorter. Tapering can be accomplished in one of two ways. If an application is presenting a set of data such as current quotes for a stock portfolio, drop out unnecessary words once a pattern is established. For example:

"As of 15 minutes ago,	*Sun Microsystems was*	*trading at 45 up 1/2,*
	Motorola was	*at 83 up 1/8, and*
	IBM was	*at 106 down 1/4"*

  Tapering can also happen over time. That is, if you need to tell the user the same information more than once, make it shorter each time. For example, you may wish to remind users about the correct way to record a message. The first time they record a message in a session, the instructions might be lengthy. The next time shorter and the third time just a quick reminder. For example:

  "Start recording after the tone. Pause for several seconds when done."
  "Record after the tone, then pause."
  "Record then pause."

- Use *incremental prompts* to speed interaction for expert users and provide help for less experienced users. This technique involves starting with a short prompt. If the user does not respond within a time-out period, the application prompts again with more detailed instructions. For example, the initial prompt might be: "Which service?" If the user says nothing, then the prompt could be expanded to: "Say banking, address book, or yellow pages."

3.4.3 Handling Errors

How a system handles recognition errors can dramatically affect the quality of a user's experience. If either the application or the user detects an error, an effective speech user interface should provide one or more mechanisms for correcting the error. While this seems obvious, correcting a speech input error is not always easy! If the user speaks a word or phrase again, the same error is likely to reoccur.

Techniques for handling rejection errors are somewhat different than those for handling misrecognitions and misfires. Perhaps the most important advice when handling rejection errors is not to repeat the same error message if the user experiences more than one rejection error in a row. Users find repetition to be hostile. Instead, try to provide *progressive assistance*. The first message might simply be "What?" If another error occurs, then perhaps, "Sorry. Please rephrase" will get the user to say something different. A third message might provide a tip on how to speak, "Still no luck. Speak clearly, but don't overemphasize."

Another technique is to reprompt with a more explicit prompt (such as a yes/no question) and switch to a more constrained grammar. If possible, provide an alternate input modality. For example, prompt the user to press a key on the telephone pad as an alternative to speaking.

As mentioned above, misrecognitions and misfires are harder to detect, and therefore harder to handle. One good strategy is to *filter recognition results* for unlikely user input. For example, a scheduling application might assume that an error has occurred if the user appears to want to schedule a meeting for 3am.

Flexible correction mechanisms that allow a user to correct a portion of the input are helpful. For example, if the user asks for a weather forecast for Boston for Tuesday, the system might respond "Tomorrow's weather for Boston is..." A flexible correction mechanism would allow the user to just correct the day: "No, I said Tuesday."

3.5 Design Issues for Multi-Modal Applications

Multi-modal applications include other input and output modalities along with speech. For example, speech integrated with a desktop application would be multi-modal, as would speech augmenting the controls of a personal note taker or a radio. While many of the design issues for a multi-modal application are the same as for a speech-only one, some specific issues are unique to applications that provide users with multiple input mechanisms, particularly graphical interfaces driven by keyboard and mouse.

3.5.1 Feedback & Latency

As in speech-only systems, performance delays can cause confusion for users. Fortunately, a graphic display can show the user the state of the recognizer (processing or waiting for input) which a speech-only interface cannot. If a screen is available, displaying the results of the recognizer makes it obvious if the recognizer has heard and if the results were accurate.

As mentioned earlier, the transient nature of speech sometimes causes people to forget what they just said. When dictating, particularly when dictating large amounts of text, this problem is compounded by recognition errors. When a user looks at dictated text and sees it is different from what they recall saying, making a correction is not always easy since they will not necessarily remember what they said or even what they were thinking. Access to a recording of the original speech is extremely helpful in aiding users in the correction of dictated text.

The decision of whether or not to show unfinalized results is a problem in continuous dictation applications. Unfinalized results are words that the recognizer is hypothesizing that the user has said, but for which it has not yet committed a decision. As the user says more, these words may change. Unfinalized text can be hidden from the user, displayed in the text stream in reverse video (or some other highlighted fashion), or shown in a separate window. Eventually, the recognizer makes its best guess and finalizes the words. An application designer makes a trade-off between showing users words that may change and having a delay before the recognizer is able to provide the finalized results. Showing the unfinalized results can be confusing, but not showing any words can lead the user to believe that the system has not heard them.

3.5.2 Prompting

Prompts in multi-modal systems can be spoken or printed. Deciding on an appropriate strategy depends greatly on the content and context of the application. If privacy is an issue, it is probably better not to have the computer speak out loud.

On the other hand, even a little bit of spoken output can enable eyes-free interaction and can provide the user with the sense of having a conversational partner rather than speaking to an inanimate object.

With a screen available, explicit prompts usually involve providing the user with a list of valid spoken commands. These lists can become cumbersome unless they are organized hierarchically.

Another strategy is to let users speak any text they see on the screen, whether it is menu text or button text or field names. In applications that support more than simple spoken commands, one strategy is to list examples of what the user can say next, rather than a complete laundry list of every possible utterance.

3.5.3 Handling Errors

Multi-modal speech systems that display recognition results make it easier for users to detect errors. If a rejection error occurs, no text will appear in the area where recognition results are displayed. If the recognizer makes a misrecognition or misfire error, the user can see what the recognizer thinks was said and correct any errors.

Even with feedback displayed, an application should not assume that users will always catch errors. Filtering for unexpected input is still helpful, as is allowing the user to switch to a different input modality if recognition is not working reliably.

3.6 Involving Users

Involving users in the design process throughout the lifecycle of a speech application is crucial. A natural, effective interface can only be achieved by understanding how and where and why target users will interact with the application.

3.6.1 Natural Dialog Studies

At the very early stages of design, users can help to define application functionality and, critical to speech interface design, provide input on how humans carry out conversations in the domain of the application. This information can be collected by performing a *natural dialog study,* which involves asking target users to talk with each other while working through a scenario. For example, if you are designing a telephone-based e-mail program, you might work with pairs of study participants. Put the participants in two separate rooms. Give one participant a telephone and a computer with an e-mail program. Give the other only a telephone. Have the participant with only the telephone call the participant with the computer and ask to have his or her mail read aloud. Leave the task open ended, but add a few guidelines such as “be sure to answer all messages that require a response.”

In some natural dialog studies it is advantageous to include a subject matter expert. For example, if you wish to automate a telephone-based financial service, study participants might call up and speak with an expert customer service representative from the financial service company.

Natural dialog studies are an effective technique for collecting vocabulary, establishing commonly used grammatical patterns, and providing ideas for prompt and feedback design. When a subject matter expert is involved, prompt and feedback design can be based on phrases and responses the expert uses when speaking with customers.

In general, natural dialog studies are quick and inexpensive. It is not necessary to include large numbers of participants.

3.6.2 Wizard-of-Oz Studies

Once a preliminary application design is complete, but before the speech application is implemented, a *wizard-of-oz study* can help test and refine the interface. In these studies, a human *wizard* — usually using software tools — simulates the speech interface. Major usability problems are often uncovered with these types of simulations. (The term “Wizard of Oz” comes from the classic movie in which the wizard controls an impressive display while hidden behind a curtain.)

Continuing the e-mail example, a wizard-of-oz study might involve bringing in study participants and telling them that the computer is going to read them their e-mail. When they call a telephone number, the human wizard answers, but manipulates the computer so that a synthesized voice speaks to the participant. As the participant asks to navigate through the mailbox, hear messages, or reply to messages, the wizard carries out the operations and has the computer speak the responses.

Since computer tools are usually necessary to carry out a convincing simulation, wizard-of-oz studies are more time-consuming and complicated to run than natural dialog studies. If a prototype of the final application can be built quickly, it may be more cost-effective to move directly to a usability study.

3.6.3 Usability Studies

A *usability study* assesses how well users are able to carry out the primary tasks that an application is designed to support. Conducting such a study requires at least a preliminary software implementation. The application need not be complete, but some of the core functionality must be working. Usability studies can be conducted either in a laboratory or in the field. Study participants are typically presented with one or more tasks that they must figure out how to accomplish using the application.

With speech applications, usability studies are particularly important for uncovering problems due to recognition errors, which are difficult to simulate effectively in a wizard-of-oz study, but are a leading cause of usability problems. The effectiveness of an application's error recovery functionality must be tested in the environments in which real users will use the application.

Conducting usability tests of speech applications can be a bit tricky. Two standard techniques used in tests of graphical applications -- facilitated discussions and speak-aloud protocols -- cannot be used effectively for speech applications. A facilitated discussion involves having a facilitator in the room with the study participant. Any human-human conversation, however, can interfere with the human-computer conversation, causing recognition errors. Speak-aloud protocols involve asking the study participant to verbalize their thoughts as they work with the software. Obviously this is not desirable when dealing with a speech recognizer. It is best, therefore, to have study participants work in isolation, speaking only into a telephone or microphone. A tester should not intervene unless the participant becomes completely stuck. A follow-up interview can be used to collect the participant's comments and reactions.

3.7 Summary

An effective speech application is one that uses speech to enhance a user's performance of a task or enable an activity that cannot be done without it. Designing an application with speech in mind from the outset is a key success factor. Basing the dialog design on a natural dialog study ensures that the input grammar will match the phrasing actually used by people when speaking in the domain of the application. A natural dialog study also assures that prompts and feedback follow conversational conventions that users expect in a cooperative interaction. Once an application is designed, wizard-of-oz and usability studies provide opportunities to test interaction techniques and refine application behavior based on feedback from prototypical users.

3.8 For More Information

The following sources provide additional information on speech user interface design.

- Fraser, N.M. and G.N. Gilbert, "Simulating Speech Systems," Computer Speech and Language, Vol. 5, Academic Press Limited, 1991.
- Raman, T.V. *Auditory User Interfaces: Towards the Speaking Computer.* Kluwer Academic Publishers, Boston, MA, 1997.
- Roe, D.B. and N.M. Wilpon, editors. *Voice Communication Between Humans and Machines*. National Academy Press, Washington D.C., 1994.
- Schmandt, C. *Voice Communication with Computers: Conversational Systems*. Van Nostrand Reinhold, New York, 1994.
- Yankelovich, N, G.A. Levow, and M. Marx, *"Designing SpeechActs: Issues in Speech User Interfaces,"* CHI '95 Conference on Human Factors in Computing Systems, Denver, CO, May 7-11, 1995.

CHAPTER 4

Speech Engines: javax.speech

This chapter introduces the `javax.speech` package.This package defines the behavior of all speech engines (speech recognizers and synthesizers). The topics covered include:

- ♦ What is a Speech Engine?
- ♦ Properties of a Speech Engine
- ♦ Locating, Selecting and Creating Engines
- ♦ Engine States
- ♦ Speech Events
- ♦ Other Engine Functions

4.1 What is a Speech Engine?

The `javax.speech` package of the Java Speech API defines an abstract software representation of a *speech engine*. "Speech engine" is the generic term for a system designed to deal with either speech input or speech output. Speech synthesizers and speech recognizers are both speech engine instances. Speaker verification systems and speaker identification systems are also speech engines but are not currently supported through the Java Speech API.

The `javax.speech` package defines classes and interfaces that define the basic functionality of an engine. The `javax.speech.synthesis` package and `javax.speech.recognition` package extend and augment 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 JSAPI engine: that it provides a true implementation of the Java classes and interfaces defined by the API. In supporting those classes and interfaces, an engine may completely software-based or may be a combination of software and hardware. The engine may be local to the client computer or remotely operating on a server. The engine may be written entirely as Java software 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.

1. Identify the application's functional requirements for an engine (e.g, language or dictation capability).
2. Locate and create an engine that meets those functional requirements.
3. Allocate the resources for the engine.
4. Set up the engine.
5. Begin operation of the engine - technically, resume it.
6. Use the engine
7. Deallocate the resources of the engine.

Steps 4 and 6 in this process operate differently for the two types of speech engine - recognizer or synthesizer. The other steps apply to all speech engines and are described in the remainder of this chapter.

The "Hello World!" code example for speech synthesis (see page 58) and the "Hello World!" code example for speech recognition (see page 72) both illustrate the 7 steps described above. They also show that simple speech applications are simple to write with the Java Speech API - writing your first speech application should not be too hard.

4.2 Properties of a Speech Engine

Applications are responsible for determining their functional requirements for a speech synthesizer and/or speech recognizer. For example, an application might determine that it needs a dictation recognizer for the local language or a speech synthesizer for Korean with a female voice. Applications are also responsible for determining behavior when there is no speech engine available with the required features. Based on specific functional requirements, a speech engine can be selected, created, and started. This section explains how the features of a speech engine are used in engine selection, and how those features are handled in Java software.

Functional requirements are handled in applications as *engine selection properties*. Each installed speech synthesizer and speech recognizer is defined by a set of properties. An installed engine may have one or many *modes of operation*, each defined by a unique set of properties, and encapsulated in a *mode descriptor* object.

The basic engine properties are defined in the `EngineModeDesc` class. Additional specific properties for speech recognizers and synthesizers are defined by the `RecognizerModeDesc` and `SynthesizerModeDesc` classes that are contained in the `javax.speech.recognition` and `javax.speech.synthesis` packages respectively.

In addition to *mode descriptor* objects provided by speech engines to describe their capabilities, an application can create its own mode descriptor objects to indicate its functional requirements. The same Java classes are used for both purposes. An engine-provided mode descriptor describes an actual mode of operation whereas an application-defined mode descriptor defines a preferred or desired mode of operation. (*Locating, Selecting and Creating Engines* on page 39 describes the use of a mode descriptor.)

The basic properties defined for all speech engines are listed in Table 4-1.

Table 4-1 Basic engine selection properties: EngineModeDesc

Property Name	**Description**
`EngineName`	A `String` that defines the name of the speech engine. e.g., “Acme Dictation System”.
`ModeName`	A `String` that defines a specific mode of operation of the speech engine. e.g. “Acme Spanish Dictator”.
`Locale`	A `java.util.Locale` object that indicates the language supported by the speech engine, and optionally, a country and a variant. The `Locale` class uses standard ISO 639 language codes and ISO 3166 country codes. For example, `Locale("fr", "ca")` represents a Canadian French locale, and `Locale("en", "")` represents English (the language).

Table 4-1 Basic engine selection properties: EngineModeDesc

Property Name	Description
`Running`	A `Boolean` object that is `TRUE` for engines which are already running on a platform, otherwise `FALSE`. Selecting a running engine allows for sharing of resources and may also allow for fast creation of a speech engine object.

The one additional property defined by the `SynthesizerModeDesc` class for speech synthesizers is shown in Table 4-2.

Table 4-2 Synthesizer selection properties: SynthesizerModeDesc

Property Name	Description
`List of voices`	An array of voices that the synthesizer is capable of producing. Each voice is defined by an instance of the `Voice` class which encapsulates voice name, gender, age and speaking style.

The two additional properties defined by the `RecognizerModeDesc` class for speech recognizers are shown in Table 4-3.

Table 4-3 Recognizer selection properties: RecognizerModeDesc

Property Name	Description
`Dictation supported`	A `Boolean` value indicating whether this mode of operation of the recognizer supports a dictation grammar.
`Speaker profiles`	A list of `SpeakerProfile` objects for speakers who have trained the recognizer. Recognizers that do not support training return a `null` list.

All three mode descriptor classes, `EngineModeDesc`, `SynthesizerModeDesc` and `RecognizerModeDesc` use the get and set property patterns for JavaBeans™. For example, the `Locale` property has get and set methods of the form:

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

Furthermore, all the properties are defined by class objects, never by primitives (primitives in the Java programming language include `boolean`, `int` etc.). With this design, a `null` value always represents "don't care" and is used by applications to indicate that a particular property is unimportant to its functionality. For instance, a `null` value for the "dictation supported" property indicates that dictation is not relevant to engine selection. Since that property is represented by the `Boolean` class, a value of `TRUE` indicates that dictation is required and `FALSE` indicates explicitly that dictation should not be provided.

4.3 Locating, Selecting and Creating Engines

4.3.1 Default Engine Creation

The simplest way to create a speech engine is to request a default engine. This is appropriate when an application wants an engine for the default locale (specifically for the local language) and does not have any special functional requirements for the engine. The `Central` class in the `javax.speech` package is used for locating and creating engines. Default engine creation uses two static methods of the `Central` class.

```
Synthesizer Central.createSynthesizer(EngineModeDesc mode);
Recognizer Central.createRecognizer(EngineModeDesc mode);
```

The following code creates a default `Recognizer` and `Synthesizer`.

```
import javax.speech.*;
import javax.speech.synthesis.*;
import javax.speech.recognition.*;

{
   // Get a synthesizer for the default locale
   Synthesizer synth = Central.createSynthesizer(null);
   // Get a recognizer for the default locale
```

```
    Recognizer rec = Central.createRecognizer(null);
}
```

For both the `createSynthesizer` and `createRecognizer` the `null` parameters indicate that the application doesn't care about the properties of the synthesizer or recognizer. However, both creation methods have an implicit selection policy. Since the application did not specify the language of the engine, the language from the system's default locale returned by `java.util.Locale.getDefault()` is used. In all cases of creating a speech engine, the Java Speech API forces language to be considered since it is fundamental to correct engine operation.

If more than one engine supports the default language, the `Central` then gives preference to an engine that is running (running property is true), and then to an engine that supports the country defined in the default locale.

If the example above is performed in the US locale, a recognizer and synthesizer for the English language will be returned if one is available. Furthermore, if engines are installed for both British and US English, the US English engine would be created.

4.3.2 Simple Engine Creation

The next easiest way to create an engine is to create a mode descriptor, define desired engine properties and pass the descriptor to the appropriate engine creation method of the `Central` class. When the mode descriptor passed to the `createSynthesizer` or `createRecognizer` methods is non-null, an engine is created which matches all of the properties defined in the descriptor. If no suitable engine is available, the methods return `null`.

The list of properties is described in the *Properties of a Speech Engine* section on page 36. All the properties in `EngineModeDesc` and its sub-classes `RecognizerModeDesc` and `SynthesizerModeDesc` default to `null` to indicate "don't care".

The following code sample shows a method that creates a dictation-capable recognizer for the default locale. It returns `null` if no suitable engine is available.

```
/** Get a dictation recognizer for the default locale */
Recognizer createDictationRecognizer()
{
    // Create a mode descriptor with all required features
    RecognizerModeDesc required = new RecognizerModeDesc();
    required.setDictationGrammarSupported(Boolean.TRUE);
    return Central.createRecognizer(required);
}
```

Since the `required` object provided to the `createRecognizer` method does not have a specified locale (it is not set, so it is `null`) the `Central` class again enforces a policy of selecting an engine for the language specified in the system's default locale. The `Central` class will also give preference to running engines and then to engines that support the country defined in the default locale.

In the next example we create a `Synthesizer` for Spanish with a male voice.

```
/**
 * Return a speech synthesizer for Spanish.
 * Return null if no such engine is available.
 */
Synthesizer createSpanishSynthesizer()
{
   // Create a mode descriptor with all required features
   // "es" is the ISO 639 language code for "Spanish"
   SynthesizerModeDesc required = new SynthesizerModeDesc();
   required.setLocale(new Locale("es", null));
   required.addVoice(new Voice(
                 null, GENDER_MALE, AGE_DONT_CARE, null));
   return Central.createSynthesizer(required);
}
```

Again, the method returns `null` if no matching synthesizer is found and the application is responsible for determining how to handle the situation.

4.3.3 Advanced Engine Selection

This section explains more advanced mechanisms for locating and creating speech engines. Most applications do not need to use these mechanisms. Readers may choose to skip this section.

In addition to performing engine creation, the `Central` class can provide lists of available recognizers and synthesizers from two static methods.

```
EngineList availableSynthesizers(EngineModeDesc mode);
EngineList availableRecognizers(EngineModeDesc mode);
```

If the mode passed to either method is `null`, then all known speech recognizers or synthesizers are returned. Unlike the `createRecognizer` and `createSynthesizer` methods, there is no policy that restricts the list to the default locale or to running engines — in advanced selection such decisions are the responsibility of the application.

Both `availableSynthesizers` and `availableRecognizers` return an `EngineList` object, a sub-class of `Vector`. If there are no available engines, or no engines that match the properties defined in the mode descriptor, the list is zero length (not `null`) and its `isEmpty` method returns `true`. Otherwise the list contains a set of `SynthesizerModeDesc` or `RecognizerModeDesc` objects each defining a mode of operation of an engine. These mode descriptors are engine-defined so all their features are defined (non-null) and applications can test these features to refine the engine selection.

Because `EngineList` is a sub-class of `Vector`, each element it contains is a Java `Object`. Thus, when accessing the elements applications need to cast the objects to `EngineModeDesc`, `SynthesizerModeDesc` or `RecognizerModeDesc`.

The following code shows how an application can obtain a list of speech synthesizers with a female voice for German. All other parameters of the mode descriptor remain `null` for "don't care" (engine name, mode name etc.).

```
import javax.speech.*;
import javax.speech.synthesis.*;

// Define the set of required properties in a mode descriptor
SynthesizerModeDesc required = new SynthesizerModeDesc();
required.setLocale(new Locale("de", ""));
required.addVoice(new Voice(
             null, GENDER_FEMALE, AGE_DONT_CARE, null));

// Get the list of matching engine modes
EngineList list = Central.availableSynthesizers(required);

// Test whether the list is empty - any suitable synthesizers?
if (list.isEmpty()) ...
```

If the application specifically wanted Swiss German and a running engine it would add the following before calling `availableSynthesizers`:

```
required.setLocale(new Locale("de", "CH"));
required.setRunning(Boolean.TRUE);
```

To create a speech engine from a mode descriptor obtained through the `availableSynthesizers` and `availableRecognizers` methods, an application simply calls the `createSynthesizer` or `createRecognizer` method. Because the engine created the mode descriptor and because it provided values for all the properties, it has sufficient information to create the engine directly. An example later in this section illustrates the creation of a `Recognizer` from an engine-provided mode descriptor.

Although applications do not normally care, engine-provided mode descriptors are special in two other ways. First, all engine-provided mode descriptors are required to implement the `EngineCreate` interface which includes a single `createEngine` method. The `Central` class uses this interface to perform the creation. Second, engine-provided mode descriptors may extend the `SynthesizerModeDesc` and `RecognizerModeDesc` classes to encapsulate additional features and information. Applications should not access that information if they want to be portable, but engines will use that information when creating a running `Synthesizer` or `Recognizer`.

4.3.3.1 Refining an Engine List

If more than one engine matches the required properties provided to `availableSynthesizers` or `availableRecognizers` then the list will have more than one entry and the application must choose from amongst them.

In the simplest case, applications simply select the first in the list which is obtained using the `EngineList.first` method. For example:

```
EngineModeDesc required;
...
EngineList list = Central.availableRecognizers(required);

if (!list.isEmpty()) {
    EngineModeDesc desc = (EngineModeDesc)(list.first());
    Recognizer rec = Central.createRecognizer(desc);
}
```

More sophisticated selection algorithms may test additional properties of the available engine. For example, an application may give precedence to a synthesizer mode that has a voice called “Victoria”.

The list manipulation methods of the `EngineList` class are convenience methods for advanced engine selection.

- `anyMatch(EngineModeDesc)` returns true if at least one mode descriptor in the list has the required properties.
- `requireMatch(EngineModeDesc)` removes elements from the list that do not match the required properties.
- `rejectMatch(EngineModeDesc)` removes elements from the list that match the specified properties.
- `orderByMatch(EngineModeDesc)` moves list elements that match the properties to the head of the list.

The following code shows how to use these methods to obtain a Spanish dictation recognizer with preference given to a recognizer that has been trained for a specified speaker passed as an input parameter.

```
import javax.speech.*;
import javax.speech.recognition.*;
import java.util.Locale;

Recognizer getSpanishDictation(String name)
{
   RecognizerModeDesc required = new RecognizerModeDesc();
   required.setLocale(new Locale("es", ""));
   required.setDictationGrammarSupported(Boolean.TRUE);

   // Get a list of Spanish dictation recognizers
   EngineList list = Central.availableRecognizers(required);

   if (list.isEmpty()) return null; // nothing available

   // Create a description for an engine trained for the speaker
   SpeakerProfile profile = new SpeakerProfile(null, name, null);
   RecognizerModeDesc requireSpeaker = new RecognizerModeDesc();
   requireSpeaker.addSpeakerProfile(profile);

   // Prune list if any recognizers have been trained for speaker
   if (list.anyMatch(requireSpeaker))
       list.requireMatch(requireSpeaker);

   // Now try to create the recognizer
   RecognizerModeDesc first =
                (RecognizerModeDesc)(list.firstElement());
   try {
       return Central.createRecognizer(first);
   } catch (SpeechException e) {
       return null;
   }
}
```

4.4 Engine States

4.4.1 State systems

The `Engine` interface includes a set of methods that define a generalized state system manager. Here we consider the operation of those methods. In the following sections we consider the two core state systems implemented by all speech engines: the allocation state system and the pause-resume state system. In Chapter 5, the state system for synthesizer queue management is described. In Chapter 6, the state systems for recognizer focus and for recognition activity are described.

A state defines a particular mode of operation of a speech engine. For example, the output queue moves between the `QUEUE_EMPTY` and `QUEUE_NOT_EMPTY` states. The following are the basics of state management.

The `getEngineState` method of the `Engine` interface returns the current engine state. The engine state is represented by a `long` value (64-bit value). Specified bits of the state represent the engine being in specific states. This bit-wise representation is used because *an engine can be in more than one state at a time*, and usually is during normal operation.

Every speech engine must be in one and only one of the four allocation states (described in detail in Section 4.4.2). These states are `DEALLOCATED`, `ALLOCATED`, `ALLOCATING_RESOURCES` and `DEALLOCATING_RESOURCES`. The `ALLOCATED` state has multiple sub-states. Any `ALLOCATED` engine must be in either the `PAUSED` or the `RESUMED` state (described in detail in Section 4.4.4).

Synthesizers have a separate sub-state system for queue status. Like the paused/resumed state system, the `QUEUE_EMPTY` and `QUEUE_NOT_EMPTY` states are both sub-states of the `ALLOCATED` state. Furthermore, the queue status and the paused/resumed status are independent.

Recognizers have three independent sub-state systems to the `ALLOCATED` state (the `PAUSED`/`RESUMED` system plus two others). The `LISTENING`, `PROCESSING` and `SUSPENDED` states indicate the current activity of the recognition process. The `FOCUS_ON` and `FOCUS_OFF` states indicate whether the recognizer currently has speech focus. For a recognizer, all three sub-state systems of the `ALLOCATED` state operate independently (with some exceptions that are discussed in the recognition chapter).

Each of these state names is represented by a static long in which a single unique bit is set. The & and | operators of the Java programming language are used to manipulate these state bits. For example, the state of an allocated, resumed synthesizer with an empty speech output queue is defined by:

```
(Engine.ALLOCATED | Engine.RESUMED | Synthesizer.QUEUE_EMPTY)
```

To test whether an engine is resumed, we use the test:

```
if ((engine.getEngineState() & Engine.RESUMED) != 0) ...
```

For convenience, the `Engine` interface defines two additional methods for handling engine states. The `testEngineState` method is passed a state value and returns `true` if all the state bits in that value are currently set for the engine. Again, to test whether an engine is resumed, we use the test:

```
if (engine.testEngineState(Engine.RESUMED)) ...
```

Technically, the `testEngineState(state)` method is equivalent to:

```
if ((engine.getEngineState() & state) == state)...
```

The final state method is `waitEngineState`. This method blocks the calling thread until the engine reaches the defined state. For example, to wait until a synthesizer stops speaking because its queue is empty we use:

```
engine.waitEngineState(Synthesizer.QUEUE_EMPTY);
```

In addition to method calls, applications can monitor state through the event system. Every state transition is marked by an `EngineEvent` being issued to each `EngineListener` attached to the `Engine`. The `EngineEvent` class is extended by the `SynthesizerEvent` and `RecognizerEvent` classes for state transitions that are specific to those engines. For example, the `RECOGNIZER_PROCESSING` `RecognizerEvent` indicates a transition from the `LISTENING` state to the `PROCESSING` (which indicates that the recognizer has detected speech and is producing a result).

4.4.2 Allocation State System

Engine allocation is the process in which the resources required by a speech recognizer or synthesizer are obtained. Engines are not automatically allocated when created because speech engines can require substantial resources (CPU, memory and disk space) and because they may need exclusive access to an audio resource (e.g. microphone input or speaker output). Furthermore, allocation can be a slow procedure for some engines (perhaps a few seconds or over a minute).

The `allocate` method of the `Engine` interface requests the engine to perform allocation and is usually one of the first calls made to a created speech engine. A newly created engine is always in the `DEALLOCATED` state. A call to the `allocate` method is, technically speaking, a request to the engine to transition to the `ALLOCATED` state. During the transition, the engine is in a temporary `ALLOCATING_RESOURCES` state.

The `deallocate` method of the `Engine` interface requests the engine to perform deallocation of its resources. All well-behaved applications call `deallocate` once they have finished using an engine so that its resources are freed up for other applications. The `deallocate` method returns the engine to the `DEALLOCATED` state. During the transition, the engine is in a temporary `DEALLOCATING_RESOURCES` state.

Figure 4-1 shows the state diagram for the allocation state system.

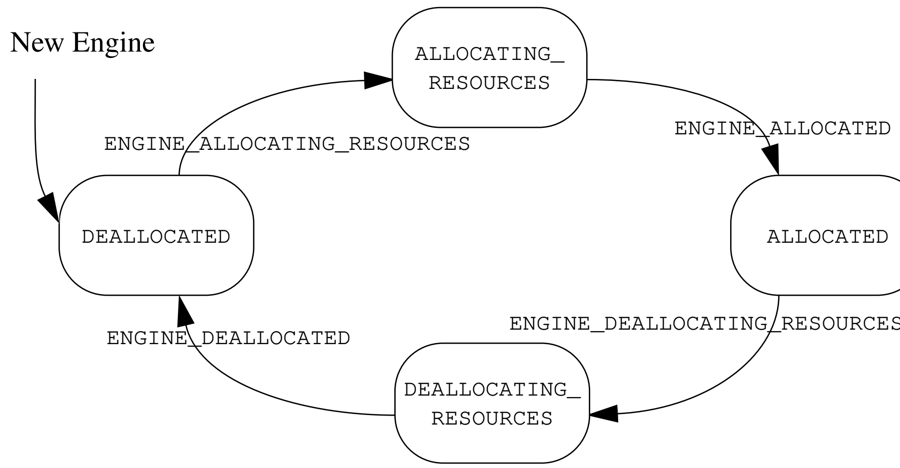

Figure 4-1 Engine allocation state system

Each block represents a state of the engine. An engine must always be in one of the four specified states. As the engine transitions between states, the event labelled on the transition arc is issued to the `EngineListeners` attached to the engine.

The normal operational state of an engine is `ALLOCATED`. The paused-resumed state of an engine is described in the next section. The sub-state systems of `ALLOCATED` synthesizers and recognizers are described in Chapter 5 and Chapter 6 respectively.

4.4.3 Allocated States and Call Blocking

For advanced applications, it is often desirable to start up the allocation of a speech engine in a background thread while other parts of the application are being initialized. This can be achieved by calling the `allocate` method in a separate thread. The following code shows an example of this using an inner class implementation of the `Runnable` interface. To determine when the allocation method is complete, we check later in the code for the engine being in the `ALLOCATED` state.

```
Engine engine;
{
   engine = Central.createRecognizer();

   new Thread(new Runnable() {
      public void run() {
         try {
            engine.allocate();
         }
         catch (Exception e) {
            e.printStackTrace();
         }
      }
   }).start();

   // Do other stuff while allocation takes place
   ...

   // Now wait until allocation is complete
   engine.waitEngineState(Engine.ALLOCATED);
   }
}
```

A full implementation of an application that uses this approach to engine allocation needs to consider the possibility that the allocation fails. In that case, the allocate method throws an `EngineException` and the engine returns to the `DEALLOCATED` state.

Another issue advanced applications need to consider is class blocking. Most methods of the `Engine`, `Recognizer` and `Synthesizer` are defined for normal operation in the `ALLOCATED` state. What if they are called for an engine in another allocation state? For most methods, the operation is defined as follows:

- `ALLOCATED` state: for nearly all methods normal behavior is defined for this state. (An exception is the `allocate` method).
- `ALLOCATING_RESOURCES` state: most methods *block* in this state. The calling thread waits until the engine reaches the `ALLOCATED` state. Once that state is reached, the method behaves as normally defined.
- `DEALLOCATED` state: most methods are not defined for this state, so an `EngineStateError` is thrown. (Exceptions include the `allocate` method and certain methods listed below.)
- `DEALLOCATING_RESOURCES` state: most methods are not defined for this state, so an `EngineStateError` is thrown.

A small subset of engine methods will operate correctly in all engine states. The `getEngineProperties` always allows runtime engine properties to be set and tested (although properties only take effect in the `ALLOCATED` state). The `getEngineModeDesc` method can always return the mode descriptor for the engine. Finally, the three engine state methods — `getEngineState`, `testEngineState` and `waitEngineState` — always operated as defined.

4.4.4 Pause - Resume State System

All `ALLOCATED` speech engines have `PAUSED` and `RESUMED` states. Once an engine reaches the `ALLOCATED` state, it enters either the `PAUSED` or the `RESUMED` state. The factors that affect the initial `PAUSED`/`RESUMED` state are described below.

The `PAUSED`/`RESUMED` state indicates whether the audio input or output of the engine is on or off. A resumed recognizer is receiving audio input. A paused recognizer is ignoring audio input. A resumed synthesizer produces audio output as it speaks. A paused synthesizer is not producing audio output.

As part of the engine state system, the Engine interface provides several methods to test `PAUSED`/`RESUMED` state. The general state system is described previously in Section 4.4 (on page 44).

An application controls an engine's `PAUSED`/`RESUMED` state with the `pause` and `resume` methods. An application may pause or resume an engine indefinitely. Each time the `PAUSED`/`RESUMED` state changes an `ENGINE_PAUSED` or `ENGINE_RESUMED` type of `EngineEvent` is issued each `EngineListener` attached to the `Engine`.

Figure 4-2 shows the basic pause and resume diagram for a speech engine. As a sub-state system of the `ALLOCATED` state, the pause and resume states represented within the `ALLOCATED` state as shown in Figure 4-1.

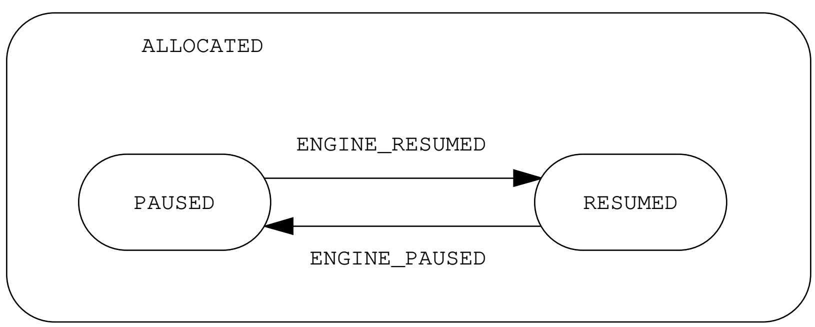

Figure 4-2 PAUSED and RESUMED Engine states

As with Figure 4-1, Figure 4-2 represents states as labelled blocks, and the engine events as labelled arcs between those blocks. In this diagram the large block is the `ALLOCATED` state which contains both the `PAUSED` and `RESUMED` states.

4.4.5 State Sharing

The `PAUSED`/`RESUMED` state of a speech engine may, in many situations, be shared by multiple applications. Here we must make a distinction between the Java object that represents a `Recognizer` or `Synthesizer` and the underlying engine that may have multiple Java and non-Java applications connected to it. For example, in personal computing systems (e.g., desktops and laptops), there is typically a single engine running and connected to microphone input or speaker/headphone output and all application share that resource.

When a `Recognizer` or `Synthesizer` (the Java software object) is paused and resumed the shared underlying engine is paused and resumed and all applications connected to that engine are affected.

There are three key implications from this architecture:

- An application should pause and resume an engine only in response to a user request (e.g., because a microphone button is pressed for a recognizer). For example, it should not pause an engine before deallocating it.
- A `Recognizer` or `Synthesizer` may be paused and resumed because of a request by another application. The application will receive an `ENGINE_PAUSED` or `ENGINE_RESUMED` event and the engine state value is updated to reflect the current engine state.
- Because an engine could be resumed without explicitly requesting a resume it should always be prepared for that resume. For example, it should not place text on the synthesizer's output queue unless it would expect it to be spoken upon a resume. Similarly, the set of enabled grammars of a recognizer should always be appropriate to the application context, and the application should be prepared to accept input results from the recognizer if an enabled grammar is unexpectedly resumed.

4.4.6 Synthesizer Pause

For a speech synthesizer — a speech output device — pause immediately stops the audio output of synthesized speech. Resume recommences speech output from the point at which the pause took effect. This is analogous to pause and resume on a tape player or CD player.

Chapter 5 describes an additional state system of synthesizers. An ALLOCATED Synthesizer has sub-states for QUEUE_EMPTY and QUEUE_NOT_EMPTY. This represents whether there is text on the speech output queue of the synthesizer that is being spoken or waiting to be spoken. The queue state and pause/resume state are independent. It is possible, for example, for a RESUMED synthesizer to have an empty output queue (QUEUE_EMPTY state). In this case, the synthesizer is silent because it has nothing to say. If any text is provided to be spoken, speech output will start immediately because the synthesizer is RESUMED.

4.4.7 Recognizer Pause

For a recognizer, pausing and resuming turns audio input off and on and is analogous to switching the microphone off and on. When audio input is off the audio is lost. Unlike a synthesizer, for which a resume continues speech output from the point at which it was paused, resuming a recognizer restarts the processing of audio input from the time at which resume is called.

Under normal circumstances, pausing a recognizer will stop the recognizer's internal processes that match audio against grammars. If the user was in the middle of speaking at the instant at which the recognizer was paused, the recognizer is forced to finalize its recognition process. This is because a recognizer cannot assume that the audio received just before pausing is in any way linked to the audio data that it will receive after being resumed. Technically speaking, pausing introduces a discontinuity into the audio input stream.

One complexity for pausing and resuming a recognizer (not relevant to synthesizers) is the role of internal buffering. For various reasons, described in Chapter 6, 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 to the grammars. If recognizer is performing in real-time the buffer is empty or nearly empty. 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; i.e, the recognizer stops putting data into the buffer. At the other end of the buffer — where the actual recognition is performed —the recognizer continues to process audio data until the buffer is empty. This means that the recognizer can continue to produce recognition results for a limited period of time even after it has been paused. (A Recognizer also provides a forceFinalize method with an option to flush the audio input buffer.)

Chapter 6 describes an additional state system of recognizers. An ALLOCATED Recognizer has a separate sub-state system for LISTENING, RECOGNIZING and SUSPENDED. These states indicate the current activity of the internal recognition process. These states are largely decoupled from the `PAUSED` and `RESUMED` states except that, as described in detail in Chapter 6, a paused recognizer eventually returns to the `LISTENING` state when it runs out of audio input (the `LISTENING` state indicates that the recognizer is listening to background silence, not to speech).

The `SUSPENDED` state of a `Recognizer` is superficially similar to the `PAUSED` state. In the `SUSPENDED` state the recognizer is not processing audio input from the buffer, but is temporarily halted while an application updates its grammars. A key distinction between the `PAUSED` state and the `SUSPENDED` state is that in the `SUSPENDED` state audio input can be still be coming into the audio input buffer. When the recognizer leaves the `SUSPENDED` state the audio is processed. The `SUSPENDED` state allows a user to continue talking to the recognizer even while the recognizer is temporarily `SUSPENDED`. Furthermore, by updating grammars in the `SUSPENDED` state, an application can apply multiple grammar changes instantaneously with respect to the audio input stream.

4.5 Speech Events

Speech engines, both recognizers and synthesizers, generate many types of events. Applications are not required to handle all events, however, some events are particularly important for implementing speech applications. For example, some result events must be processed to receive recognized text from a recognizer.

Java Speech API events follow the JavaBeans event model. Events are issued to a listener attached to an object involved in generating that event. All the speech events are derived from the `SpeechEvent` class in the `javax.speech` package.

The events of the `javax.speech` package are listed in Table 4-4.

Table 4-4 Speech events: javax.speech package

Name	Description
`SpeechEvent`	Parent class of all speech events.
`EngineEvent`	Indicates a change in speech engine state.
`AudioEvent`	Indicates an audio input or output event.
`EngineErrorEvent`	Sub-class of `EngineEvent` that indicates an asynchronous problems has occurred in the engine.

The events of the `javax.speech.synthesis` package are listed in Table 4-5.

Table 4-5 Speech events: javax.speech.synthesis package

Name	Description
`SynthesizerEvent`	Extends the `EngineEvent` for the specialized events of a `Synthesizer`.
`SpeakableEvent`	Indicates the progress in output of synthesized text.

The events of the `javax.speech.recognition` package are listed in Table 4-6.

Table 4-6 Speech events: javax.speech.recognition package

Name	Description
`RecognizerEvent`	Extends the `EngineEvent` for the specialized events of a `Recognizer`.
`GrammarEvent`	Indicates an update of or a status change of a recognition grammar.
`ResultEvent`	Indicates status and data changes of recognition results.
`RecognizerAudioEvent`	Extends `AudioEvent` with events for start and stop of speech and audio level updates.

4.5.1 Event Synchronization

A speech engine is required to provide all its events in synchronization with the AWT event queue whenever possible. The reason for this constraint is that it simplifies to integration of speech events with AWT events and the Java Foundation Classes events (e.g., keyboard, mouse and focus events). This constraint does not adversely affect applications that do not provide graphical interfaces.

Synchronization with the AWT event queue means that the AWT event queue is not issuing another event when the speech event is being issued. To implement this, speech engines need to place speech events onto the AWT event queue. The queue is obtained through the AWT `Toolkit`:

```
EventQueue q = Toolkit.getDefaultToolkit().getSystemEventQueue();
```

The `EventQueue` runs a separate thread for event dispatch. Speech engines are not required to issue the events through that thread, but should ensure that thread is blocked while the speech event is issued.

Note that `SpeechEvent` is not a sub-class of `AWTEvent`, and that speech events are not actually placed directly on the AWT event queue. Instead, a speech engine is performing internal activities to keep its internal speech event queue synchronized with the AWT event queue to make an application developer's life easier.

4.6 Other Engine Functions

4.6.1 Runtime Engine Properties

Speech engines each have a set of properties that can be changed while the engine is running. The `EngineProperties` interface defined in the `javax.speech` package is the root interface for accessing runtime properties. It is extended by the `SynthesizerProperties` interface defined in the `javax.speech.synthesis` package, and the `RecognizerProperties` interface defined in the `javax.speech.recognition` package.

For any engine, the `EngineProperties` is obtained by calling the `EngineProperties` method defined in the `Engine` interface. To avoid casting the return object, the `getSynthesizerProperties` method of the `Synthesizer` interface and the `getRecognizerProperties` method of the `Recognizer` interface are also provided to return the appropriate type. For example:

```
{
   Recognizer rec = ...;
   RecognizerProperties props = rec.getRecognizerProperties();
}
```

The `EngineProperties` interface provides three types of functionality.

- ♦ The `addPropertyChangeListener` and `removePropertyChangeListener` methods add or remove a JavaBeans `PropertyChangeListener`. The listener receives an event notification any time a property value changes.

- The `getControlComponent` method returns an engine-provided AWT `Component` or `null` if one is not provided by the engine. This component can be displayed for a user to modify the engine properties. In some cases this component may allow customization of properties that are not programmatically accessible.
- The `reset` method is used to set all engine properties to default values.

The `SynthesizerProperties` and `RecognizerProperties` interfaces define the sets of runtime features of those engine types. These specific properties defined by these interfaces are described in Chapter 5 and Chapter 6 respectively.

For each property there is a get and a set method, both using the JavaBeans property patterns. For example, the methods for handling a synthesizer's speaking voice are:

```
float getVolume()
void setVolume(float voice) throws PropertyVetoException;
```

The get method returns the current setting. The set method attempts to set a new volume. A set method throws an exception if it fails. Typically, this is because the engine rejects the set value. In the case of volume, the legal range is 0.0 to 1.0. Values outside of this range cause an exception.

The set methods of the `SynthesizerProperties` and `RecognizerProperties` interfaces are asynchronous - they may return before the property change takes effect. For example, a change in the voice of a synthesizer may be deferred until the end of the current word, the current sentence or even the current document. So that an application knows when a change occurs, a `PropertyChangeEvent` is issued to each `PropertyChangeListener` attached to the properties object.

A property change event may also be issued because another application has changed a property, because changing one property affects another (e.g., changing a synthesizer's voice from male to female will usually cause an increase in the pitch setting), or because the property values have been reset.

4.6.2 Audio Management

The `AudioManager` of a speech engine is provided for management of the engine's speech input or output. For the Java Speech API Version 1.0 specification, the `AudioManager` interface is minimal. As the audio streaming interfaces for the Java platform are established, the `AudioManager` interface will be enhanced for more advanced functionality.

For this release, the `AudioManager` interface defines the ability to attach and remove `AudioListener` objects. For this release, the `AudioListener` interface is simple: it is empty. However, the `RecognizerAudioListener` interface extends the `AudioListener` interface to receive three audio event types (`SPEECH_STARTED`, `SPEECH_STOPPED` and `AUDIO_LEVEL` events). These events are described in detail in Chapter 6. As a type of `AudioListener`, a `RecognizerAudioListener` is attached and removed through the `AudioManager`.

4.6.3 Vocabulary Management

An engine can optionally provide a `VocabManager` for control of the pronunciation of words and other vocabulary. This manager is obtained by calling the `getVocabManager` method of a `Recognizer` or `Synthesizer` (it is a method of the `Engine` interface). If the engine does not support vocabulary management, the method returns `null`.

The manager defines a list of `Word` objects. Words can be added to the `VocabManager`, removed from the `VocabManager`, and searched through the `VocabManager`.

The `Word` class is defined in the `javax.speech` package. Each `Word` is defined by the following features.

- *Written form*: a required `String` that defines how the `Word` should be presented visually.
- *Spoken form*: an optional `String` that indicates how the `Word` is spoken. For English, the spoken form might be used for defining how acronyms are spoken. For Japanese, the spoken form could provide a *kana* representation of how *kanji* in the written form is pronounced.
- *Pronunciations*: an optional `String` array containing one or more phonemic representations of the pronunciations of the `Word`. The International Phonetic Alphabet subset of Unicode is used throughout the Java Speech API for representing pronunciations.
- *Grammatical categories*: an optional set of or'ed grammatical categories. The `Word` class defines 16 different classes of words (noun, verb, conjunction etc.). These classes do not represent a complete linguistic breakdown of all languages. Instead they are intended to provide a `Recognizer` or `Synthesizer` with additional information about a word that may assist in correctly recognizing or correctly speaking it.

CHAPTER 5

Speech Synthesis: javax.speech.synthesis

A speech synthesizer is a speech engine that converts text to speech. The `javax.speech.synthesis` package defines the `Synthesizer` interface to support speech synthesis plus a set of supporting classes and interfaces. The basic functional capabilities of speech synthesizers, some of the uses of speech synthesis and some of the limitations of speech synthesizers are described in Section 2.1 (on page 9).

As a type of speech engine, much of the functionality of a `Synthesizer` is inherited from the `Engine` interface in the `javax.speech` package and from other classes and interfaces in that package. The `javax.speech` package and generic speech engine functionality are described in Chapter 4.

This chapter describes how to write Java applications and applets that use speech synthesis. We begin with a simple example, and then review the speech synthesis capabilities of the API in more detail.

- "Hello World!": a simple example of speech synthesis
- Synthesizer as an Engine
- Speaking Text
- Speech Output Queue
- Monitoring Speech Output
- Synthesizer Properties

5.1 "Hello World!"

The following code shows a simple use of speech synthesis to speak the string "Hello World".

```
import javax.speech.*;
import javax.speech.synthesis.*;
import java.util.Locale;

public class HelloWorld {
   public static void main(String args[]) {
       try {
           // Create a synthesizer for English
           Synthesizer synth = Central.createSynthesizer(
              new SynthesizerModeDesc(Locale.ENGLISH));

           // Get it ready to speak
           synth.allocate();
           synth.resume();

           // Speak the "Hello world" string
           synth.speakPlainText("Hello, world!", null);

           // Wait till speaking is done
           synth.waitEngineState(Synthesizer.QUEUE_EMPTY);

           // Clean up
           synth.deallocate();
       } catch (Exception e) {
           e.printStackTrace();
       }
   }
}
```

This example illustrates the four basic steps which all speech synthesis applications must perform. Let's examine each step in detail.

- *Create:* The `Central` class of `javax.speech` package is used to obtain a speech synthesizer by calling the `createSynthesizer` method. The `SynthesizerModeDesc` argument provides the information needed to locate an appropriate synthesizer. In this example a synthesizer that speaks English is requested.
- *Allocate and Resume:* The `allocate` and `resume` methods prepare the `Synthesizer` to produce speech by allocating all required resources and put-

ting it in the `RESUMED` state.

- *Generate:* The `speakPlainText` method requests the generation of synthesized speech from a string.
- *Deallocate:* The `waitEngineState` method blocks the caller until the `Synthesizer` is in the `QUEUE_EMPTY` state — until it has finished speaking the text. The `deallocate` method frees the synthesizer's resources.

5.2 Synthesizer as an Engine

The basic functionality provided by a `Synthesizer` is speaking text, management of a queue of text to be spoken and producing events as these functions proceed. The `Synthesizer` interface extends the `Engine` interface to provide this functionality.

The following is a list of the functionality that the `javax.speech.synthesis` package inherits from the `javax.speech` package and outlines some of the ways in which that functionality is specialized.

- The properties of a speech engine defined by the `EngineModeDesc` class apply to synthesizers. The `SynthesizerModeDesc` class adds information about synthesizer voices. Both `EngineModeDesc` and `SynthesizerModeDesc` are described in Section 4.2 (on page 36).
- Synthesizers are searched, selected and created through the `Central` class in the `javax.speech` package as described in Section 4.3 (on page 39). That section explains default creation of a synthesizer, synthesizer selection according to defined properties, and advanced selection and creation mechanisms.
- Synthesizers inherit the basic state system of an engine from the `Engine` interface. The basic engine states are `ALLOCATED`, `DEALLOCATED`, `ALLOCATING_RESOURCES` and `DEALLOCATING_RESOURCES` for allocation state, and `PAUSED` and `RESUMED` for audio output state. The `getEngineState` method and other methods are inherited for monitoring engine state. An `EngineEvent` indicates state changes. The engine state systems are described in Section 4.4 (on page 44). (The `QUEUE_EMPTY` and `QUEUE_NOT_EMPTY` states added by synthesizers are described in Section 5.4.)
- Synthesizers produce all the standard engine events (see Section 4.5). The `javax.speech.synthesis` package also extends the `EngineListener` interface as `SynthesizerListener` to provide events that are specific to synthesizers.

- Other engine functionality inherited as an engine includes the runtime properties (see Section 4.6.1 and Section 5.6), audio management (see Section 4.6.2) and vocabulary management (see Section 4.6.3).

5.3 Speaking Text

The `Synthesizer` interface provides four methods for submitting text to a speech synthesizer to be spoken. These methods differ according to the formatting of the provided text, and according to the type of object from which the text is produced. All methods share one feature; they all allow a listener to be passed that will receive notifications as output of the text proceeds.

The simplest method — `speakPlainText` — takes text as a `String` object. This method is illustrated in the *"Hello World!"* example at the beginning of this chapter. As the method name implies, this method treats the input text as plain text without any of the formatting described below.

The remaining three speaking methods — all named `speak` — treat the input text as being specially formatted with the Java Speech Markup Language (JSML). JSML is an application of XML (eXtensible Markup Language), a data format for structured document interchange on the internet. JSML allows application developers to annotate text with structural and presentation information to improve the speech output quality. JSML is defined in detail in a separate technical document, *"The Java Speech Markup Language Specification."*

The three `speak` methods retrieve the JSML text from different Java objects. The three methods are:

```
void speak(Speakable text, SpeakableListener listener);
void speak(URL text, SpeakableListener listener);
void speak(String text, SpeakableListener listener);
```

The first version accepts an object that implements the `Speakable` interface. The `Speakable` interface is a simple interface defined in the `javax.speech.synthesis` package that contains a single method: `getJSMLText`. This method should return a `String` containing text formatted with JSML.

Virtually any Java object can implement the `Speakable` interface by implementing the `getJSMLText` method. For example, the cells of spread-sheet, the text of an editing window, or extended AWT classes could all implement the `Speakable` interface.

The `Speakable` interface is intended to provide the spoken version of the `toString` method of the `Object` class. That is, `Speakable` allows an object to define how it should be spoken. For example:

```
public class MyAWTObj extends Component implements Speakable {
   ...
   public String getJSMLText() {
       ...
   }
}

{
   MyAWTObj obj = new MyAWTObj();
   synthesizer.speak(obj, null);
}
```

The second variant of the `speak` method allows JSML text to be loaded from a URL to be spoken. This allows JSML text to be loaded directly from a web site and be spoken.

The third variant of the `speak` method takes a JSML string. Its use is straightforward.

For each of the three `speak` methods that accept JSML formatted text, a `JSMLException` is thrown if any formatting errors are detected. Developers familiar with editing HTML documents will find that XML is strict about syntax checks. It is generally advisable to check XML documents (such as JSML) with XML tools before publishing them.

The following sections describe the speech output onto which objects are placed with calls to the speak methods and the mechanisms for monitoring and managing that queue.

5.4 Speech Output Queue

Each call to the `speak` and `speakPlainText` methods places an object onto the synthesizer's *speech output queue*. The speech output queue is a FIFO queue: first-in-first-out. This means that objects are spoken in the order in which they are received.

The *top of queue* item is the head of the queue. The top of queue item is the item currently being spoken or is the item that will be spoken next when a paused synthesizer is resumed.

The `Synthesizer` interface provides a number of methods for manipulating the output queue. The `enumerateQueue` method returns an `Enumeration` object containing a `SynthesizerQueueItem` for each object on the queue. The first object in the enumeration is the top of queue. If the queue is empty the `enumerateQueue` method returns `null`.

Each `SynthesizerQueueItem` in the enumeration contains four properties. Each property has a accessor method:

- `getSource` returns the source object for the queue item. The source is the object passed to the `speak` and `speakPlainText` method: `a Speakable` object, a `URL` or a `String`.
- `getText` returns the text representation for the queue item. For a `Speakable` object it is the `String` returned by the `getJSMLText` method. For a `URL` it is the `String` loaded from that URL. For a string source, it is that string object.
- `isPlainText` allows an application to distinguish between plain text and JSML objects. If this method returns true the string returned by `getText` is plain text.
- `getSpeakableListener` returns the listener object to which events associated with this item will be sent. If no listener was provided in the call to `speak` and `speakPlainText` then the call returns `null`.

The state of the queue is an explicit state of the `Synthesizer`. The `Synthesizer` interface defines a state system for `QUEUE_EMPTY` and `QUEUE_NOT_EMPTY`. Any `Synthesizer` in the `ALLOCATED` state must be in one and only one of these two states.

The `QUEUE_EMPTY` and `QUEUE_NOT_EMPTY` states are parallel states to the `PAUSED` and `RESUMED` states. These two state systems operate independently as shown in Figure 5-1 (an extension of Figure 4-2 on page 49).

The `SynthesizerEvent` class extends the `EngineEvent` class with the `QUEUE_UPDATED` and `QUEUE_EMPTIED` events which indicate changes in the queue state.

The *"Hello World!"* example shows one use of the queue status. It calls the `waitEngineState` method to test when the synthesizer returns to the `QUEUE_EMPTY` state. This test determines when the synthesizer has completed output of all objects on the speech output queue.

The queue status and transitions in and out of the `ALLOCATED` state are linked. When a `Synthesizer` is newly `ALLOCATED` it always starts in the `QUEUE_EMPTY` state since no objects have yet been placed on the queue. Before a synthesizer is deallocated (before leaving the `ALLOCATED` state) a synthesizer must return to the `QUEUE_EMPTY` state. If the speech output queue is not empty when the `deallocate` method is called, all objects on the speech output queue are automatically cancelled by the synthesizer. By contrast, the initial and final states for `PAUSED` and `RESUMED` are not defined because the pause/resume state may be shared by multiple applications.

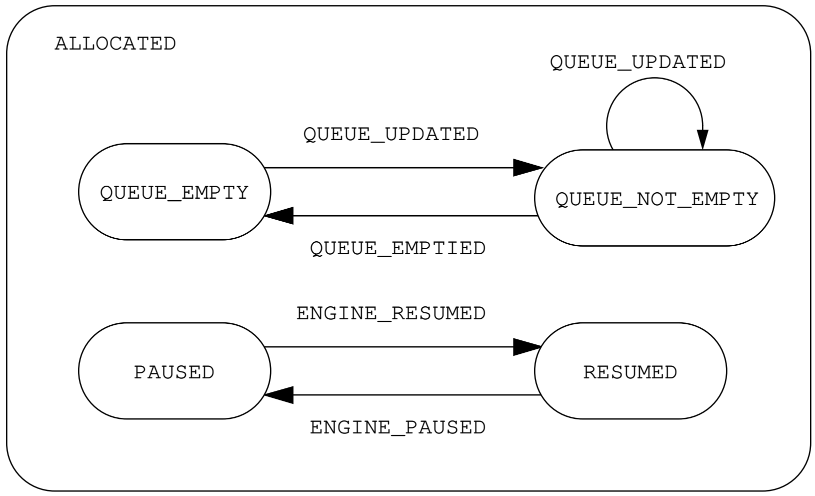

Figure 5-1 Synthesizer states

The `Synthesizer` interface defines three cancel methods that allow an application to request that one or more objects be removed from the speech output queue:

```
void cancel();
void cancel(Object source);
void cancelAll();
```

The first of these three methods cancels the object at the top of the speech output queue. If that object is currently being spoken, the speech output is stopped and then the object is removed from the queue. The `SpeakableListener` for the item receives a `SPEAKABLE_CANCELLED` event. The `SynthesizerListener` receives a `QUEUE_UPDATED` event, unless the item was the last one on the queue in which case a `QUEUE_EMPTIED` event is issued.

The second cancel method requires that a source object be specified. The object should be one of the items currently on the queue: a `Speakable`, a `URL`, or a `String`. The actions are much the same as for the first cancel method except that if the item is not top-of-queue, then speech output is not affected.

The final cancel method — `cancelAll` — removes all items from the speech output queue. Each item receives a `SPEAKABLE_CANCELLED` event and the

SynthesizerListener receives a QUEUE_EMPTIED event. The SPEAKABLE_CANCELLED events are issued to items in the order of the queue.

5.5 Monitoring Speech Output

All the speak and speakPlainText methods accept a SpeakableListener as the second input parameter. To request notification of events as the speech object is spoken an application provides a non-null listener.

Unlike a SynthesizerListener that receives synthesizer-level events, a SpeakableListener receives events associated with output of individual text objects: output of Speakable objects, output of URLs, output of JSML strings, or output of plain text strings.

The mechanism for attaching a SpeakableListener through the speak and speakPlainText methods is slightly different from the normal attachment and removal of listeners. There are, however, addSpeakableListener and removeSpeakableListener methods on the Synthesizer interface. These add and remove methods allow listeners to be provided to receive notifications of events associated with *all* objects being spoken by the Synthesizer.

The SpeakableEvent class defines eight events that indicate progress of spoken output of a text object. For each of these eight event types, there is a matching method in the SpeakableListener interface. For convenience, a SpeakableAdapter implementation of the SpeakableListener interface is provided with trivial (empty) implementations of all eight methods.

The normal sequence of events as an object is spoken is as follows:

- TOP_OF_QUEUE: the object has reached to the top of the speech output queue and is the next object to be spoken.
- SPEAKABLE_STARTED: audio output has commenced for this text object.
- WORD_STARTED: audio output has reached the start of a word. The event includes information on the location of the word in the text object. This event is issued for each word in the text object. This event is often used to highlight words in a text document as they are spoken.
- MARKER_REACHED: audio output has reached the location of a MARKER tag explicitly embedded in the JSML text. The event includes the marker text from the tag. For container JSML elements, a MARKER_REACHED event is issued at both the start and end of the element. MARKER_REACHED events are not produced for plain text because formatting is required to add the markers.
- SPEAKABLE_ENDED: audio output has been completed and the object has

been removed from the speech output queue.

The remaining event types are modifications to the normal event sequence.

- `SPEAKABLE_PAUSED`: the `Synthesizer` has been paused so audio output of this object is paused. This event is only issued to the text object at the top of the speech output queue.
- `SPEAKABLE_RESUMED`: the `Synthesizer` has been resumed so audio output of this object has resumed. This event is only issued to the text object at the top of the speech output queue.
- `SPEAKABLE_CANCELLED`: the object has been removed from the speech output queue. Any or all objects in the speech output queue may be removed by one of the cancel methods (described in Section 5.4 on page 61).

The following is an example of the use of the `SpeakableListener` interface to monitor the progress of speech output. It shows how a training application could synchronize speech synthesis with animation.

It places two JSML string objects onto the output queue and requests notifications to itself. The speech output will be:

```
"First, use the mouse to open the file menu.
 Then, select the save command."
```

At the start of the output of each string the `speakableStarted` method will be called. By checking the source of the event we determine which text is being spoken and so the appropriate animation code can be triggered.

```
public class TrainingApp extends SpeakableAdapter {

   String openMenuText =
            "First, use the mouse to open the file menu.";
   // The EMP element indicates emphasis of a word
   String selectSaveText =
            "Then, select the <EMP>save</EMP> command.";

   public void sendText(Synthesizer synth) {
      // Insert the two objects into the speech queue
      // specifying self as recipient of SpeakableEvents.
      synth.speak(openMenuText, this);
      synth.speak(selectSaveText, this);
   }

   // Override the empty method in SpeakableAdapter
```

```
        public void speakableStarted(SpeakableEvent e) {
            if (e.getSource() == openMenuText) {
                // animate the opening of the file menu
            }
            else if (e.getSource() == selectSaveText) {
                // animate the selection of 'save'
            }
        }
    }
```

5.6 Synthesizer Properties

The `SynthesizerProperties` interface extends the `EngineProperties` interface described in Section 4.6.1 (on page 54). The JavaBeans property mechanisms, the asynchronous application of property changing, and the property change event notifications are all inherited engine behavior and are described in that section.

The `SynthesizerProperties` object is obtained by calling the `getEngineProperties` method (inherited from the `Engine` interface) or the `getSynthesizerProperties` method. Both methods return the same object instance, but the latter is more convenient since it is an appropriately cast object.

The `SynthesizerProperties` interface defines five synthesizer properties that can be modified during operation of a synthesizer to effect speech output.

The *voice* property is used to control the speaking voice of the synthesizer. The set of voices supported by a synthesizer can be obtained by the `getVoices` method of the synthesizer's `SynthesizerModeDesc` object. Each voice is defined by a voice name, gender, age and speaking style. Selection of voices is described in more detail in *Selecting Voices* on page 67.

The remaining four properties control *prosody*. Prosody is a set of features of speech including the pitch and intonation, rhythm and timing, stress and other characteristics which affect the style of the speech. The prosodic features controlled through the `SynthesizerProperties` interface are:

- *Volume:* a float value that is set on a scale from 0.0 (silence) to 1.0 (loudest).
- *Speaking rate:* a float value indicating the speech output rate in words per minute. Higher values indicate faster speech output. Reasonable speaking rates depend upon the synthesizer and the current voice (voices may have different natural speeds). Also, speaking rate is also dependent upon the language because of different conventions for what is a "word". For English, a typical speaking rate is around 200 words per minute.

- *Pitch:* the baseline pitch is a float value given in Hertz. Different voices have different natural sounding ranges of pitch. Typical male voices are between 80 and 180 Hertz. Female pitches typically vary from 150 to 300 Hertz.
- *Pitch range:* a float value indicating a preferred range for variation in pitch above the baseline setting. A narrow pitch range provides monotonous output while wide range provide a more lively voice. The pitch range is typically between 20% and 80% of the baseline pitch.

The following code shows how to increase the speaking rate for a synthesizer by 30 words per minute.

```
float increaseSpeakingRate(Synthesizer synth) {
    SynthesizerProperties props = synth.getEngineProperties();
    float newSpeakingRate = props.getSpeakingRate() + 30.0;
    props.setSpeakingRate(newSpeakingRate);
    return newSpeakingRate;
}
```

As with all engine properties, changes to synthesizer properties are not necessarily instant. The change should take effect as soon as the synthesizer can apply it. Depending on the underlying technology, a property change may take effect immediately, or at the next phoneme, word, phrase or sentence boundary, or at the beginning of output of the next item in the synthesizer's queue.

So that an application knows when the change has actual taken effect, the synthesizer generates a property change event for each call to a set method in the `SynthesizerProperties` interface.

5.6.1 Selecting Voices

Most speech synthesizers are able to produce a number of voices. In most cases voices attempt to sound natural and human, but some voices may be deliberately mechanical or robotic.

The `Voice` class is used to encapsulate the four features that describe each voice: voice name, gender, age and speaking style. The voice name and speaking style are both `String` objects and the contents of those strings are determined by the synthesizer. Typical voice names might be “Victor”, “Monica”, “Ahmed”, “Jose”, “My Robot” or something completely different. Speaking styles might include “casual”, “business”, “robotic” or “happy” (or similar words in other languages) but the API does not impose any restrictions upon the speaking styles. For both voice name and speaking style, synthesizers are encouraged to use strings that are meaningful to users so that they can make sensible judgements when selecting voices.

By contrast the gender and age are both defined by the API so that programmatic selection is possible. The gender of a voice can be `GENDER_FEMALE`, `GENDER_MALE`, `GENDER_NEUTRAL` or `GENDER_DONT_CARE`. Male and female are hopefully self-explanatory. Gender neutral is intended for voices that are not clearly male or female such as some robotic or artificial voices. The "don't care" values are used when selecting a voice and the feature is not relevant.

The age of a voice can be `AGE_CHILD` (up to 12 years), `AGE_TEENAGER` (13-19), `AGE_YOUNGER_ADULT` (20-40), `AGE_MIDDLE_ADULT` (40-60), `AGE_OLDER_ADULT` (60+), `AGE_NEUTRAL`, and `AGE_DONT_CARE`.

Both gender and age are OR'able values for both applications and engines. For example, an engine could specify a voice as:

```
Voice("name", GENDER_MALE, AGE_CHILD | AGE_TEENAGER, "style");
```

In the same way that mode descriptors are used by engines to describe themselves and by applications to select from amongst available engines, the `Voice` class is used both for description and selection. The `match` method of `Voice` allows an application to test whether an engine-provided voice has suitable properties.

The following code shows the use of the `match` method to identify voices of a synthesizer that are either male or female voices and that are younger or middle adults (between 20 and 60). The `SynthesizerModeDesc` object may be one obtained through the `Central` class or through the `getEngineModeDesc` method of a created `Synthesizer`.

```
SynthesizerModeDesc desc = ...;
Voice[] voices = desc.getVoices();

// Look for male or female voices that are young/middle adult
Voice myVoice = new Voice();
myVoice.setGender(GENDER_MALE | GENDER_FEMALE);
myVoice.setAge(AGE_YOUNGER_ADULT | AGE_MIDDLE_ADULT);

for (int i = 0; i < voices.length; i++)
    if (voices[i].match(myVoice))
        doAction(voices[i]);
```

The `Voice` object can also be used in the selection of a speech synthesizer. The following code illustrates how to create a synthesizer with a young female Japanese voice.

```
SynthesizerModeDesc required = new SynthesizerModeDesc();
Voice voice = new Voice(null, GENDER_FEMALE,
                AGE_CHILD | AGE_TEENAGER, null);

required.addVoice(voice);
required.setLocale(Locale.JAPAN);

Synthesizer synth = Central.createSynthesizer(required);
```

5.6.2 Property Changes in JSML

In addition to control of speech output through the `SynthesizerProperties` interface, all five synthesizer properties can be controlled in JSML text provided to a synthesizer. The advantage of control through JSML text is that property changes can be finely controlled within a text document. By contrast, control of the synthesizer properties through the `SynthesizerProperties` interface is not appropriate for word-level changes but is instead useful for setting the default configuration of the synthesizer. Control of the `SynthesizerProperties` interface is often presented to the user as a graphical configuration window.

Applications that generate JSML text should respect the default settings of the user. To do this, relative settings of parameters such as pitch and speaking rate should be used rather than absolute settings.

For example, users with vision impairments often set the speaking rate extremely high — up to 500 words per minute — so high that most people do not understand the synthesized speech. If a document uses an absolute speaking rate change (to say 200 words per minute which is fast for most users), then the user will be frustrated.

Changes made to the synthesizer properties through the `SynthesizerProperties` interface are persistent: they affect all succeeding speech output. Changes in JSML are explicitly localized (all property changes in JSML have both start and end tags).

5.6.3 Controlling Prosody

The prosody and voice properties can be used within JSML text to substantially improve the clarity and naturalness of the speech output. For example, one time to change prosodic settings is when providing new, important or detailed information. In this instance it is typical for a speaker to slow down, emphasise more words and often add extra pauses. Putting equivalent changes into synthetic speech will help a listener understand the message.

For example, in response to the question “How many Acme shares do I have?”, the answer might be “You currently have 1,500 Acme shares.” The number will spoken more slowly because it is new information. To represent this in JSML text the `<PROS>` element is used:

```
You currently have <PROS RATE=”-20%”>1500</PROS> Acme shares.
```

The following example illustrates how an email message header object can implement the `Speakable` interface and generate JSML text with prosodic controls to improve understandability.

```
public class MailHeader implements Speakable {
   public String subject;
   public String sender;    // sender of the message, eg John Doe
   public String date;

   /** getJSMLText is the only method of Speakable */
   public String getJSMLText() {
      StringBuffer buf = new StringBuffer();

      // Speak the sender’s name slower to be clearer
      buf.append("Message from " +
         "<PROS RATE=-30>" + sender + ",</PROS>");

      // Make sure the date is interpreted correctly
      // But we don’t need it slow - it’s not so important
      buf.append(" delivered " +
         "<SAYAS class=\"date\">" + date + "</SAYAS>");

      // Subject slower too
      buf.append(", with subject: " +
         "<PROS RATE=-30>" + subject + "</PROS>");

      return buf.toString();
   }
}

public class myMailApp {
   ...
   void newMessageRecieved(MailHeader header) {
      synth.speakPlainText("You have new mail!");
      synth.speak(header, mySpeakableListener);
   }
}
```

CHAPTER 6

Speech Recognition: javax.speech.recognition

A speech recognizer is a speech engine that converts speech to text. The `javax.speech.recognition` package defines the `Recognizer` interface to support speech recognition plus a set of supporting classes and interfaces. The basic functional capabilities of speech recognizers, some of the uses of speech recognition and some of the limitations of speech recognizers are described in Section 2.2 (on page 13).

As a type of speech engine, much of the functionality of a `Recognizer` is inherited from the `Engine` interface in the `javax.speech` package and from other classes and interfaces in that package. The `javax.speech` package and generic speech engine functionality are described in Chapter 4.

The Java Speech API is designed to keep simple speech applications simple and to make advanced speech applications possible for non-specialist developers. This chapter covers both the simple and advanced capabilities of the `javax.speech.recognition` package. Where appropriate, some of the more advanced sections are marked so that you can choose to skip them. We begin with a simple code example, and then review the speech recognition capabilities of the API in more detail through the following sections:

- *"Hello World!":* a simple example of speech recognition
- Recognizer as an Engine
- Recognizer State Systems
- Recognition Grammars
- Rule Grammars

- ♦ Dictation Grammars
- ♦ Recognition Results
- ♦ Recognizer Properties
- ♦ Speaker Management
- ♦ Recognizer Audio

6.1 “Hello World!”

The following example shows a simple application that uses speech recognition. For this application we need to define a *grammar* of everything the user can say, and we need to write the Java software that performs the recognition task.

A grammar is provided by an application to a speech recognizer to define the words that a user can say, and the patterns in which those words can be spoken. In this example, we define a grammar that allows a user to say “Hello World” or a variant. The grammar is defined using the Java Speech Grammar Format. This format is documented in the *Java Speech Grammar Format Specification* (available from `http://java.sun.com/products/java-media/speech/`).

Place this grammar into a file.

```
grammar javax.speech.demo;

public <sentence> = hello world | good morning |
                    hello mighty computer;
```

This trivial grammar has a single *public rule* called “`sentence`”. A rule defines what may be spoken by a user. A public rule is one that may be *activated* for recognition.

The following code shows how to create a recognizer, load the grammar, and then wait for the user to say something that matches the grammar. When it gets a match, it deallocates the engine and exits.

```
import javax.speech.*;
import javax.speech.recognition.*;
import java.io.FileReader;
import java.util.Locale;

public class HelloWorld extends ResultAdapter {
   static Recognizer rec;
```

```
    // Receives RESULT_ACCEPTED event: print it, clean up, exit
    public void resultAccepted(ResultEvent e) {
        Result r = (Result)(e.getSource());
        ResultToken tokens[] = r.getBestTokens();

        for (int i = 0; i < tokens.length; i++)
            System.out.print(tokens[i].getSpokenText() + " ");
        System.out.println();

        // Deallocate the recognizer and exit
        rec.deallocate();
        System.exit(0);
    }

    public static void main(String args[]) {
        try {
            // Create a recognizer that supports English.
            rec = Central.createRecognizer(
                            new EngineModeDesc(Locale.ENGLISH));

            // Start up the recognizer
            rec.allocate();

            // Load the grammar from a file, and enable it
            FileReader reader = new FileReader(args[0]);
            RuleGrammar gram = rec.loadJSGF(reader);
            gram.setEnabled(true);

            // Add the listener to get results
            rec.addResultListener(new HelloWorld());

            // Commit the grammar
            rec.commitChanges();

            // Request focus and start listening
            rec.requestFocus();
            rec.resume();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

This example illustrates the basic steps which all speech recognition applications must perform. Let's examine each step in detail.

- *Create:* The `Central` class of `javax.speech` package is used to obtain a speech recognizer by calling the `createRecognizer` method. The `EngineModeDesc` argument provides the information needed to locate an appropriate recognizer. In this example we requested a recognizer that understands English (since the grammar is written for English).

- *Allocate:* The `allocate` methods requests that the `Recognizer` allocate all necessary resources.
- *Load and enable grammars:* The `loadJSGF` method reads in a JSGF document from a reader created for the file that contains the `javax.speech.demo` grammar. (Alternatively, the `loadJSGF` method can load a grammar from a URL.) Next, the grammar is *enabled*. Once the recognizer receives focus (see below), an enabled grammar is *activated* for recognition: that is, the recognizer compares incoming audio to the active grammars and listens for speech that matches those grammars.
- *Attach a ResultListener:* The `HelloWorld` class extends the `ResultAdapter` class which is a trivial implementation of the `ResultListener` interface. An instance of the `HelloWorld` class is attached to the Recognizer to receive result events. These events indicate progress as the recognition of speech takes place. In this implementation, we process the `RESULT_ACCEPTED` event, which is provided when the recognizer completes recognition of input speech that matches an active grammar.
- *Commit changes:* Any changes in grammars and the grammar enabled status needed to be *committed* to take effect (that includes creation of a new grammar). The reasons for this are described in Section 6.4.2.
- *Request focus and resume:* For recognition of the grammar to occur, the recognizer must be in the `RESUMED` state and must have the speech focus. The `requestFocus` and `resume` methods achieve this.
- *Process result:* Once the `main` method is completed, the application waits until the user speaks. When the user speaks something that matches the loaded grammar, the recognizer issues a `RESULT_ACCEPTED` event to the listener we attached to the recognizer. The source of this event is a `Result` object that contains information about what the recognizer heard. The `getBestTokens` method returns an array of `ResultTokens`, each of which represents a single spoken word. These words are printed.
- *Deallocate:* Before exiting we call `deallocate` to free up the recognizer's resources.

6.2 Recognizer as an Engine

The basic functionality provided by a `Recognizer` includes grammar management and the production of results when a user says things that match active grammars. The `Recognizer` interface extends the `Engine` interface to provide this functionality.

The following is a list of the functionality that the `javax.speech.recognition` package inherits from the `javax.speech` package and outlines some of the ways in which that functionality is specialized.

- The properties of a speech engine defined by the `EngineModeDesc` class apply to recognizers. The `RecognizerModeDesc` class adds information about dictation capabilities of a recognizer and about users who have trained the engine. Both `EngineModeDesc` and `RecognizerModeDesc` are described in Section 4.2 (on page 36).
- Recognizers are searched, selected and created through the `Central` class in the `javax.speech` package as described in Section 4.3 (on page 39). That section explains default creation of a recognizer, recognizer selection according to defined properties, and advanced selection and creation mechanisms.
- Recognizers inherit the basic state systems of an engine from the `Engine` interface, including the four allocation states, the pause and resume state, the state monitoring methods and the state update events. The engine state systems are described in Section 4.4 (on page 44). The two state systems added by recognizers are described in Section 6.3.
- Recognizers produce all the standard engine events (see Section 4.5). The `javax.speech.recognition` package also extends the `EngineListener` interface as `RecognizerListener` to provide events that are specific to recognizers.
- Other engine functionality inherited as an engine includes the runtime properties (see Section 4.6.1 and Section 6.8), audio management (see Section 4.6.2) and vocabulary management (see Section 4.6.3).

6.3 Recognizer State Systems

6.3.1 Inherited States

As mentioned above, a `Recognizer` inherits the basic state systems defined in the `javax.speech` package, particularly through the `Engine` interface. The basic engine state systems are described in Section 4.4 (on page 44). In this section the two state systems added for recognizers are described. These two states systems represent the status of recognition processing of audio input against grammars, and the recognizer focus.

As a summary, the following state system functionality is inherited from the `javax.speech` package.

- The basic engine state system represents the current allocation state of the engine: whether resources have been obtained for 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 states of a recognizer indicate whether audio input is on or off. Pausing a recognizer is analogous to turning off the input microphone: input audio is lost. Section 4.4.7 (on page 51) describes the effect of pausing and resuming a recognizer in more detail.
- The `getEngineState` method of the `Engine` interface returns a `long` value representing the current engine state. The value has a bit set for each of the current states of the recognizer. For example and `ALLOCATED` recognizer in the `RESUMED` state will have both the `ALLOCATED` and `RESUMED` bits set.
- The `testEngineState` and `waitEngineState` methods are convenience methods for monitoring engine state. The test method tests for presence in a specified state. The wait method blocks until a specific state is reached.
- An `EngineEvent` is issued to `EngineListeners` each time an engine changes state. The event class includes the new and old engine states.

The recognizer adds two sub-state systems to the `ALLOCATED` state: that's in addition to the inherited pause and resume sub-state system. The two new sub-state systems represent the current activities of the recognizer's internal processing (the `LISTENING`, `PROCESSING` and `SUSPENDED` states) and the current recognizer focus (the `FOCUS_ON` and `FOCUS_OFF` states).

These new sub-state systems are parallel states to the `PAUSED` and `RESUMED` states and operate nearly independently as shown in Figure 6-1 (an extension of Figure 4-2 on page 49).

6.3.2 Recognizer Focus

The `FOCUS_ON` and `FOCUS_OFF` states indicate whether this instance of the `Recognizer` currently has the speech focus. Recognizer focus is a major determining factor in grammar activation, which, in turn, determines what the

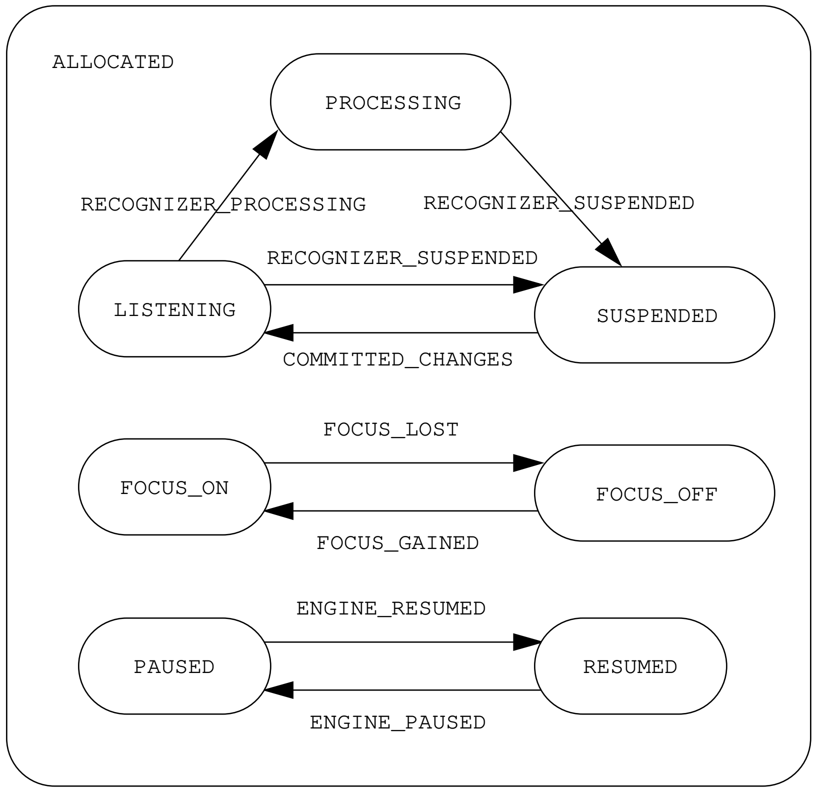

Figure 6-1 Recognizer states

recognizer is listening for at any time. The role of recognizer focus in activation and deactivation of grammars is described in Section 6.4.3 (on page 86).

A change in engine focus is indicated by a `RecognizerEvent` (which extends `EngineEvent`) being issued to `RecognizerListeners`. A `FOCUS_LOST` event indicates a change in state from `FOCUS_ON` to `FOCUS_OFF`. A `FOCUS_GAINED` event indicates a change in state from `FOCUS_OFF` to `FOCUS_ON`.

When a `Recognizer` has focus, the `FOCUS_ON` bit is set in the engine state. When a `Recognizer` does not have focus, the `FOCUS_OFF` bit is set. The following code examples monitor engine state:

```
Recognizer rec;

if (rec.testEngineState(Recognizer.FOCUS_ON)) {
```

```
        // we have focus so release it
        rec.releaseFocus();
    }
    // wait until we lose it
    rec.waitEngineState(Recognizer.FOCUS_OFF);
```

Recognizer focus is relevant to computing environments in which more than one application is using an underlying recognition. For example, in a desktop environment a user might be running a single speech recognition product (the underlying engine), but have multiple applications using the speech recognizer as a resource. These applications may be a mixture of Java and non-Java applications. Focus is not usually relevant in a telephony environment or in other speech application contexts in which there is only a single application processing the audio input stream.

The recognizer's focus should track the application to which the user is currently talking. When a user indicates that it wants to talk to an application (e.g., by selecting the application window, or explicitly saying "switch to application X"), the application requests speech focus by calling the `requestFocus` method of the `Recognizer`.

When speech focus is no longer required (e.g., the application has been iconized) it should call `releaseFocus` method to free up focus for other applications.

Both methods are asynchronous —the methods may return before the focus is gained or lost — since focus change may be deferred. For example, if a recognizer is in the middle of recognizing some speech, it will typically defer the focus change until the result is completed. The focus events and the engine state monitoring methods can be used to determine when focus is actually gained or lost.

The focus policy is determined by the underlying recognition engine — it is not prescribed by the `java.speech.recognition` package. In most operating environments it is reasonable to assume a policy in which the last application to request focus gets the focus.

Well-behaved applications adhere to the following convention to maximize recognition performance, to minimize their impact upon other applications and to maintain a satisfactory user interface experience. An application should only request focus when it is confident that the user's speech focus (attention) is directed towards it, and it should release focus when it is not required.

6.3.3 Recognition States

The most important (and most complex) state system of a recognizer represents the current recognition activity of the recognizer. An `ALLOCATED` `Recognizer` is always in one of the following three states:

- `LISTENING` state: The `Recognizer` is listening to incoming audio for speech that may match an active grammar but has not detected speech yet. A recognizer remains in this state while 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. While in this state, the recognizer is producing a result.
- `SUSPENDED` state: The `Recognizer` is temporarily suspended while grammars are updated. While suspended, audio input is buffered for processing once the recognizer returns to the `LISTENING` and `PROCESSING` states.

This sub-state system is shown in Figure 6-1. The typical state cycle of a recognizer is triggered by user speech. The recognizer starts in the `LISTENING` state, moves to the `PROCESSING` state while a user speaks, moves to the `SUSPENDED` state once recognition of that speech is completed and while grammars are updates in response to user input, and finally returns to the `LISTENING` state.

In this first event cycle a `Result` is typically produced that represents what the recognizer heard. Each `Result` has a state system and the `Result` state system is closely coupled to this `Recognizer` state system. The `Result` state system is discussed in Section 6.7 (on page 101). Many applications (including the *"Hello World!"* example) do not care about the recognition state but do care about the simpler `Result` state system.

The other typical event cycle also starts in the `LISTENING` state. Upon receipt of a non-speech event (e.g., keyboard event, mouse click, timer event) the recognizer is suspended temporarily while grammars are updated in response to the event, and then the recognizer returns to listening.

Applications in which grammars are affected by more than speech events need to be aware of the recognition state system.

The following sections explain these event cycles in more detail and discuss why speech input events are different in some respects from other event types.

6.3.3.1 Speech Events vs. Other Events

A keyboard event, a mouse event, a timer event, a socket event are all instantaneous in time — there is a defined instant at which they occur. The same is not true of speech for two reasons.

Firstly, speech is a temporal activity. Speaking a sentence takes time. For example, a short command such as "reload this web page" will take a second or two to speak, thus, it is not instantaneous. At the start of the speech the recognizer changes state, and as soon as possible after the end of the speech the recognizer produces a result containing the spoken words.

Secondly, recognizers cannot always recognize words immediately when they are spoken and cannot determine immediately when a user has stopped speaking. The reasons for these technical constraints upon recognition are outside the scope of this guide, but knowing about them is helpful in using a recognizer. (Incidentally, the same principals are generally true of human perception of speech.)

A simple example of why recognizers cannot always respond might be listening to a currency amount. If the user says "two dollars" or says "two dollars, fifty seconds" with a short pause after the word "dollars" the recognizer can't know immediately whether the user has finished speaking after the "dollars". What a recognizer must do is wait a short period — usually less than a second — to see if the user continues speaking. A second is a long time for a computer and complications can arise if the user clicks a mouse or does something else in that waiting period. (Section 6.8 on page 133 explains the time-out parameters that affect this delay.)

A further complication is introduced by the input audio buffering described in Section 6.3.

Putting all this together, there is a requirement for the recognizers to explicitly represent internal state through the `LISTENING`, `PROCESSING` and `SUSPENDED` states.

6.3.3.2 Speech Input Event Cycle

The typical recognition state cycle for a `Recognizer` occurs as speech input occurs. Technically speaking, this cycle represents the recognition of a single `Result`. The result state system and result events are described in detail in Section 6.7. The cycle described here is a clockwise trip through the `LISTENING`, `PROCESSING` and `SUSPENDED` states of an `ALLOCATED` recognizer as shown in Figure 6-1.

The `Recognizer` starts in the `LISTENING` state with a certain set of grammars enabled and active. When incoming audio is detected that may match an active grammar, the `Recognizer` transitions from the `LISTENING` state to the `PROCESSING` state with a `RECOGNIZER_PROCESSING` event.

The `Recognizer` then creates a new `Result` object and issues a `RESULT_CREATED` event (a `ResultEvent`) to provide the result to the application. At this point the result is usually empty: it does not contain any recognized words. As recognition proceeds words are added to the result along with other useful information.

The `Recognizer` remains in the `PROCESSING` state until it completes recognition of the result. While in the `PROCESSING` state the `Result` may be updated with new information.

The recognizer indicates completion of recognition by issuing a `RECOGNIZER_SUSPENDED` event to transition from the `PROCESSING` state to the `SUSPENDED` state. Once in that state, the recognizer issues a result *finalization* event to `ResultListeners` (`RESULT_ACCEPTED` or `RESULT_REJECTED` event) to indicate that all information about the result is finalized (words, grammars, audio etc.).

The `Recognizer` remains in the `SUSPENDED` state until processing of the result finalization event is completed. Applications will often make grammar changes during the result finalization because the result causes a change in application state or context.

In the `SUSPENDED` state the `Recognizer` buffers incoming audio. This buffering allows a user to continue speaking without speech data being lost. Once the `Recognizer` returns to the `LISTENING` state the buffered audio is processed to give the user the perception of real-time processing.

Once the result finalization event has been issued to all listeners, the `Recognizer` automatically commits all grammar changes and issues a `CHANGES_COMMITTED` event to return to the `LISTENING` state. (It also issues `GRAMMAR_CHANGES_COMMITTED` events to `GrammarListeners` of changed grammars.) The commit applies all grammar changes made at any point up to the end of result finalization, such as changes made in the result finalization events.

The `Recognizer` is now back in the `LISTENING` state listening for speech that matches the new grammars.

In this event cycle the first two recognizer state transitions (marked by `RECOGNIZER_PROCESSING` and `RECOGNIZER_SUSPENDED` events) are triggered by user actions: starting and stopping speaking. The third state transition (`CHANGES_COMMITTED` event) is triggered programmatically some time after the `RECOGNIZER_SUSPENDED` event.

The `SUSPENDED` state serves as a temporary state in which recognizer configuration can be updated without loosing audio data.

6.3.3.3 Non-Speech Event Cycle

For applications that deal only with spoken input the state cycle described above handles most normal speech interactions. For applications that handle other asynchronous input, additional state transitions are possible. Other types of asynchronous input include graphical user interface events (e.g., `AWTEvent`), timer events, multi-threading events, socket events and so on.

The cycle described here is temporary transition from the `LISTENING` state to the `SUSPENDED` and back as shown in Figure 6-1.

When a non-speech event occurs which changes the application state or application data it may be necessary to update the recognizer's grammars. The `suspend` and `commitChanges` methods of a `Recognizer` are used to handle non-speech asynchronous events. The typical cycle for updating grammars in response to a non-speech asynchronous events is as follows.

Assume that the `Recognizer` is in the `LISTENING` state (the user is not currently speaking). As soon as the event is received, the application calls `suspend` to indicate that it is about to change grammars. In response, the recognizer issues a `RECOGNIZER_SUSPENDED` event and transitions from the `LISTENING` state to the `SUSPENDED` state.

With the `Recognizer` in the `SUSPENDED` state, the application makes all necessary changes to the grammars. (The grammar changes affected by this event cycle and the pending commit are described in Section 6.4.2 on page 85.)

Once all grammar changes are completed the application calls the `commitChanges` method. In response, the recognizer applies the new grammars and issues a `CHANGES_COMMITTED` event to transition from the `SUSPENDED` state back to the `LISTENING` state. (It also issues `GRAMMAR_CHANGES_COMMITTED` events to all changed grammars.)

Finally, the `Recognizer` resumes recognition of the buffered audio and then live audio with the new grammars.

The suspend and commit process is designed to provide a number of features to application developers which help give users the perception of a responsive recognition system.

Because audio is buffered from the time of the asynchronous event to the time at which the `CHANGES_COMMITTED` occurs, the audio is processed as if the new grammars were applied exactly at the time of the asynchronous event. The user has the perception of real-time processing.

Although audio is buffered in the `SUSPENDED` state, applications should make grammar changes and call `commitChanges` as quickly as possible. This minimizes the amount of data in the audio buffer and hence the amount of time it takes for the recognizer to "catch up". It also minimizes the possibility of a buffer overrun.

Technically speaking, an application is not required to call `suspend` prior to calling `commitChanges`. If the `suspend` call is committed the `Recognizer` behaves as if suspend had been called immediately prior to calling `commitChanges`. However, an application that does not call `suspend` risks a commit occurring unexpectedly while it updates grammars with the effect of leaving grammars in an inconsistent state.

6.3.4 Interactions of State Systems

The three sub-state systems of an allocated recognizer (shown in Figure 6-1) normally operate independently. There are, however, some indirect interactions.

When a recognizer is paused, audio input is stopped. However, recognizers have a buffer between audio input and the internal process that matches audio against grammars, so recognition can continue temporarily after a recognizer is paused. In other words, a `PAUSED` recognizer may be in the `PROCESSING` state.

Eventually the audio buffer will empty. If the recognizer is in the `PROCESSING` state at that time then the result it is working on is immediately finalized and the recognizer transitions to the `SUSPENDED` state. Since a well-behaved application treats `SUSPENDED` state as a temporary state, the recognizer will eventually leave the `SUSPENDED` state by committing grammar changes and will return to the `LISTENING` state.

The `PAUSED`/`RESUMED` state of an engine is shared by multiple applications, so it is possible for a recognizer to be paused and resumed because of the actions of another application. Thus, an application should always leave its grammars in a state that would be appropriate for a `RESUMED` recognizer.

The focus state of a recognizer is independent of the `PAUSED` and `RESUMED` states. For instance, it is possible for a paused `Recognizer` to have `FOCUS_ON`. When the recognizer is resumed, it will have the focus and its grammars will be activated for recognition.

The focus state of a recognizer is very loosely coupled with the recognition state. An application that has no `GLOBAL` grammars (described in Section 6.4.3) will not receive any recognition results unless it has recognition focus.

6.4 Recognition Grammars

A *grammar* defines what a recognizer should listen for in incoming speech. Any grammar defines the set of tokens a user can say (a token is typically a single word) and the patterns in which those words are spoken.

The Java Speech API supports two types of grammars: *rule grammars* and *dictation grammars*. These grammars differ in how patterns of words are defined. They also differ in their programmatic use: a rule grammar is defined by an application, whereas a dictation grammar is defined by a recognizer and is built into the recognizer.

A rule grammar is provided by an application to a recognizer to define a set of rules that indicates what a user may say. Rules are defined by tokens, by references to other rules and by logical combinations of tokens and rule references. Rule grammars can be defined to capture a wide range of spoken input from users by the progressive combination of simple grammars and rules.

A dictation grammar is built into a recognizer. It defines a set of words (possibly tens of thousands of words) which may be spoken in a relatively unrestricted way. Dictation grammars are closest to the goal of unrestricted natural speech input to computers. Although dictation grammars are more flexible than rule grammars, recognition of rule grammars is typically faster and more accurate.

Support for a dictation grammar is optional for a recognizer As Section 4.2 (on page 36) explains, an application that requires dictation functionality can request it when creating a recognizer.

A recognizer may have many rule grammars loaded at any time. However, the current `Recognizer` interface restricts a recognizer to a single dictation grammar. The technical reasons for this restriction are outside the scope of this guide.

6.4.1 Grammar Interface

The `Grammar` interface is the root interface that is extended by all grammars. The grammar functionality that is shared by all grammars is presented through this interface.

The `RuleGrammar` interface is an extension of the `Grammar` interface to support rule grammars. The `DictationGrammar` interface is an extension of the `Grammar` interface to support dictation grammars.

The following are the capabilities presented by the grammar interface:

- *Grammar naming*: Every grammar loaded into a recognizer must have a unique name. The `getName` method returns that name. Grammar names allow references to be made between grammars. The grammar naming convention is described in the Java Speech Grammar Format Specification. Briefly, the grammar naming convention is very similar to the class naming convention for the Java programming language. For example, a grammar from Acme Corp. for dates might be called "`com.acme.speech.dates`".
- *Enabling and disabling*: Grammars may be enabled or disabled using the `setEnabled` method. When a grammar is enabled and when specified

activation conditions are met, the grammar is activated. Once a grammar is active a recognizer will listen to incoming audio for speech that matches that grammar. Enabling and activation are described in more detail below (Section 6.4.3).

- *Activation mode*: This is the property of a grammar that determines which conditions need to be met for a grammar to be activated. The activation mode is managed through the `getActivationMode` and `setActivationMode` methods (described in Section 6.4.3). The three available activation modes are defined as constants of the `Grammar` interface: `RECOGNIZER_FOCUS`, `RECOGNIZER_MODAL` and `GLOBAL`.
- *Activation*: the `isActive` method returns a `boolean` value that indicates whether a `Grammar` is currently active for recognition.
- *GrammarListener*: the `addGrammarListener` and `removeGrammarListener` methods allow a `GrammarListener` to be attached to and removed from a `Grammar`. The `GrammarEvents` issued to the listener indicate when grammar changes have been committed and whenever the grammar activation state changes.
- *ResultListener*: the `addResultListener` and `removeResultListener` methods allow a `ResultListener` to be attached to and removed from a `Grammar`. This listener receives notification of all events for any result that matches the grammar.
- *Recognizer*: the `getRecognizer` method returns a reference to the `Recognizer` that owns the `Grammar`.

6.4.2 Committing Changes

The Java Speech API supports *dynamic grammars*; that is, it supports the ability for an application to modify grammars at runtime. In the case of rule grammars any aspect of any grammar can be changed at any time.

After making any change to a grammar through the `Grammar`, `RuleGrammar` or `DictationGrammar` interfaces an application must *commit the changes*. This applies to changes in definitions of rules in a `RuleGrammar`, to changing context for a `DictationGrammar`, to changing the enabled state, or to changing the activation mode. (It does not apply to adding or removing a `GrammarListener` or `ResultListener`.)

Changes are committed by calling the `commitChanges` method of the `Recognizer`. The commit is required for changes to affect the recognition process: that is, the processing of incoming audio.

The commit changes mechanism has two important properties:

- ♦ Updates to grammar definitions and the enabled property take effect *atomically* (all changes take effect at once). There are no intermediate states in which some, but not all, changes have been applied.
- ♦ The `commitChanges` method is a method of `Recognizer` so all changes to all grammars are committed at once. Again, there are no intermediate states in which some, but not all, changes have been applied.

There is one instance in which changes are committed without an explicit call to the `commitChanges` method. Whenever a recognition result is *finalized* (completed), an event is issued to `ResultListeners` (it is either a `RESULT_ACCEPTED` or `RESULT_REJECTED` event). Once processing of that event is completed changes are normally committed. This supports the common situation in which changes are often made to grammars in response to something a user says.

The event-driven commit is closely linked to the underlying state system of a `Recognizer`. The state system for recognizers is described in detail in Section 6.3.

6.4.3 Grammar Activation

A grammar is *active* when the recognizer is matching incoming audio against that grammar to determine whether the user is saying anything that matches that grammar. When a grammar is inactive it is not being used in the recognition process.

Applications to do not directly activate and deactivate grammars. Instead they provided methods for (1) enabling and disabling a grammar, (2) setting the activation mode for each grammar, and (3) requesting and releasing the speech focus of a recognizer (as described in Section 6.3.2.)

The enabled state of a grammar is set with the `setEnabled` method and tested with the `isEnabled` method. For programmers familiar with AWT or Swing, enabling a speech grammar is similar to enabling a graphical component.

Once enabled, certain conditions must be met for a grammar to be activated. The activation mode indicates when an application wants the grammar to be active. There are three activation modes: `RECOGNIZER_FOCUS`, `RECOGNIZER_MODAL` and `GLOBAL`. For each mode a certain set of activation conditions must be met for the grammar to be activated for recognition. The activation mode is managed with the `setActivationMode` and `getActivationMode` methods.

The enabled flag and the activation mode are both parameters of a grammar that need to be committed to take effect. As Section 6.4.2 described, changes need to be committed to affect the recognition processes.

Recognizer focus is a major determining factor in grammar activation and is relevant in computing environments in which more than one application is using an underlying recognition (e.g., desktop computing with multiple speech-enabled applications). Section 6.3.2 (on page 76) describes how applications can request and release focus and monitor focus through `RecognizerEvents` and the engine state methods.

Recognizer focus is used to turn on and off activation of grammars. The roll of focus depends upon the activation mode. The three activation modes are described here in order from highest priority to lowest. An application should always use the lowest priority mode that is appropriate to its user interface functionality.

- `GLOBAL` activation mode: if enabled, the `Grammar` is always active irrespective of whether the `Recognizer` of this application has focus.
- `RECOGNIZER_MODAL` activation mode: if enabled, the `Grammar` is always active when the application's `Recognizer` has focus. Furthermore, enabling a modal grammar deactivates any grammars in the same `Recognizer` with the `RECOGNIZER_FOCUS` activation mode. (The term "modal" is analogous to "modal dialog boxes" in graphical programming.)
- `RECOGNIZER_FOCUS` activation mode (default mode): if enabled, the `Grammar` is active when the `Recognizer` of this application has focus. The exception is that if any other grammar of this application is enabled with `RECOGNIZER_MODAL` activation mode, then this grammar is not activated.

The current activation state of a grammar can be tested with the `isActive` method. Whenever a grammar's activation changes either a `GRAMMAR_ACTIVATED` or `GRAMMAR_DEACTIVATED` event is issued to each attached `GrammarListener`. A grammar activation event typically follows a `RecognizerEvent` that indicates a change in focus (`FOCUS_GAINED` or `FOCUS_LOST`), or a `CHANGES_COMMMITTED` `RecognizerEvent` that indicates that a change in the enabled setting of a grammar has been applied to the recognition process.

An application may have zero, one or many grammars enabled at any time. Thus, an application may have zero, one or many grammars active at any time. As the conventions below indicate, well-behaved applications always *minimize* the number of active grammars.

The activation and deactivation of grammars is independent of `PAUSED` and `RESUMED` states of the `Recognizer`. For instance, a grammar can be active even when a recognizer is `PAUSED`. However, when a `Recognizer` is paused, audio input to the `Recognizer` is turned off, so speech won't be detected. This is useful, however, because when the recognizer is resumed, recognition against the active grammars immediately (and automatically) resumes.

Activating too many grammars and, in particular, activating multiple complex grammars has an adverse impact upon a recognizer's performance. In general terms, increasing the number of active grammars and increasing the complexity of those grammars can both lead to slower recognition response time, greater CPU load and reduced recognition accuracy (i.e., more mistakes).

Well-behaved applications adhere to the following conventions to maximize recognition performance and minimize their impact upon other applications:

- Never apply the `GLOBAL` activation mode to a `DictationGrammar` (most recognizers will throw an exception if this is attempted).
- Always use the default activation mode `RECOGNIZER_FOCUS` unless there is a good reason to use another mode.
- Only use the `RECOGNIZER_MODAL` when it is certain that deactivating the `RECOGNIZER_FOCUS` grammars will not adversely affect the user interface.
- Minimize the complexity and the number of `RuleGrammars` with `GLOBAL` activation mode. As a general rule, one very simple `GLOBAL` rule grammar should be sufficient for nearly all applications.
- Only enable a grammar when it is appropriate for a user to say something matching that grammar. Otherwise disable the grammar to improve recognition response time and recognition accuracy for other grammars.
- Only request focus when confident that the user's speech focus (attention) is directed to grammars of your application. Release focus when it is not required.

6.5 Rule Grammars

6.5.1 Rule Definitions

A rule grammar is defined by a set of *rules*. These rules are defined by logical combinations of tokens to be spoken and references to other rules. The references may refer 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 Specification*). Any grammar defined in the JSGF can be converted to a `RuleGrammar` object. Any `RuleGrammar` object can be printed out in JSGF. (Note that conversion from JSGF to a `RuleGrammar` and back to JSGF will preserve the logic of the grammar but may lose comments and may change formatting.)

Since the `RuleGrammar` interface extends the `Grammar` interface, a `RuleGrammar` inherits the basic grammar functionality described in the previous sections (naming, enabling, activation etc.).

The easiest way to load a `RuleGrammar`, or set of `RuleGrammar` objects is from a Java Speech Grammar Format file or URL. The `loadJSGF` methods of the `Recognizer` perform this task. If multiple grammars must be loaded (where a grammar references one or more imported grammars), importing by URL is most convenient. The application must specify the base URL and the name of the root grammar to be loaded.

```
Recognizer rec;
URL base = new URL("http://www.acme.com/app”);
String grammarName = "com.acme.demo";

Grammar gram = rec.loadURL(base, grammarName);
```

The recognizer converts the base URL and grammar name to a URL using the same conventions as `ClassLoader` (the Java platform mechanism for loading class files). By converting the periods in the grammar name to slashes ('/'), appending a `".gram"` suffix and combining with the base URL, the location is “`http://www.acme.com/app/com/acme/demo.gram`”.

If the demo grammar imports sub-grammars, they will be loaded automatically using the same location mechanism.

Alternatively, a `RuleGrammar` can be created by calling the `newRuleGrammar` method of a `Recognizer`. This method creates an empty grammar with a specified grammar name.

Once a `RuleGrammar` has been loaded, or has been created with the `newRuleGrammar` method, the following methods of a `RuleGrammar` are used to create, modify and manage the rules of the grammar.

Table 6-1 RuleGrammar methods for Rule management

Name	Description
`setRule`	Assign a `Rule` object to a rulename.
`getRule`	Return the `Rule` object for a rulename.

Table 6-1 RuleGrammar methods for Rule management (cont'd)

Name	Description
`getRuleInternal`	Return a reference to the recognizer's internal `Rule` object for a rulename (for fast, read-only access).
`listRuleNames`	List known rulenames.
`isRulePublic`	Test whether a rulename is public.
`deleteRule`	Delete a rule.
`setEnabled`	Enable and disable this `RuleGrammar` or rules of the grammar.
`isEnabled`	Test whether a `RuleGrammar` or a specified rule is enabled.

Any of the methods of `RuleGrammar` that affect the grammar (`setRule`, `deleteRule`, `setEnabled` etc.) take effect only after they are committed (as described in Section 6.4.2).

The rule definitions of a `RuleGrammar` can be considered as a collection of named `Rule` objects. Each `Rule` object is referenced by its rulename (a `String`). The different types of `Rule` object are described in Section 6.5.3.

Unlike most collections in Java, the `RuleGrammar` is a collection that does not share objects with the application. This is because recognizers often need to perform special processing of the rule objects and store additional information internally. The implication for applications is that a call to `setRule` is required to change any rule. The following code shows an example where changing a rule object does not affect the grammar.

```
RuleGrammar gram;

// Create a rule for the word blue
// Add the rule to the RuleGrammar and make it public
RuleToken word = new RuleToken("blue");
gram.setRule("ruleName", word, true);

// Change the word
word.setText("green");
```

```
// getRule returns blue (not green)
System.out.println(gram.getRule("ruleName"));
```

To ensure that the changed `"green"` token is loaded into the grammar, the application must call `setRule` again after changing the word to `"green"`. Furthermore, for either change to take effect in the recognition process, the changes need to be committed (see Section 6.4.2).

6.5.2 Imports

Complex systems of rules are most easily built by dividing the rules into multiple grammars. For example, a grammar could be developed for recognizing numbers. That grammar could then be *imported* into two separate grammars that defines dates and currency amounts. Those two grammars could then be imported into a travel booking application and so on. This type of hierarchical grammar construction is similar in many respects to object oriented and shares the advantage of easy reusage of grammars.

An import declaration in JSGF and an import in a `RuleGrammar` are most similar to the import statement of the Java programming language. Unlike a "#include" in the C programming language, the imported grammar is not copied, it is simply referencable. (A full specification of import semantics is provided in the Java Speech Grammar Format specification.)

The `RuleGrammar` interface defines three methods for handling imports as shown in Table 6-2.

Table 6-2 RuleGrammar import methods

Name	Description
`addImport`	Add a grammar or rule for import.
`removeImport`	Remove the import of a rule or grammar.
`getImports`	Return a list of all imported grammars or all rules imported from a specific grammar.

The resolve method of the RuleGrammar interface is useful in managing imports. Given any rulename, the resolve method returns an object that represents the fully-qualified rulename for the rule that it references.

6.5.3 Rule Classes

A RuleGrammar is primarily a collection of defined rules. The programmatic rule structure used to control Recognizers follows exactly the definition of rules in the Java Speech Grammar Format. Any rule is defined by a Rule object. It may be any one of the Rule classes described Table 6-3. The exceptions are the RuleParse class, which is returned by the parse method of RuleGrammar, and the Rule class which is an abstract class and the parent of all other Rule objects.

Table 6-3 Rule objects

Name	Description
Rule	Abstract root object for rules.
RuleName	Rule that references another defined rule. JSGF example: <ruleName>
RuleToken	Rule consisting of a single speakable token (e.g. a word). JSGF examples: elephant, "New York"
RuleSequence	Rule consisting of a sequence of sub-rules. JSGF example: buy <number> shares of <company>
RuleAlternatives	Rule consisting of a set of alternative sub-rules. JSGF example: green | red | yellow
RuleCount	Rule containing a sub-rule that may be spoken optionally, zero or more times, or one or more times. JSGF examples: <color>*, [optional]
RuleTag	Rule that attaches a tag to a sub-rule. JSGF example: {action=open}
RuleParse	Special rule object used to represent results of a parse.

The following is an example of a grammar in Java Speech Grammar Format. The *“Hello World!”* example (page 72) shows how this JSGF grammar can be loaded from a text file. Below we consider how to create the same grammar programmatically.

```
grammar com.sun.speech.test;

public <test> = [a] test {TAG} | another <rule>;
<rule> = word;
```

The following code shows the simplest way to create this grammar. It uses the `ruleForJSGF` method to convert partial JSGF text to a `Rule` object. Partial JSGF is defined as any legal JSGF text that may appear on the right hand side of a rule definition — technically speaking, any legal JSGF rule expansion.

```
Recognizer rec;

// Create a new grammar
RuleGrammar gram = rec.newRuleGrammar("com.sun.speech.test");

// Create the <test> rule
Rule test = gram.ruleForJSGF("[a] test {TAG} | another <rule>");
gram.setRule("test", // rulename
      test, // rule definition
      true); // true -> make it public

// Create the <rule> rule
gram.setRule("rule", gram.ruleForJSGF("word”), false);

// Commit the grammar
rec.commitChanges();
```

6.5.3.1 Advanced Rule Programming

In advanced programs there is often a need to define rules using the set of `Rule` objects described above. For these applications, using rule objects is more efficient than creating a JSGF string and using the `ruleForJSGF` method.

To create a rule by code, the detailed structure of the rule needs to be understood. At the top level of our example grammar, the `<test>` rule is an alternative: the user may say something that matches `"[a] test {TAG}"` or say something matching `"another <rule>"`. The two alternatives are each sequences containing two items. In the first alternative, the brackets around the token `"a"` indicate it is optional. The `"{TAG}"` following the second token (`"test"`) attaches a tag to the token. The second alternative is a sequence with a token (`"another"`) and a reference to another rule (`"<rule>"`).

The code to construct this `Grammar` follows (this code example is not compact — it is written for clarity of details).

```
Recognizer rec;

RuleGrammar gram = rec.newRuleGrammar("com.sun.speech.test");

// Rule we are building
RuleAlternatives test;

// Temporary rules
RuleCount r1;
RuleTag r2;
RuleSequence seq1, seq2;

// Create "[a]"
r1 = new RuleCount(new RuleToken("a"), RuleCount.OPTIONAL);

// Create "test {TAG}" - a tagged token
r2 = new RuleTag(new RuleToken("test"), "TAG");

// Join "[a]" and "test {TAG}" into a sequence "[a] test {TAG}"
seq1 = new RuleSequence(r1);
seq1.append(r2);

// Create the sequence "another <rule>";
seq2 = new RuleSequence(new RuleToken("another"));
seq2.append(new RuleName("rule"));

// Build "[a] test {TAG} | another <rule>"
test = new RuleAlternatives(seq1);
test.append(seq2);

// Add <test> to the RuleGrammar as a public rule
gram.setRule("test", test, true);

// Provide the definition of <rule>, a non-public RuleToken
gram.setRule("rule", new RuleToken("word"), false);

// Commit the grammar changes
rec.commitChanges();
```

6.5.4 Dynamic Grammars

Grammars may be modified and updated. The changes allow an application to account for shifts in the application's context, changes in the data available to it, and so on. This flexibility allows application developers considerable freedom in creating dynamic and natural speech interfaces.

For example, in an email application the list of known users may change during the normal operation of the program. The `<sendEmail>` command,

```
<sendEmail> = send email to <user>;
```

references the `<user>` rule which may need to be changed as new email arrives. This code snippet shows the update and commit of a change in users.

```
Recognizer rec;
RuleGrammar gram;

String names[] = {"amy", "alan", "paul"};
Rule userRule = new RuleAlternatives(names);

gram.setRule("user", userRule);

// apply the changes
rec.commitChanges();
```

Committing grammar changes can, in certain cases, be a slow process. It might take a few tenths of seconds or up to several seconds. The time to commit changes depends on a number of factors. First, recognizers have different mechanisms for committing changes making some recognizers faster than others. Second, the time to commit changes may depend on the extent of the changes — more changes may require more time to commit. Thirdly, the time to commit may depend upon the type of changes. For example, some recognizers optimize for changes to lists of tokens (e.g. name lists). Finally, faster computers make changes more quickly.

The other factor which influences dynamic changes is the timing of the commit. As Section 6.4.2 describes, grammar changes are not always committed instantaneously. For example, if the recognizer is busy recognizing speech (in the `PROCESSING` state), then the commit of changes is deferred until the recognition of that speech is completed.

6.5.5 Parsing

Parsing is the process of matching text to a grammar. Applications use parsing to break down spoken input into a form that is more easily handled in software. Parsing is most useful when the structure of the grammars clearly separates the parts of spoken text that an application needs to process. Examples are given below of this type of structuring.

The text may be in the form of a `String` or array of `String` objects (one `String` per token), or in the form of a `FinalRuleResult` object that represents what a recognizer heard a user say. The `RuleGrammar` interface defines three forms of the `parse` method — one for each form of text.

The `parse` method returns a `RuleParse` object (a descendent of `Rule`) that represents how the text matches the `RuleGrammar`. The structure of the `RuleParse` object mirrors the structure of rules defined in the `RuleGrammar`. Each `Rule` object in the structure of the rule being parsed against is mirrored by a matching `Rule` object in the returned `RuleParse` object.

The difference between the structures comes about because the text being parsed defines a single phrase that a user has spoken whereas a `RuleGrammar` defines all the phrases the user could say. Thus the text defines a single path through the grammar and all the choices in the grammar (alternatives, and rules that occur optionally or occur zero or more times) are resolvable.

The mapping between the objects in the rules defined in the `RuleGrammar` and the objects in the `RuleParse` structure is shown in Table 6-4. Note that except for the `RuleCount` and `RuleName` objects, the object in the parse tree are of the same type as rule object being parsed against (marked with "**"), but the internal data may differ.

Table 6-4 Matching Rule definitions and RuleParse objects

Object in definition	Matching object in RuleParse
`RuleToken`	Maps to an identical `RuleToken` object.
`RuleTag`	Maps to a `RuleTag` object with the same tag and with the contained rule mapped according to its rule type.
`RuleSequence`	Maps to a `RuleSequence` object with identical length and with each rule in the sequence mapped according to its rule type.

Table 6-4 Matching Rule definitions and RuleParse objects (cont'd)

Object in definition	Matching object in RuleParse
`RuleAlternatives`	Maps to a `RuleAlternatives` object containing a single item which is the one rule in the set of alternatives that was spoken.
`RuleCount **`	Maps to a `RuleSequence` object containing an item for each time the rule contained by the `RuleCount` object is spoken. The sequence may have a length of zero, one or more.
`RuleName **`	Maps to a `RuleParse` object with the name in the `RuleName` object being the fully-qualified version of the original rulename, and with the `Rule` object contained by the `RuleParse` object being an appropriate match of the definition of `RuleName`.

As an example, take the following simple extract from a grammar. The public rule, `<command>`, may be spoken in many ways. For example, "open", "move that door" or "close that door please".

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

Note how the rules are defined to clearly separate the segments of spoken input that an application must process. Specifically, the `<action>` and `<object>` rules indicate how an application must respond to a command. Furthermore, anything said that matches the `<polite>` rule can be safely ignored, and usually the `<this_that_etc>` rule can be ignored too.

The parse for "open" against `<command>` has the following structure which matches the structure of the grammar above.

```
RuleParse(<command> =
  RuleSequence(
```

```
        RuleParse(<action> =
            RuleAlternatives(
                RuleTag(
                    RuleToken("open"), "OP")))))
```

The match of the `<command>` rule is represented by a `RuleParse` object. Because the definition of `<command>` is a sequence of 3 items (2 of which are optional), the parse of `<command>` is a sequence. Because only one of the 3 items is spoken (in "open"), the sequence contains a single item. That item is the parse of the `<action>` rule.

The reference to `<action>` in the definition of `<command>` is represented by a `RuleName` object in the grammar definition, and this maps to a `RuleParse` object when parsed. The `<action>` rule is defined by a set of three alternatives (`RuleAlternatives` object) which maps to another `RuleAlternatives` object in the parse but with only the single spoken alternative represented. Since the phrase spoken was "open", the parse matches the first of the three alternatives which is a tagged token. Therefore the parse includes a `RuleTag` object which contains a `RuleToken` object for "open".

The following is the parse for "close that door please".

```
RuleParse(<command> =
   RuleSequence(
       RuleParse(<action> =
           RuleAlternatives(
               RuleTag(
                   RuleToken("close"), "CL")))
       RuleSequence(
           RuleParse(<object> =
               RuleSequence(
                   RuleSequence(
                       RuleParse(<this_that_etc> =
                           RuleAlternatives(
                               RuleToken("that"))))
                   RuleAlternatives(
                       RuleToken("door"))))
       RuleSequence(
           RuleParse(<polite> =
               RuleAlternatives(
                   RuleToken("please"))))
   ))
```

There are three parsing issues that application developers should consider.

- Parsing may fail because there is no legal match. In this instance the `parse` methods return `null`.
- There may be several legal ways to parse the text against the grammar. This is known as an *ambiguous* parse. In this instance the `parse` method will return one of the legal parses but the application is not informed of the ambiguity. As a general rule, most developers will want to avoid ambiguous parses by proper grammar design. Advanced applications will use specialized parsers if they need to handle ambiguity.
- If a `FinalRuleResult` is parsed against the `RuleGrammar` and the rule within that grammar that it matched, then it should successfully parse. However, it is not guaranteed to parse if the `RuleGrammar` has been modified of if the `FinalRuleResult` is a `REJECTED` result. (Result rejection is described in Section 6.7.)

6.6 Dictation Grammars

Dictation grammars come closest to the ultimate goal of a speech recognition system that takes natural spoken input and transcribes it as text. Dictation grammars are used for free text entry in applications such as email and word processing.

A `Recognizer` that supports dictation provides a single `DictationGrammar` which is obtained from the recognizer's `getDictationGrammar` method. A recognizer that supports the Java Speech API is not required to provide a `DictationGrammar`. Applications that require a recognizer with dictation capability can explicitly request dictation when creating a recognizer by setting the `DictationGrammarSupported` property of the `RecognizerModeDesc` to true (see Section 4.2 for details).

A `DictationGrammar` is more complex than a rule grammar, but fortunately, a `DictationGrammar` is often easier to use than an rule grammar. This is because the `DictationGrammar` is built into the recognizer so most of the complexity is handled by the recognizer and hidden from the application. However, recognition of a dictation grammar is typically more computationally expensive and less accurate than that of simple rule grammars.

The `DictationGrammar` inherits its basic functionality from the `Grammar` interface. That functionality is detailed in Section 6.4 and includes grammar naming, enabling, activation, committing and so on.

As with all grammars, changes to a `DictationGrammar` need to be committed before they take effect. Commits are described in Section 6.4.2.

In addition to the specific functionality described below, a `DictationGrammar` is typically adaptive. In an adaptive system, a recognizer improves its performance (accuracy and possibly speed) by adapting to the style of language used by a speaker. The recognizer may adapt to the specific sounds of a speaker (the way they say words). Equally importantly for dictation, a recognizer can adapt to a user's normal vocabulary and to the patterns of those words. Such adaptation (technically known as language model adaptation) is a part of the recognizer's implementation of the `DictationGrammar` and does not affect an application. The adaptation data for a dictation grammar is maintained as part of a speaker profile (see Section 6.9).

The `DictationGrammar` extends and specializes the `Grammar` interface by adding the following functionality:

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

The following methods provided by the DictationGrammar interface allow an application to manage word lists and text context.

Table 6-5 DictationGrammar interface methods

Name	Description
`setContext`	Provide the recognition engine with the preceding and following textual context.
`addWord`	Add a word to the `DictationGrammar`.
`removeWord`	Remove a word from the `DictationGrammar`.
`listAddedWords`	List the words that have been added to the `DictationGrammar`.
`listRemovedWords`	List the words that have been removed from the `DictationGrammar`.

6.6.1 Dictation Context

Dictation recognizers use a range of information to improve recognition accuracy. Learning the words a user speaks and the patterns of those words can substantially improve accuracy.

Because patterns of words are important, *context* is important. The context of a word is simply the set of surrounding words. As an example, consider the following sentence *"If I have seen further it is by standing on the shoulders of Giants"* (Sir Isaac Newton). If we are editing this sentence and place the cursor after the word "*standing*" then the preceding context is *"...further it is by standing"* and the following context is *"on the shoulders of Giants..."*.

Given this context, the recognizer is able to more reliably predict what a user might say, and greater predictability can improve recognition accuracy. In this example, the user might insert the word *"up"* but is less likely to insert the word *"JavaBeans"*.

Through the `setContext` method of the `DictationGrammar` interface, an application should tell the recognizer the current textual context. Furthermore, if the context changes (for example, due to a mouse click to move the cursor) the application should update the context.

Different recognizers process context differently. The main consideration for the application is the amount of context to provide to the recognizer. As a minimum, a few words of preceding and following context should be provided. However, some recognizers may take advantage of several paragraphs or more.

There are two `setContext` methods:

```
void setContext(String preceding, String following);
void setContext(String preceding[], String following[]);
```

The first form takes plain text context strings. The second version should be used when the result tokens returned by the recognizer are available. Internally, the recognizer processes context according to tokens so providing tokens makes the use of context more efficient and more reliable because it does not have to guess the tokenization.

6.7 Recognition Results

A recognition *result* is provided by a `Recognizer` to an application when the recognizer "hears" incoming speech that matches an active grammar. The result tells the application what words the user said and provides a range of other useful information, including alternative guesses and audio data.

In this section, both the basic and advanced capabilities of the result system in the Java Speech API are described. The sections relevant to basic rule grammar-based applications are those that cover result finalization (Section 6.7.1, page 102), the hierarchy of result interfaces (Section 6.7.2, page 104), the data provided through those interfaces (Section 6.7.3, page 106), and common techniques for handling finalized rule results (Section 6.7.9, page 114).

For dictation applications the relevant sections include those listed above plus the sections covering token finalization (Section 6.7.8, page 112), handling of finalized dictation results (Section 6.7.10, page 119) and result correction and training (Section 6.7.12, page 127).

For more advanced applications relevant sections might include the result life cycle (Section 6.7.4, page 108), attachment of ResultListeners (Section 6.7.5, page 109), the relationship of recognizer and result states (Section 6.7.6, page 110), grammar finalization (Section 6.7.7, page 111), result audio (Section 6.7.11, page 125), rejected results (Section 6.7.13, page 129), result timing (Section 6.7.14, page 131), and the loading and storing of vendor formatted results (Section 6.7.15, page 132).

6.7.1 Result Finalization

The *"Hello World!"* example (on page 72) illustrates the simplest way to handle results. In that example, a `RuleGrammar` was loaded, committed and enabled, and a `ResultListener` was attached to a `Recognizer` to receive events associated with every result that matched that grammar. In other words, the `ResultListener` was attached to receive information about words spoken by a user that is heard by the recognizer.

The following is a modified extract of the *"Hello World!"* example to illustrate the basics of handling results. In this case, a `ResultListener` is attached to a `Grammar` (instead of a `Recognizer`) and it prints out every thing the recognizer hears that matches that grammar. (There are, in fact, three ways in which a `ResultListener` can be attached: see Section 6.7.5 on page 109.)

```
import javax.speech.*;
import javax.speech.recognition.*;

public class MyResultListener extends ResultAdapter {
   // Receives RESULT_ACCEPTED event: print it
   public void resultAccepted(ResultEvent e) {
      Result r = (Result)(e.getSource());
      ResultToken tokens[] = r.getBestTokens();

      for (int i = 0; i < tokens.length; i++)
         System.out.print(tokens[i].getSpokenText() + " ");
      System.out.println();
   }

   // somewhere in app, add a ResultListener to a grammar
```

```
    {
        RuleGrammar gram = ...;
        gram.addResultListener(new MyResultListener());
    }
}
```

The code shows the `MyResultListener` class which is as an extension of the `ResultAdapter` class. The `ResultAdapter` class is a convenience implementation of the `ResultListener` interface (provided in the `javax.speech.recognition` package). When extending the `ResultAdapter` class we simply implement the methods for the events that we care about.

In this case, the `RESULT_ACCEPTED` event is handled. This event is issued to the `resultAccepted` method of the `ResultListener` and is issued when a result is *finalized*. Finalization of a result occurs after a recognizer completed processing of a result. More specifically, finalization occurs when all information about a result has been produced by the recognizer and when the recognizer can guarantee that the information will not change. (Result finalization should not be confused with object finalization in the Java programming language in which objects are cleaned up before garbage collection.)

There are actually two ways to finalize a result which are signalled by the `RESULT_ACCEPTED` and `RESULT_REJECTED` events. A result is accepted when a recognizer is confidently that it has correctly heard the words spoken by a user (i.e., the tokens in the `Result` exactly represent what a user said).

Rejection occurs when a `Recognizer` is not confident that it has correctly recognized a result: that is, the tokens and other information in the result do not necessarily match what a user said. Many applications will ignore the `RESULT_REJECTED` event and most will ignore the detail of a result when it is rejected. In some applications, a `RESULT_REJECTED` event is used simply to provide users with feedback that something was heard but no action was taken, for example, by displaying "???" or sounding an error beep. Rejected results and the differences between accepted and rejected results are described in more detail in Section 6.7.13 (on page 129).

An accepted result is not necessarily a correct result. As is pointed out in Section 2.2.3 (on page 16), recognizers make errors when recognizing speech for a range of reasons. The implication is that even for an accepted result, application developers should consider the potential impact of a misrecognition. Where a misrecognition could cause an action with serious consequences or could make changes that can't be undone (e.g., "delete all files"), the application should check with users before performing the action. As recognition systems continue to improve the number of errors is steadily decreasing, but as with human speech recognition there will always be a chance of a misunderstanding.

6.7.2 Result Interface Hierarchy

A finalized result can include a considerable amount of information. This information is provided through four separate interfaces and through the implementation of these interfaces by a recognition system.

```
// Result: the root result interface
interface Result;

// FinalResult: info on all finalized results
interface FinalResult extends Result;

// FinalRuleResult: a finalized result matching a RuleGrammar
interface FinalRuleResult extends FinalResult;

// FinalDictationResult: a final result for a DictationGrammar
interface FinalDictationResult extends FinalResult;

// A result implementation provided by a Recognizer
public class EngineResult
         implements FinalRuleResult, FinalDictationResult;
```

At first sight, the result interfaces may seem complex. The reasons for providing several interfaces are as follows:

- The information available for a result is different in different states of the result. Before finalization, a limited amount of information is available through the `Result` interface. Once a result is finalized (accepted or rejected), more detailed information is available through the `FinalResult` interface and either the `FinalRuleResult` or `FinalDictationResult` interface.
- The type of information available for a finalized result is different for a result that matches a `RuleGrammar` than for a result that matches a `DictationGrammar`. The differences are explicitly represented by having separate interfaces for `FinalRuleResult` and `FinalDictationResult`.
- Once a result object is created as a specific Java class it cannot change be changed to another class. Therefore, because a result object must eventually support the final interface it must implement them when first created. Therefore, every result implements all three final interfaces when it is first created: `FinalResult`, `FinalRuleResult` and `FinalDictationResult`.
- When a result is first created a recognizer does not always know whether it

will eventually match a `RuleGrammar` or a `DictationGrammar`. Therefore, every result object implements both the `FinalRuleResult` and `FinalDictationResult` interfaces.

- A call made to any method of any of the final interfaces before a result is finalized causes a `ResultStateException`.
- A call made to any method of the `FinalRuleResult` interface for a result that matches a `DictationGrammar` causes a `ResultStateException`. Similarly, a call made to any method of the `FinalDictationResult` interface for a result that matches a `RuleGrammar` causes a `ResultStateException`.
- All the result functionality is provided by interfaces in the `java.speech.recognition` package rather than by classes. This is because the Java Speech API can support multiple recognizers from multiple vendors and interfaces allow the vendors greater flexibility in implementing results.

The multitude of interfaces is, in fact, designed to simplify application programming and to minimize the chance of introducing bugs into code by allowing compile-time checking of result calls. The two basic principles for calling the result interfaces are the following:

1. If it is safe to call the methods of a particular interface then it is safe to call the methods of any of the parent interfaces. For example, for a finalized result matching a `RuleGrammar`, the methods of the `FinalRuleResult` interface are safe, so the methods of the `FinalResult` and `Result` interfaces are also safe. Similarly, for a finalized result matching a `DictationGrammar`, the methods of `FinalDictationResult`, `FinalResult` and `Result` can all be called safely.
2. Use type casting of a result object to ensure compile-time checks of method calls. For example, in events to an unfinalized result, cast the result object to the `Result` interface. For a `RESULT_ACCEPTED` finalization event with a result that matches a `DictationGrammar`, cast the result to the `FinalDictationResult` interface.

In the next section the different information available through the different interfaces is described. In all the following sections that deal with result states and result events, details are provided on the appropriate casting of result objects.

6.7.3 Result Information

As the previous section describes, different information is available for a result depending upon the state of the result and, for finalized results, depending upon the type of grammar it matches (`RuleGrammar` or `DictationGrammar`).

6.7.3.1 Result Interface

The information available through the `Result` interface is available for any result in any state — finalized or unfinalized — and matching any grammar.

- *Result state*: The `getResultState` method returns the current state of the result. The three possible state values defined by static values of the `Result` interface are `UNFINALIZED`, `ACCEPTED` and `REJECTED`. (Result states are described in more detail in Section 6.7.4.)
- *Grammar*: The `getGrammar` method returns a reference to the matched `Grammar`, if it is known. For an `ACCEPTED` result, this method will return a `RuleGrammar` or a `DictationGrammar`. For a `REJECTED` result, this method may return a grammar, or may return `null` if the recognizer could not identify the grammar for this result. In the `UNFINALIZED` state, this method returns `null` before a `GRAMMAR_FINALIZED` event, and non-null afterwards.
- *Number of finalized tokens*: The `numTokens` method returns the total number of finalized tokens for a result. For an unfinalized result this may be zero or greater. For a finalized result this number is always greater than zero for an `ACCEPTED` result but may be zero or more for a `REJECTED` result. Once a result is finalized this number will not change.
- *Finalized tokens*: The `getBestToken` and `getBestTokens` methods return either a specified finalized best-guess token of a result or all the finalized best-guess tokens. The `ResultToken` object and token finalization are described in the following sections.
- *Unfinalized tokens*: In the `UNFINALIZED` state, the `getUnfinalizedTokens` method returns a list of unfinalized tokens. An unfinalized token is a recognizer's current guess of what a user has said, but the recognizer may choose to change these tokens at any time and any way. For a finalized result, the `getUnfinalizedTokens` method always returns `null`.

In addition to the information detailed above, the `Result` interface provides the `addResultListener` and `removeResultListener` methods which allow a `ResultListener` to be attached to and removed from an individual result.

`ResultListener` attachment is described in more detail in Section 6.7.5 (on page 109).

6.7.3.2 FinalResult Interface

The information available through the `FinalResult` interface is available for any finalized result, including results that match either a `RuleGrammar` or `DictationGrammar`.

- *Audio data*: a `Recognizer` may optionally provide audio data for a finalized result. This data is provided as `AudioClip` for a token, a sequence of tokens, or for the entire result. Result audio and its management are described in more detail in Section 6.7.11 (on page 125).
- *Training data*: many recognizer's have the ability to be trained and corrected. By training a recognizer or correcting its mistakes, a recognizer can adapt its recognition processes so that performance (accuracy and speed) improve over time. Several methods of the FinalResult interface support this capability and are described in detail in Section 6.7.12 (on page 127).

6.7.3.3 FinalDictationResult Interface

The `FinalDictationResult` interface contains a single method.

- *Alternative tokens*: The `getAlternativeTokens` method allows an application to request a set of alternative guesses for a single token or for a sequence of tokens in that result. In dictation systems, alternative guesses are typically used to facilitate correction of dictated text. Dictation recognizers are designed so that when they do make a misrecognition, the correct word sequence is usually amongst the best few alternative guesses. Section 6.7.10 (on page 119) explains alternatives in more detail.

6.7.3.4 FinalRuleResult Interface

Like the `FinalDictationResult` interface, the `FinalRuleResult` interface provides alternative guesses. The `FinalRuleResult` interface also provides some additional information that is useful in processing results that match a `RuleGrammar`.

- *Alternative tokens*: The `getAlternativeTokens` method allows an application to request a set of alternative guesses for the entire result (not for tokens). The `getNumberGuesses` method returns the actual number of alternative guesses available.

- *Alternative grammars*: The alternative guesses of a result matching a `RuleGrammar` do not all necessarily match the same grammar. The `getRuleGrammar` method returns a reference to the `RuleGrammar` matched by an alternative.
- *Rulenames*: When a result matches a `RuleGrammar`, it matches a specific defined rule of that `RuleGrammar`. The `getRuleName` method returns the rulename for the matched rule. Section 6.7.9 (on page 114) explains how this additional information is useful in processing `RuleGrammar` results.
- *Tags*: A tag is a string attached to a component of a `RuleGrammar` definition. Tags are useful in simplifying the software for processing results matching a `RuleGrammar` (explained in Section 6.7.9). The `getTags` method returns the tags for the best guess for a `FinalRuleResult`.

6.7.4 Result Life Cycle

A `Result` is produced in response to a user's speech. Unlike keyboard input, mouse input and most other forms of user input, speech is not instantaneous (see Section 6.3.3.1 for more detail). As a consequence, a speech recognition result is not produced instantaneously. Instead, a `Result` is produced through a sequence of events starting some time after a user starts speaking and usually finishing some time after the user stops speaking.

Figure 6-2 shows the state system of a `Result` and the associated `ResultEvents`. As in the recognizer state diagram (Figure 6-1), the blocks represent states, and the labelled arcs represent transitions that are signalled by `ResultEvents`.

Every result starts in the `UNFINALIZED` state when a `RESULT_CREATED` event is issued. While unfinalized, the recognizer provides information including finalized and unfinalized tokens and the identity of the grammar matched by the result. As this information is added, the `RESULT_UPDATED` and `GRAMMAR_FINALIZED` events are issued

Once all information associated with a result is finalized, the entire result is finalized. As Section 6.7.1 explained, a result is finalized with either a `RESULT_ACCEPTED` or `RESULT_REJECTED` event placing it in either the `ACCEPTED` or `REJECTED` state. At that point all information associated with the result becomes available including the best guess tokens and the information provided through the three final result interfaces (see Section 6.7.3).

Once finalized the information available through all the result interfaces is fixed. The only exceptions are for the release of audio data and training data. If

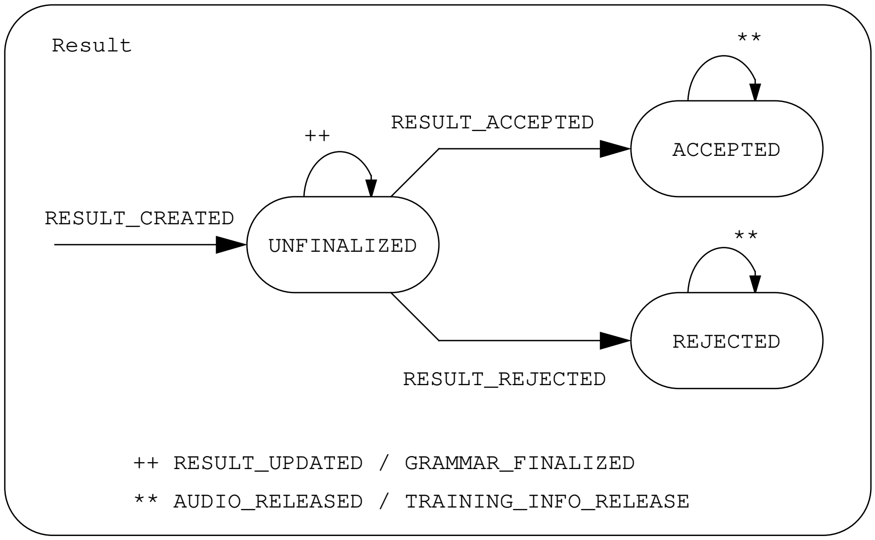

Figure 6-2 Result states

audio data is released, an `AUDIO_RELEASED` event is issued (see detail in Section 6.7.11). If training information is released, an `TRAINING_INFO_RELEASED` event is issued (see detail in Section 6.7.12).

Applications can track result states in a number of ways. Most often, applications handle result in `ResultListener` implementation which receives `ResultEvents` as recognition proceeds.

As Section 6.7.3 explains, a recognizer conveys a range of information to an application through the stages of producing a recognition result. However, as the example in Section 6.7.1 shows, many applications only care about the last step and event in that process — the `RESULT_ACCEPTED` event.

The state of a result is also available through the `getResultState` method of the `Result` interface. That method returns one of the three result states: `UNFINALIZED`, `ACCEPTED` or `REJECTED`.

6.7.5 ResultListener Attachment

A `ResultListener` can be attached in one of three places to receive events associated with results: to a `Grammar`, to a `Recognizer` or to an individual `Result`. The different places of attachment give an application some flexibility in how they handle results.

To support ResultListeners the Grammar, Recognizer and Result interfaces all provide the addResultListener and removeResultListener methods.

Depending upon the place of attachment a listener receives events for different results and different subsets of result events.

- Grammar: A ResultListener attached to a Grammar receives all ResultEvents for any result that has been finalized to match that grammar. Because the grammar is known once a GRAMMAR_FINALIZED event is produced, a ResultListener attached to a Grammar receives that event and subsequent events. Since grammars are usually defined for specific functionality it is common for most result handling to be done in the methods of listeners attached to each grammar.
- Result: A ResultListener attached to a Result receives all ResultEvents starting at the time at which the listener is attached to the Result. Note that because a listener cannot be attached until a result has been created with the RESULT_CREATED event, it can never receive that event.
- Recognizer: A ResultListener attached to a Recognizer receives all ResultEvents for all results produced by that Recognizer for all grammars. This form of listener attachment is useful for very simple applications (e.g., *"Hello World!"*) and when centralized processing of results is required. Only ResultListeners attached to a Recognizer receive the RESULT_CREATED event.

6.7.6 Recognizer and Result States

The state system of a recognizer is tied to the processing of a result. Specifically, the LISTENING, PROCESSING and SUSPENDED state cycle described in Section 6.3.3 (on page 79) and shown in Figure 6-1 (on page 77) follows the production of an event.

The transition of a Recognizer from the LISTENING state to the PROCESSING state with a RECOGNIZER_PROCESSING event indicates that a recognizer has started to produce a result. The RECOGNIZER_PROCESSING event is followed by the RESULT_CREATED event to ResultListeners.

The RESULT_UPDATED and GRAMMAR_FINALIZED events are issued to ResultListeners while the recognizer is in the PROCESSING state.

As soon as the recognizer completes recognition of a result, it makes a transition from the PROCESSING state to the SUSPENDED state with a RECOGNIZER_SUSPENDED event. Immediately following that recognizer event, the result finalization event (either RESULT_ACCEPTED or RESULT_REJECTED) is issued. While the result finalization event is processed, the recognizer remains suspended. Once result finalization event is completed, the recognizer automatically transitions from the SUSPENDED state back to the LISTENING state with a CHANGES_COMMITTED event. Once back in the LISTENING state the recognizer resumes processing of audio input with the grammar committed with the CHANGES_COMMITTED event.

6.7.6.1 Updating Grammars

In many applications, grammar definitions and grammar activation need to be updated in response to spoken input from a user. For example, if speech is added to a traditional email application, the command “save this message” might result in a window being opened in which a mail folder can be selected. While that window is open, the grammars that control that window need to be activated. Thus during the event processing for the “save this message” command grammars may need be created, updated and enabled. All this would happen during processing of the RESULT_ACCEPTED event.

For any grammar changes to take effect they must be committed (see Section 6.4.2 on page 85). Because this form of grammar update is so common while processing the RESULT_ACCEPTED event (and sometimes the RESULT_REJECTED event), recognizers implicitly commit grammar changes after either result finalization event has been processed.

This implicit is indicated by the CHANGES_COMMITTED event that is issued when a Recognizer makes a transition from the SUSPENDED state to the LISTENING state following result finalization and the result finalization event processing (see Section 6.3.3 for details).

One desirable effect of this form of commit becomes useful in component systems. If changes in multiple components are triggered by a finalized result event, and if many of those components change grammars, then they do not each need to call the commitChanges method. The downside of multiple calls to the commitChanges method is that a syntax check be performed upon each. Checking syntax can be computationally expensive and so multiple checks are undesirable. With the implicit commit once all components have updated grammars computational costs are reduced.

6.7.7 Grammar Finalization

At any time during processing a result a GRAMMAR_FINALIZED event can be issued for that result indicating the Grammar matched by the result has been determined. This event is issued is issued only once. It is required for any ACCEPTED result, but is optional for result that is eventually rejected.

As Section 6.7.5 describes, the GRAMMAR_FINALIZED event is the first event received by a ResultListener attached to a Grammar.

The `GRAMMAR_FINALIZED` event behaves the same for results that match either a `RuleGrammar` or a `DictationGrammar`.

Following the `GRAMMAR_FINALIZED` event, the `getGrammar` method of the `Result` interface returns a non-null reference to the matched grammar. By issuing a `GRAMMAR_FINALIZED` event the `Recognizer` guarantees that the `Grammar` will not change.

Finally, the `GRAMMAR_FINALIZED` event does not change the result's state. A `GRAMMAR_FINALIZED` event is issued only when a result is in the `UNFINALIZED` state, and leaves the result in that state.

6.7.8 Token Finalization

A result is a dynamic object a it is being recognized. One way in which a result can be dynamic is that tokens are updated and finalized as recognition of speech proceeds. The result events allow a recognizer to inform an application of changes in the either or both the finalized and unfinalized tokens of a result.

The finalized and unfinalized tokens can be updated on any of the following result event types: `RESULT_CREATED`, `RESULT_UPDATED`, `RESULT_ACCEPTED`, `RESULT_REJECTED`.

Finalized tokens are accessed through the `getBestTokens` and `getBestToken` methods of the `Result` interface. The unfinalized tokens are accessed through the `getUnfinalizedTokens` method of the `Result` interface. (See Section 6.7.3 on page 106 for details.)

A finalized token is a `ResultToken` in a `Result` that has been recognized in the incoming speech as matching a grammar. Furthermore, when a recognizer finalizes a token it indicates that it will not change the token at any point in the future. The `numTokens` method returns the number of finalized tokens.

Many recognizers do not finalize tokens until recognition of an entire result is complete. For these recognizers, the `numTokens` method returns zero for a result in the `UNFINALIZED` state.

For recognizers that do finalize tokens while a `Result` is in the `UNFINALIZED` state, the following conditions apply:

- The `Result` object may contain zero or more finalized tokens when the `RESULT_CREATED` event is issued.
- The recognizer issues `RESULT_UPDATED` events to the `ResultListener` during recognition each time one or more tokens are finalized.
- Tokens are finalized strictly in the order in which they are spoken (i.e., left to right in English text).

A result in the `UNFINALIZED` state may also have unfinalized tokens. An unfinalized token is a token that the recognizer has heard, but which it is not yet ready to finalize. Recognizers are not required to provide unfinalized tokens, and applications can safely choose to ignore unfinalized tokens.

For recognizers that provide unfinalized tokens, the following conditions apply:

- The `Result` object may contain zero or more unfinalized tokens when the `RESULT_CREATED` event is issued.
- The recognizer issues `RESULT_UPDATED` events to the `ResultListener` during recognition each time the unfinalized tokens change.
- For an unfinalized result, unfinalized tokens may be updated at any time and in any way. Importantly, the number of unfinalized tokens may increase, decrease or return to zero and the values of those tokens may change in any way the recognizer chooses.
- Unfinalized tokens always represent a guess for the speech following the finalized tokens.

Unfinalized tokens are highly changeable, so why are they useful? Many applications can provide users with visual feedback of unfinalized tokens — particularly for dictation results. This feedback informs users of the progress of the recognition and helps the user to know that something is happening. However, because these tokens may change and are more likely than finalized tokens to be incorrect, the applications should visually distinguish the unfinalized tokens by using a different font, different color or even a different window.

The following is an example of finalized tokens and unfinalized tokens for the sentence “I come from Australia”. The lines indicate the token values after the single `RESULT_CREATED` event, the multiple `RESULT_UPDATED` events and the final `RESULT_ACCEPTED` event. The finalized tokens are in bold, the unfinalized tokens are in italics.

1. `RESULT_CREATED`: *I come*
2. `RESULT_UPDATED`: *I come from*
3. `RESULT_UPDATED`: **I come** *from*
4. `RESULT_UPDATED`: **I come from** *a strange land*
5. `RESULT_UPDATED`: **I come from** *Australia*
6. `RESULT_ACCEPTED`: **I come from Australia**

Recognizers can vary in how they support finalized and unfinalized tokens in a number of ways. For an unfinalized result, a recognizer may provide finalized tokens, unfinalized tokens, both or neither. Furthermore, for a recognizer that does support finalized and unfinalized tokens during recognition, the behavior may depend upon the number of active grammars, upon whether the result is for a `RuleGrammar` or `DictationGrammar`, upon the length of spoken sentences, and upon other more complex factors. Fortunately, unless there is a functional requirement to display or otherwise process intermediate result, an application can safely ignore all but the `RESULT_ACCEPTED` event.

6.7.9 Finalized Rule Results

The are some common design patterns for processing accepted finalized results that match a `RuleGrammar`. First we review what we know about these results.

- It is safe to cast an accepted result that matches a `RuleGrammar` to the `FinalRuleResult` interface. It is safe to call any method of the `FinalRuleResult` interface or its parents: `FinalResult` and `Result`.
- The `getGrammar` method of the `Result` interface return a reference to the matched `RuleGrammar`. The `getRuleGrammar` method of the `FinalRuleResult` interface returns references to the `RuleGrammars` matched by the alternative guesses.
- The `getBestToken` and `getBestTokens` methods of the `Result` interface return the recognizer's best guess of what a user said.
- The `getAlternativeTokens` method returns alternative guesses for the entire result.
- The tags for the best guess are available from the `getTags` method of the `FinalRuleResult` interface.
- Result audio (see Section 6.7.11) and training information (see Section 6.7.12) are optionally available.

6.7.9.1 Result Tokens

A `ResultToken` in a result matching a `RuleGrammar` contains the same information as the `RuleToken` object in the `RuleGrammar` definition. This means that the tokenization of the result follows the tokenization of the grammar definition including compound tokens. For example, consider a grammar with the following Java Speech Grammar Format fragment which contains four tokens:

```
<rule> = I went to "San Francisco";
```

If the user says “I went to New York” then the result will contain the four tokens defined by JSGF: “I”, “went”, “to”, “San Francisco”.

The `ResultToken` interface defines more advanced information. Amongst that information the `getStartTime` and `getEndTime` methods may optionally return time-stamp values (or `-1` if the recognizer does not provide time-alignment information).

The `ResultToken` interface also defines several methods for a recognizer to provide presentation hints. Those hints are ignored for `RuleGrammar` results — they are only used for dictation results (see Section 6.7.10.2).

Furthermore, the `getSpokenText` and `getWrittenText` methods will return an identical string which is equal to the string defined in the matched grammar.

6.7.9.2 Alternative Guesses

In a `FinalRuleResult`, alternative guesses are alternatives for the entire result, that is, for a complete utterance spoken by a user. (A `FinalDictationResult` can provide alternatives for single tokens or sequences of tokens.) Because more than one `RuleGrammar` can be active at a time, an alternative token sequence may match a rule in a different `RuleGrammar` than the best guess tokens, or may match a different rule in the same `RuleGrammar` as the best guess. Thus, when processing alternatives for a `FinalRuleResult`, an application should use the `getRuleGrammar` and `getRuleName` methods to ensure that they analyze the alternatives correctly.

Alternatives are numbered from zero up. The 0^{th} alternative is actually the best guess for the result so `FinalRuleResult.getAlternativeTokens(0)` returns the same array as `Result.getBestTokens()`. (The duplication is for programming convenience.) Likewise, the `FinalRuleResult.getRuleGrammar(0)` call will return the same result as `Result.getGrammar()`.

The following code is an implementation of the `ResultListener` interface that processes the `RESULT_ACCEPTED` event. The implementation assumes that a `Result` being processed matches a `RuleGrammar`.

```
class MyRuleResultListener extends ResultAdapter
{
  public void resultAccepted(ResultEvent e)
  {
     // Assume that the result matches a RuleGrammar.
     // Cast the result (source of event) appropriately
     FinalRuleResult res = (FinalRuleResult) e.getSource();

     // Print out basic result information
```

```
            PrintStream out = System.out;
            out.println("Number guesses: " + res.getNumberGuesses());

            // Print out the best result and all alternatives
            for (int n=0; n < res.getNumberGuesses(); n++) {
                // Extract the n-best information
                String gname = res.getRuleGrammar(n).getName();
                String rname = res.getRuleName(n);
                ResultToken[] tokens = res.getAlternativeTokens(n);

                out.print("Alt " + n + ": ");
                out.print("<" + gname + "." + rname + "> :");
                for (int t=0; t < tokens.length; t++)
                    out.print(" " + tokens[t].getSpokenText());
                out.println();
            }
        }
    }
```

For a grammar with commands to control a windowing system (shown below), a result might look like:

```
Number guesses: 3
Alt 0: <com.acme.actions.command>: move the window to the back
Alt 1: <com.acme.actions.command>: move window to the back
Alt 2: <com.acme.actions.command>: open window to the front
```

If more than one grammar or more than one public rule was active, the `<grammarName.ruleName>` values could vary between the alternatives.

6.7.9.3 Result Tags

Processing commands generated from a `RuleGrammar` becomes increasingly difficult as the complexity of the grammar rises. With the Java Speech API, speech recognizers provide two mechanisms to simplify the processing of results: tags and parsing.

A tag is a label attached to an entity within a `RuleGrammar`. The Java Speech Grammar Format and the `RuleTag` class define how tags can be attached to a grammar. The following is a grammar for very simple control of windows which includes tags attached to the important words in the grammar.

```
grammar com.acme.actions;

public <command> = <action> <object> [<where>]
<action> = open {ACT_OP}| close {ACT_CL} | move {ACT_MV};
<object> = [a | an | the] (window {OBJ_WIN} | icon {OBJ_ICON});
<where> = [to the] (back {WH_BACK} | front {WH_FRONT});
```

This grammar allows users to speak commands such as

open window
move the *icon*
move the *window* to the *back*
move window back

The italicized words are the ones that are tagged in the grammar — these are the words that the application cares about. For example, in the third and fourth example commands, the spoken words are different but the tagged words are identical. Tags allow an application to ignore trivial words such as “the” and “to”.

The `com.acme.actions` grammar can be loaded and enabled using the code in the *“Hello World!”* example (on page 72). Since the grammar has a single public rule, `<command>`, the recognizer will listen for speech matching that rule, such as the example results given above.

The tags for the best result are available through the `getTags` method of the `FinalRuleResult` interface. This method returns an array of tags associated with the tokens (words) and other grammar entities matched by the result. If the best sequence of tokens is “move the window to the front”, the list of tags is the following `String` array:

```
String tags[] = {"ACT_MV", "OBJ_WIN", "WH_FRONT"};
```

Note how the order of the tags in the result is preserved (forward in time). These tags are easier for most applications to interpret than the original text of what the user said.

Tags can also be used to handle synonyms — multiple ways of saying the same thing. For example, “programmer”, “hacker”, “application developer” and “computer dude” could all be given the same tag, say “DEV”. An application that looks at the “DEV” tag will not care which way the user spoke the title.

Another use of tags is for *internationalization* of applications. Maintaining applications for multiple languages and locales is easier if the code is insensitive to the language being used. In the same way that the “DEV” tag isolated an application from different ways of saying “programmer”, tags can be used to provide an application with similar input irrespective of the language being recognized.

The following is a grammar for French with the same functionality as the grammar for English shown above.

```
grammar com.acme.actions.fr;

public <command> = <action> <object> [<where>]
<action> = ouvrir {ACT_OP}| fermer {ACT_CL} | deplacer {ACT_MV};
<object> = fenetre {OBJ_WIN} | icone {OBJ_ICON};
<where> = au-dessous {WH_BACK} | au-dessus {WH_FRONT};
```

For this simple grammar, there are only minor differences in the structure of the grammar (e.g. the `"[to the]"` tokens in the `<where>` rule for English are absent in French). However, in more complex grammars the syntactic differences between languages become significant and tags provide a clearer improvement.

Tags do not completely solve internationalization problems. One issue to be considered is word ordering. A simple command like “open the window” can translate to the form “the window open” in some languages. More complex sentences can have more complex transformations. Thus, applications need to be aware of word ordering, and thus tag ordering when developing international applications.

6.7.9.4 Result Parsing

More advanced applications *parse* results to get even more information than is available with tags. Parsing is the capability to analyze how a sequence of tokens matches a `RuleGrammar`. Parsing of text against a `RuleGrammar` is discussed in Section 6.5.5 (page 96).

Parsing a `FinalRuleResult` produces a `RuleParse` object. The `getTags` method of a `RuleParse` object provides the same tag information as the `getTags` method of a `FinalRuleResult`. However, the `FinalRuleResult` provides tag information for only the best-guess result, whereas parsing can be applied to the alternative guesses.

An API requirement that simplifies parsing of results that match a `RuleGrammar` is that for a such result to be `ACCEPTED` (not rejected) it must exactly match the grammar — technically speaking, it must be possible to parse a `FinalRuleResult` against the `RuleGrammar` it matches. This is not guaranteed, however, if the result was rejected or if the `RuleGrammar` has been modified since it was committed and produced the result.

6.7.10 Finalized Dictation Results

The are some common design patterns for processing accepted finalized results that match a `DictationGrammar`. First we review what we know about these results.

- It is safe to cast an accepted result that matches a `DictationGrammar` to the `FinalDictationResult` interface. It is safe to call any method of the `FinalDictationResult` interface or its parents: `FinalResult` and `Result`.
- The `getGrammar` method of the `Result` interface return a reference to the matched `DictationGrammar`.
- The `getBestToken` and `getBestTokens` methods of the `Result` interface return the recognizer's best guess of what a user said.
- The `getAlternativeTokens` method of the `FinalDictationResult` interface returns alternative guesses for any token or sequence of tokens.
- Result audio (see Section 6.7.11) and training information (see Section 6.7.12) are optionally available.

The `ResultTokens` provided in a `FinalDictationResult` contain specialized information that includes hints on textual presentation of tokens. Section 6.7.10.2 (on page 121) discusses the presentation hints in detail. In this section the methods for obtaining and using alternative tokens are described.

6.7.10.1Alternative Guesses

Alternative tokens for a dictation result are most often used by an application for display to users for correction of dictated text. A typical scenario is that a user speaks some text — perhaps a few words, a few sentences, a few paragraphs or more. The user reviews the text and detects a recognition error. This means that the best guess token sequence is incorrect. However, very often the correct text is one of the top alternative guesses. Thus, an application will provide a user the ability to review a set of alternative guesses and to select one of them if it is the correct text. Such a correction mechanism is often more efficient than typing the correction or dictating the text again. If the correct text is not amongst the alternatives an application must support other means of entering the text.

The `getAlternativeTokens` method is passed a starting and an ending `ResultToken`. These tokens must have been obtained from the same result either through a call to `getBestToken` or `getBestTokens` in the `Result` interface, or through a previous call to `getAlternativeTokens`.

```
ResultToken[][] getAlternativeTokens(
                  . ResultToken fromToken,
                  . ResultToken toToken,
                  . int max);
```

To obtain alternatives for a single token (rather than alternatives for a sequence), set `toToken` to `null`.

The `int` parameter allows the application to specify the number of alternatives it wants. The recognizer may choose to return any number of alternatives up to the maximum number including just one alternative (the original token sequence). Applications can indicate in advance the number of alternatives it may request by setting the `NumResultAlternatives` parameter through the recognizer's `RecognizerProperties` object.

The two-dimensional array returned by the `getAlternativeTokens` method is the most difficult aspect of dictation alternatives to understand. The following example illustrates the major features of the return value.

Let's consider a dictation example where the user says "he felt alienated today" but the recognizer hears "he felt alien ate Ted today". The user says four words but the recognizer hears six words. In this example, the boundaries of the spoken words and best-guess align nicely: "alienated" aligns with "alien ate Ted" (incorrect tokens don't always align smoothly with the correct tokens).

Users are typically better at locating and fixing recognition errors than recognizers or applications — they provided the original speech. In this example, the user will likely identify the words "alien ate Ted" as incorrect (tokens 2 to 4 in the best-guess result). By an application-provided method such as selection by mouse and a pull-down menu, the user will request alternative guesses for the three incorrect tokens. The application calls the `getAlternativeTokens` method of the `FinalDictationResult` to obtain the recognizer's guess at the alternatives.

```
// Get 6 alternatives for for tokens 2 through 4.
FinalDictationResult r = ...;
ResultToken tok2 = r.getBestToken(2);
ResultToken tok4 = r.getBestToken(4);
String[][] alt = r.getAlternativeTokens(tok2, tok4, 6);
```

The return array might look like the following. Each line represents a sequence of alternative tokens to "alien ate Ted". Each word in each alternative sequence represents a `ResultToken` object in an array.

```
alt[0] = alien ate Ted   // the best guess
alt[1] = alienate Ted    // the 1st alternative
```

```
alt[2] = alienated       // the 2nd alternative
alt[3] = alien hated     // the 3rd alternative
alt[4] = a lion ate Ted  // the 4th alternative
```

The points to note are:

- ♦ The first alternative is the best guess. This is usually the case if the `toToken` and `fromToken` values are from the best-guess sequence. (From an user perspective it's not really an alternative.)
- ♦ Only five alternative sequences were returned even though six were requested. This is because a recognizer will only return alternatives it considers to reasonable guesses. It is legal for this call to return only the best guess with no alternatives if can't find any reasonable alternatives.
- ♦ The number of tokens is not the same in all the alternative sequences (3, 2, 1, 2, 4 tokens respectively). This return array is known as a *ragged array*. From a speech perspective is easy to see why different lengths are needed, but application developers do need to be careful processing a ragged array.
- ♦ The best-guess and the alternatives do not always make sense to humans.

A complex issue to understand is that the alternatives vary according to how the application (or user) requests them. The 1st alternative to "alien ate Ted" is "alienate Ted". However, the 1st alternative to "alien" might be "a lion", the 1st alternative to "alien ate" might be "alien eight", and the 1st alternative to "alien ate Ted today" might be "align ate Ted to day".

Fortunately for application developers, users learn to select sequences that are likely to give reasonable alternatives, and recognizers are developed to make the alternatives as useful and accurate as possible.

6.7.10.2Result Tokens

A `ResultToken` object represents a single token in a result. A token is most often a single word, but multi-word tokens are possible (e.g., "New York") as well as formatting characters and language-specific constructs. For a `DictationGrammar` the set of tokens is built into the recognizer.

Each `ResultToken` in a `FinalDictationResult` provides the following information.

- ♦ The *spoken form* of the token which provides a transcript of what the user says (`getSpokenText` method). In a dictation system, the spoken form is typically used when displaying unfinalized tokens.

- The *written form* of the token which indicates how to visually present the token (`getWrittenText` method). In a dictation system, the written form of finalized tokens is typically placed into the text edit window after applying the following presentation hints.
- A *capitalization hint* indicating whether the written form of the following token should be capitalized (first letter only), all uppercase, all lowercase, or left as-is (`getCapitalizationHint` method).
- An *spacing hint* indicating how the written form should be spaced with the previous and following tokens.

The presentation hints in a `ResultToken` are important for the processing of dictation results. Dictation results are typically displayed to the user, so using the written form and the capitalization and spacing hints for formatting is important. For example, when dictation is used in word processing, the user will want the printed text to be correctly formatted.

The capitalization hint indicates how the written form of the following token should be formatted. The capitalization hint takes one of four mutually exclusive values. `CAP_FIRST` indicates that the first character of the following token should be capitalized. The `UPPERCASE` and `LOWERCASE` values indicate that the following token should be either all uppercase or lowercase. `CAP_AS_IS` indicates that there should be no change in capitalization of the following token.

The spacing hint deals with spacing around a token. It is an `int` value containing three flags which are or'ed together (using the '|' operator). If none of the three spacing hint flags are set true, then `getSpacingHint` method returns the value `SEPARATE` which is the value zero.

- The `ATTACH_PREVIOUS` bit is set if the token should be attached to the previous token: no space between this token and the previous token. In English, some punctuation characters have this flag set true. For example, periods, commas and colons are typically attached to the previous token.
- The `ATTACH_FOLLOWING` bit is set if the token should be attached to the following token: no space between this token and the following token. For example, in English, opening quotes, opening parentheses and dollar signs typically attach to the following token.
- The `ATTACH_GROUP` bit is set if the token should be attached to previous or following tokens if they also have the `ATTACH_GROUP` flag set to true. In other words, tokens in an attachment group should be attached together. In English, a common use of the group flag is for numbers, digits and currency amounts. For example, the sequence of four spoken-form tokens,

`"3" "point" "1" "4"`, should have the group flag set true, so the presentation form should not have separating spaces: `"3.14"`.

Every language has conventions for textual representation of a spoken language. Since recognizers are language-specific and understand many of these presentation conventions, they provide the presentation hints (written form, capitalization hint and spacing hint) to simplify applications. However, applications may choose to override the recognizer's hints or may choose to do additional processing.

Table 6-6 shows examples of tokens in which the spoken and written forms are different:

Table 6-6 Spoken and written forms for some English tokens

Spoken Form	Written Form	Capitalization	Spacing
twenty	`20`	`CAP_AS_IS`	`SEPARATE`
new line	`'\n' '\u000A'`	`CAP_FIRST`	`ATTACH_PREVIOUS & ATTACH_FOLLOWING`
new paragraph	`'\u2029'`	`CAP_FIRST`	`ATTACH_PREVIOUS & ATTACH_FOLLOWING`
no space	`null`	`CAP_AS_IS`	`ATTACH_PREVIOUS & ATTACH_FOLLOWING`
Space bar	`' ' '\u0020'`	`CAP_AS_IS`	`ATTACH_PREVIOUS & ATTACH_FOLLOWING`
Capitalize next	`null`	`CAP_FIRST`	`SEPARATE`
Period	`'.' '\u002E'`	`CAP_FIRST`	`ATTACH_PREVIOUS`
Comma	`',' '\u002C'`	`CAP_AS_IS`	`ATTACH_PREVIOUS`
Open parentheses	`'(' '\u0028'`	`CAP_AS_IS`	`ATTACH_FOLLOWING`
Exclamation mark	`'!' '\u0021'`	`CAP_FIRST`	`ATTACH_PREVIOUS`

Table 6-6 Spoken and written forms for some English tokens (cont'd)

Spoken Form	Written Form	Capitalization	Spacing
dollar sign	`'$' '\u0024'`	`CAP_AS_IS`	`ATTACH_FOLLOWING & ATTACH_GROUP`
pound sign	`'£' '\u00A3'`	`CAP_AS_IS`	`ATTACH_FOLLOWING & ATTACH_GROUP`
yen sign	`'¥' '\u00A5'`	`CAP_AS_IS`	`ATTACH_PREVIOUS & ATTACH_GROUP`

"New line", "new paragraph", "space bar", "no space" and "capitalize next" are all examples of conversion of an implicit command (e.g. "start a new paragraph"). For three of these, the written form is a single Unicode character. Most programmers are familiar with the new-line character '\n' and space ' ', but fewer are familiar with the Unicode character for new paragraph '\u2029'. For convenience and consistency, the `ResultToken` includes static variables called `NEW_LINE` and `NEW_PARAGRAPH`.

Some applications will treat a paragraph boundary as two new-line characters, others will treat it differently. Each of these commands provides hints for capitalization. For example, in English the first letter of the first word of a new paragraph is typically capitalized.

The punctuation characters, "period", "comma", "open parentheses", "exclamation mark" and the three currency symbols convert to a single Unicode character and have special presentation hints.

An important feature of the written form for most of the examples is that the application does not need to deal with synonyms (multiple ways of saying the same thing). For example, "open parentheses" may also be spoken as "open paren" or "begin paren" but in all cases the same written form is generated.

The following is an example sequence of result tokens.

Table 6-7 Sample sequence of result tokens

Spoken Form	Written Form	Capitalization	Spacing
new line	`"\n"`	`CAP_FIRST`	`ATTACH_PREVIOUS & ATTACH_FOLLOWING`

Table 6-7 Sample sequence of result tokens (cont'd)

Spoken Form	Written Form	Capitalization	Spacing
the	`"the"`	`CAP_AS_IS`	`SEPARATE`
uppercase next	`null`	`UPPERCASE`	`SEPARATE`
index	`"index"`	`CAP_AS_IS`	`SEPARATE`
is	`"is"`	`CAP_AS_IS`	`SEPARATE`
seven	`"7"`	`CAP_AS_IS`	`ATTACH_GROUP`
dash	`"-"`	`CAP_AS_IS`	`ATTACH_GROUP`
two	`"2"`	`CAP_AS_IS`	`ATTACH_GROUP`
period	`"."`	`CAP_FIRST`	`ATTACH_PREVIOUS`

This sequence of tokens should be converted to the following string:

```
"\nThe INDEX is 7-2."
```

Conversion of spoken text to a written form is a complex task and is complicated by the different conventions of different languages and often by different conventions for the same language. The spoken form, written form and presentation hints of the `ResultToken` interface handle most simple conversions. Advanced applications should consider filtering the results to process more complex patterns, particularly cross-token patterns. For example "nineteen twenty eight" is typically converted to "1928" and "twenty eight dollars" to "$28" (note the movement of the dollar sign to before the numbers).

6.7.11 Result Audio

If requested by an application, some recognizers can provide audio data for results. Audio data has a number of uses. In dictation applications, providing audio feedback to users aids correction of text because the audio reminds users of what they said (it's not always easy to remember exactly what you dictate, especially in long sessions). Audio data also allows storage for future evaluation and debugging.

Audio data is provided for finalized results through the following methods of the `FinalResult` interface.

Table 6-8 FinalResult interface: audio methods

Name	Description
`getAudio`	Get an `AudioClip` for a token, a sequence of tokens or for an entire result.
`isAudioAvailable`	Tests whether audio data is available for a result.
`releaseAudio`	Release audio data for a result.

There are two `getAudio` methods in the `FinalResult` interface. One method accepts no parameters and returns an `AudioClip` for an entire result or `null` if audio data is not available for this result. The other `getAudio` method takes a start and end `ResultToken` as input and returns an `AudioClip` for the segment of the result including the start and end token or `null` if audio data is not available.

In both forms of the `getAudio` method, the recognizer will attempt to return the specified audio data. However, it is not always possible to exactly determine the start and end of words or even complete results. Sometimes segments are "clipped" and sometimes surrounding audio is included in the `AudioClip`.

Not all recognizers provide access to audio for results. For recognizers that do provide audio data, it is not necessarily provided for all results. For example, a recognizer might only provide audio data for dictation results. Thus, applications should always check for a null return value on a `getAudio` call.

The storage of audio data for results potentially requires large amounts of memory, particularly for long sessions. Thus, result audio requires special management. An application that wishes to use result audio should:

- Set the `ResultAudioProvided` parameter of `RecognizerProperties` to `true`. Recognizers that do not support audio data ignore this call.
- Test the availability of audio for a result using the `isAudioAvailable` method of the `FinalResult` interface.
- Use the `getAudio` methods to obtain audio data. These methods return `null` if audio data is not available.

- ♦ Once the application has finished use of the audio for a `Result`, it should call the `releaseAudio` method of `FinalResult` to free up resources.

A recognizer may choose to release audio data for a result if it is necessary to reclaim memory or other system resources.

When audio is released by either a call to `releaseAudio` or by the recognizer a `AUDIO_RELEASED` event is issued to the `audioReleased` method of the `ResultListener`.

6.7.12 Result Correction

Recognition results are not always correct. Some recognizers can be trained by informing of the correct tokens for a result — usually when a user corrects a result.

Recognizers are not required to support correction capabilities. If a recognizer does support correction, it does not need to support correction for every result. For example, some recognizers support correction only for dictation results.

Applications are not required to provide recognizers with correction information. However, if the information is available to an application and the recognizer supports correction then it is good practice to inform the recognizer of the correction so that it can improve its future recognition performance.

The `FinalResult` interface provides the methods that handle correction.

Table 6-9 FinalResult interface: correction methods

Name	Description
`tokenCorrection`	Inform the recognizer of a correction in which zero or more tokens replace a token or sequence of tokens.

`MISRECOGNITION`
`USER_CHANGE`
`DONT_KNOW`
	Indicate the type of correction.

`isTrainingInfoAvailable`	Tests whether the recognizer has information available to allow it to learn from a correction.
`releaseTrainingInfo`	Release training information for a result.

Often, but certainly not always, a correction is triggered when a user corrects a recognizer by selecting amongst the alternative guesses for a result. Other instances when an application is informed of the correct result are when the user types a correction to dictated text, or when a user corrects a misrecognized command with a follow-up command.

Once an application has obtained the correct result text, it should inform the recognizer. The correction information is provided by a call to the `tokenCorrection` method of the `FinalResult` interface. This method indicates a correction of one token sequence to another token sequence. Either token sequence may contain one or more tokens. Furthermore, the correct token sequence may contain zero tokens to indicate deletion of tokens.

The `tokenCorrection` method accepts a `correctionType` parameter that indicates the reason for the correction. The legal values are defined by constants of the `FinalResult` interface:

- `MISRECOGNITION` indicates that the new tokens are known to be the tokens actually spoken by the user: a correction of a recognition error. Applications can be confident that a selection of an alternative token sequence implies a `MISRECOGNITION` correction.
- `USER_CHANGE` indicates that the new tokens are not the tokens originally spoken by the user but instead the user has changed his/her mind. This is a “speako” (a spoken version of a “typo”). A `USER_CHANGE` may be indicated if a user types over the recognized result, but sometimes the user may choose to type in the correct result.
- `DONT_KNOW` the application does not know whether the new tokens are correcting a recognition error or indicating a change by the user. Applications should indicate this type of correction whenever unsure of the type of correction.

Why is it useful to tell a recognizer about a `USER_CHANGE`? Recognizers adapt to both the sounds and the patterns of words of users. A `USER_CHANGE` correction allows the recognizer to learn about a user’s word patterns. A `MISRECOGNITION` correction allows the recognizer to learn about both the user’s voice and the word patterns. In both cases, correcting the recognizer requests it to re-train itself based on the new information.

Training information needs to be managed because it requires substantial memory and possibly other system resources to maintain it for a result. For example, in long dictation sessions, correction data can begin to use excessive amounts of memory.

Recognizers maintain training information only when the recognizer’s `TrainingProvided` parameter is set to true through the `RecognizerProperties` interface. Recognizers that do not support correction will ignore calls to the `setTrainingProvided` method.

If the `TrainingProvided` parameter is set to true, a result may include training information when it is finalized. Once an application believes the training information is no longer required for a specific `FinalResult`, it should call the `releaseTrainingInfo` method of `FinalResult` to indicate the recognizer can release the resources used to store the information.

At any time, the availability of training information for a result can be tested by calling the `isTrainingInfoAvailable` method.

Recognizers can choose to release training information even without a request to do so by the application. This does not substantially affect an application because performing correction on a result which does not have training information is not an error.

A `TRAINING_INFO_RELEASED` event is issued to the `ResultListener` when the training information is released. The event is issued identically whether the application or recognizer initiated the release.

6.7.13 Rejected Results

First, a warning: *ignore rejected results unless you really understand them!*

Like humans, recognizers don't have perfect hearing and so they make mistakes (recognizers still tend to make more mistakes than people). An application should never completely trust a recognition result. In particular, applications should treat important results carefully, for example, "delete all files".

Recognizers try to determine whether they have made a mistake. This process is known as *rejection*. But recognizers also make mistakes in rejection! In short, a recognizer cannot always tell whether or not it has made a mistake.

A recognizer may reject incoming speech for a number of reasons:

- Detected a non-speech event (e.g. cough, laughter, microphone click).
- Detected speech that only partially matched an active grammar (e.g. user spoke only half a command).
- Speech contained "um", "ah", or some other speaking error that the recognizer could not ignore.
- Speech matched an active grammar but the recognizer was not confident that it was an accurate match.

Rejection is controlled by the `ConfidenceLevel` parameter of `RecognizerProperties` (see Section 6.8). The confidence value is a floating point number between 0.0 and 1.0. A value of 0.0 indicates weak rejection — the recognizer doesn't need to be very confident to accept a result. A value of 1.0 indicates strongest rejection, implying that the recognizer will reject a result unless it is very confident that the result is correct. A value of 0.5 is the recognizer's default.

6.7.13.1Rejection Timing

A result may be rejected with a `RESULT_REJECTED` event at any time while it is `UNFINALIZED`: that is, any time after a `RESULT_CREATED` event but without a `RESULT_ACCEPTED` event occurring. (For a description of result events see Section 6.7.4.)

This means that the sequence of result events that produce a `REJECTED` result:

- A single `RESULT_CREATED` event to issue a new result in the `UNFINALIZED` state.
- While in the `UNFINALIZED` state, zero or more `RESULT_UPDATED` events may be issued to update finalized and/or unfinalized tokens. Also, a single optional `GRAMMAR_FINALIZED` event may be issued to indicate that the matched grammar has been identified.
- A single `RESULT_REJECTED` event moves the result to the `REJECTED` state.

When a result is rejected, there is a strong probability that the information about a result normally provided through `Result`, `FinalResult`, `FinalRuleResult` and `FinalDictationResult` interfaces is inaccurate, or more typically, not available.

Some possibilities that an application must consider:

- There are no finalized tokens (`numTokens` returns 0).
- The `GRAMMAR_FINALIZED` event was not issued, so the `getGrammar` method returns `null`. In this case, all the methods of the `FinalRuleResult` and `FinalDictationResult` interfaces throw exceptions.
- Audio data and training information may be unavailable, even when requested.
- All tokens provided as best guesses or alternative guesses may be incorrect.
- If the result does match a `RuleGrammar`, there is not a guarantee that the tokens can be parsed successfully against the grammar.

Finally, a repeat of the warning. Only use rejected results if you really know what you are doing!

6.7.14 Result Timing

Recognition of speech is not an instant process. There are intrinsic delays between the time the user starts or ends speaking a word or sentence and the time at which the corresponding result event is issued by the speech recognizer.

The most significant delay for most applications is the time between when the user stops speaking and the `RESULT_ACCEPTED` or `RESULT_REJECTED` event that indicates the recognizer has finalized the result.

The minimum finalization time is determined by the `CompleteTimeout` parameter that is set through the `RecognizerProperties` interface. This time-out indicates the period of silence after speech that the recognizer should process before finalizing a result. If the time-out is too long, the response of the recognizer (and the application) is unnecessarily delayed. If the time-out is too short, the recognizer may inappropriately break up a result (e.g. finalize a result while the user is taking a quick breath). Typically values are less than a second, but not usually less than 0.3sec.

There is also an `IncompleteTimeout` parameter that indicates the period of silence a recognizer should process if the user has said something that may only partially matches an active grammar. This time-out indicates how long a recognizer should wait before rejecting an incomplete sentence. This time-out also indicates how long a recognizer should wait mid-sentence if a result could be accepted, but could also be continued and accepted after more words. The `IncompleteTimeout` is usually longer than the complete time-out.

Latency is the overall delay between a user finishing speaking and a result being produced. There are many factors that can affect latency. Some effects are temporary, others reflect the underlying design of speech recognizers. Factors that can increase latency include:

- The `CompleteTimeout` and `IncompleteTimeout` properties discussed above.
- *Computer power* (especially CPU speed and memory): less powerful computers may process speech slower than real-time. Most systems try to catch up while listening to background silence (which is easier to process than real speech).
- *Grammar complexity*: larger and more complex grammars tend to require more time to process. In most cases, rule grammars are processed more quickly than dictation grammars.
- *Suspending*: while a recognizer is in the `SUSPENDED` state, it must buffer of incoming audio. When it returns to the `LISTENING` state it must catch up by processing the buffered audio. The longer the recognizer is suspended, the longer it can take to catch up to real time and the more latency increases.

- *Client/server latencies*: in client/server architectures, communication of the audio data, results, and other information between the client and server can introduce delays.

6.7.15 Storing Results

Result objects can be stored for future processing. This is particularly useful for dictation applications in which the correction information, audio data and alternative token information is required in future sessions on the same document because that stored information can assist document editing.

The `Result` object is recognizer-specific. This is because each recognizer provides an implementation of the `Result` interface. The implications are that (a) recognizers do not usually understand each other's results, and (b) a special mechanism is required to store and load result objects (standard Java object serialization is not sufficient).

The `Recognizer` interface defines the methods `writeVendorResult` and `readVendorResult` to perform this function. These methods write to an `OutputStream` and read from an `InputStream` respectively. If the correction information and audio data for a result are available, then they will be stored by this call. Applications that do not need to store this extra data should explicitly release it before storing a result.

```
{
   Recognizer rec;
   OutputStream stream;
   Result result;
   ...
   try {
      rec.writeVendorResult(stream, result);
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

A limitation of storing vendor-specific results is that a compatible recognizer must be available to read the file. Applications that need to ensure a file containing a result can be read, even if no recognizer is available, should wrap the result data when storing it to the file. When re-loading the file at a later time, the application will unwrap the result data and provide it to a recognizer only if a suitable recognizer is available. One way to perform the wrapping is to provide the `writeVendorResult` method with a `ByteArrayOutputStream` to temporarily place the result in a byte array before storing to a file.

6.8 Recognizer Properties

A speech engine has both persistent and run-time adjustable properties. The persistent properties are defined in the `RecognizerModeDesc` which includes properties inherited from `EngineModeDesc` (see Section 4.2 on page 36). The persistent properties are used in the selection and creation of a speech recognizer. Once a recognizer has been created, the same property information is available through the `getEngineModeDesc` method of a `Recognizer` (inherited from the `Engine` interface).

A recognizer also has seven run-time adjustable properties. Applications get and set these properties through `RecognizerProperties` which extends the `EngineProperties` interface. The `RecognizerProperties` for a recognizer are provided by the `getEngineProperties` method that the `Recognizer` inherits from the `Engine` interface. For convenience a `getRecognizerProperties` method is also provided in the `Recognizer` interface to return a correctly cast object.

The get and set methods of `EngineProperties` and `RecognizerProperties` follow the JavaBeans conventions with the form:

```
Type getPropertyName();
void setPropertyName(Type);
```

A recognizer can choose to ignore unreasonable values provided to a set method, or can provide upper and lower bounds.

Table 6-10 Run-time Properties of a Recognizer

Property	**Description**
`ConfidenceLevel`	`float` value in the range 0.0 to 1.0. Results are rejected if the engine is not confident that it has correctly determined the spoken text. A value of 1.0 requires a recognizer to have maximum confidence in every result so more results are likely to be rejected. A value of 0.0 requires low confidence indicating fewer rejections. 0.5 is the recognizer's default.

Table 6-10 Run-time Properties of a Recognizer (cont'd)

Property	Description
`Sensitivity`	`float` value between 0.0 and 1.0. A value of 0.5 is the default for the recognizer. 1.0 gives maximum sensitivity, making the recognizer sensitive to quiet input but more sensitive to noise. 0.0 gives minimum sensitivity, requiring the user to speak loudly and making the recognizer less sensitive to background noise. *Note*: some recognizers set the gain automatically during use, or through a setup "Wizard". On these recognizers the sensitivity adjustment should be used only in cases where the automatic settings are not adequate.
`SpeedVsAccuracy`	`float` value between 0.0 and 1.0. 0.0 provides the fastest response. 1.0 maximizes recognition accuracy. 0.5 is the default value for the recognizer which the manufacturer determines as the best compromise between speed and accuracy.
`CompleteTimeout`	`float` value in seconds that indicates the minimum period between when a speaker stops speaking (silence starts) and the recognizer finalizing a result. The complete time-out is applied when the speech prior to the silence matches an active grammar (c.f. `IncompleteTimeout`). A long complete time-out value delays the result and makes the response slower. A short time-out may lead to an utterance being broken up inappropriately (e.g. when the user takes a breath). Complete time-out values are typically in the range of 0.3 seconds to 1.0 seconds.

Table 6-10 Run-time Properties of a Recognizer (cont'd)

Property	Description
`IncompleteTimeout`	`float` value in seconds that indicates the minimum period between when a speaker stops speaking (silence starts) and the recognizer finalizing a result. The incomplete time-out is applied when the speech prior to the silence does not match an active grammar (c.f. `CompleteTimeout`). In effect, this is the period the recognizer will wait before rejecting an incomplete utterance. The `IncompleteTimeout` is typically longer than the `CompleteTimeout`.
`ResultNumAlternatives`	`integer` value indicating the preferred maximum number of N-best alternatives in `FinalDictationResult` and `FinalRuleResult` objects (see Section 6.7.9). Returning alternatives requires additional computation. Recognizers do not always produce the maximum number of alternatives (for example, because some alternatives are rejected), and the number of alternatives may vary between results and between tokens. A value of 0 or 1 requests that no alternatives be provided — only a best guess.
`ResultAudioProvided`	`boolean` value indicating whether the application wants the recognizer to audio with `FinalResult` objects. Recognizers that do provide result audio can ignore this call. (See *Result Audio* on page 125 for details.)
`TrainingProvided`	`boolean` value indicating whether the application wants the recognizer to support training with `FinalResult` objects.

6.9 Speaker Management

A `Recognizer` may, optionally, provide a `SpeakerManager` object. The `SpeakerManager` allows an application to manage the `SpeakerProfiles` of that `Recognizer`. The `SpeakerManager` for is obtained through `getSpeakerManager` method of the `Recognizer` interface. Recognizers that do not maintain speaker profiles — known as speaker-independent recognizers — return `null` for this method.

A `SpeakerProfile` object represents a single enrollment to a recognizer. One user may have multiple `SpeakerProfiles` in a single recognizer, and one recognizer may store the profiles of multiple users.

The `SpeakerProfile` class is a reference to data stored with the recognizer. A profile is identified by three values all of which are `String` objects:

- `id`: A unique identifier for a profile (per-recognizer unique). The string may be automatically generated but should be printable.
- `name`: An identifier for a user. This may be an account name or any other name that could be entered by a user.
- `variant`: The variant identifies a particular enrollment of a user and becomes useful when one user has more than one enrollment.

The `SpeakerProfile` object is a handle to all the stored data the recognizer has about a speaker in a particular enrollment. Except for the three values defined above, the speaker data stored with a profile is internal to the recognizer.

Typical data stored by a recognizer with the profile might include:

- *Full speaker data*: Full name, age, gender and so on.
- *Speaker preferences*: Settings such as those provided through the `RecognizerProperties` (see Section 6.8).
- *Language models*: Data about the words and word patterns of the speaker.
- *Word models*: Data about the pronunciation of words by the speaker.
- *Acoustic models*: Data about the speaker's voice and speaking style.
- *History*: Records of previous training information and usage history

The primary role of stored profiles is in maintaining information that enables a recognition to adapt to characteristics of the speaker. The goal of this adaptation is to improve the performance of the speech recognizer including both recognition accuracy and speed.

The `SpeakerManager` provides management of all the profiles stored in the recognizer. Most often, the functionality of the `SpeakerManager` is used as a direct consequence of user actions, typically by providing an enrollment window to the user. The functionality provided includes:

- *Current speaker*: The `getCurrentSpeaker` and `setCurrentSpeaker` methods determine which speaker profile is currently being used to recognize incoming speech.
- *Listing profiles*: The `listKnownSpeakers` method returns an array of all the `SpeakerProfiles` known to the recognizer. A common procedure is to display that list to a user to allow the user to select a profile.
- *Creation and deletion*: The `newSpeakerProfile` and `newSpeakerProfile` methods create a new profile or delete a profile in the recognizer.
- *Read and write*: The `readVendorSpeakerProfile` and `writeVendorSpeakerProfile` methods allow a speaker profile and all the recognizer's associated data to be read from or stored to a `Stream`. The data format will typically be proprietary.
- *Save and revert*: During normal operation, a recognizer will maintain and update the speaker profile as new information becomes available. Some of the events that may modify the profile include changing the `RecognizerProperties`, making a correction to a result, producing any result that allows the recognizer to adapt its models, and more many activities. It is normal to save the updated profile at the end of any session by calling `saveCurrentSpeakerProfile`. In some cases, however, a user's data may be corrupted (e.g., because they loaned their computer to another user). In this case, the application may be requested by a user to revert the profile to the last stored version by calling `revertCurrentSpeaker`.
- *Display component*: The `getControlComponent` method optionally returns an AWT `Component` object that can be displayed to a user. If supported, this component should expose the vendor's speaker management capabilities which may be more detailed than those provided by the `SpeakerManager` interface. The vendor functionality may also be proprietary.

An individual speaker profile may be large (perhaps several MByte) so storing, loading, creating and otherwise manipulating these objects can be slow.

The `SpeakerManager` is one of the capabilities of a `Recognizer` that is available in the deallocated state. The purpose is to allow an application to indicate the initial speaker profile to be loaded when the recognizer is allocated. To achieve this, the `listKnownSpeakers`, `getCurrentSpeaker` and `setCurrentSpeaker` methods can be called before calling the `allocate` method.

To facilitate recognizer selection, the list of speaker profiles is also a property of a recognizer presented through the `RecognizerModeDesc` class. This allows an application to select a recognizer that has already been trained by a user, if one is available.

In most cases, a `Recognizer` persistently restores the last used speaker profile when allocating a recognizer, unless asked to do otherwise.

6.10 Recognizer Audio

The current audio functionality of the Java Speech API is incompletely specified. Once a standard mechanism is established for streaming input and output audio on the Java platform the API will be revised to incorporate that functionality.

In this release of the API, the only established audio functionality is provided through the `RecognizerAudioListener` interface and the `RecognizerAudioEvent` class. Audio events issued by a recognizer are intended to support simple feedback mechanisms for a user. The three types of `RecognizerAudioEvent` are as follows:

- `SPEECH_STARTED` and `SPEECH_STOPPED`: These events are issued when possible speech input is detected in the audio input stream. These events are usually based on a crude mechanism for speech detection so a `SPEECH_STARTED` event is not always followed by output of a result. Furthermore, one `SPEECH_STARTED` may be followed by multiple results, and one result might cover multiple `SPEECH_STARTED` events.
- `AUDIO_LEVEL`: This event is issued periodically to indicate the volume of audio input to the recognizer. The level is a `float` and varies on a scale from 0.0 to 1.0: silence to maximum volume. The audio level is often displayed visually as a “VU Meter” — the scale on a stereo system that goes up and down with the volume.

All the `RecognizerAudioEvents` are produced as audio reaches the input to the recognizer. Because recognizers use internal buffers between audio input and the recognition process, the audio events can run ahead of the recognition process.